> `ON UPDATE` clauses to be ignored. For these reasons, specifying `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. `InnoDB` essentially implements the semantics defined by `MATCH SIMPLE`, which permit a foreign key to be all or partially `NULL`. In that case, the (child table) row containing such a foreign key is permitted to be inserted, and does not match any row in the referenced (parent) table. It is possible to implement other semantics using triggers.
>
> Additionally, MySQL requires that the referenced columns be indexed for performance. However, `InnoDB` does not enforce any requirement that the referenced columns be declared `UNIQUE` or `NOT NULL`. 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 keys that are both `UNIQUE` (or `PRIMARY`) and `NOT NULL`.
>
> 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. For more information, see Section 1.6.2.3, "FOREIGN KEY Constraint Differences".

- *`reference_option`*

  For information about the `RESTRICT`, `CASCADE`, `SET NULL`, `NO ACTION`, and `SET DEFAULT` options, see Section 13.1.20.5, "FOREIGN KEY Constraints".

Table Options

Table options are used to optimize the behavior of the table. In most cases, you do not have to specify any of them. These options apply to all storage engines unless otherwise indicated. Options that do not apply to a given storage engine may be accepted and remembered as part of the table definition. Such options then apply if you later use `ALTER TABLE` to convert the table to use a different storage engine.

- `ENGINE`

  Specifies the storage engine for the table, using one of the names shown in the following table. The engine name can be unquoted or quoted. The quoted name `'DEFAULT'` is recognized but ignored.

Storage Engine	Description
`InnoDB`	Transaction-safe tables with row locking and foreign keys. The default storage engine for new tables. See Chapter 15, *The InnoDB Storage Engine*, and in particular Section 15.1, "Introduction to InnoDB" if you have MySQL experience but are new to `InnoDB`.
`MyISAM`	The binary portable storage engine that is primarily used for read-only or read-mostly workloads. See Section 16.2, "The MyISAM Storage Engine".
`MEMORY`	The data for this storage engine is stored only in memory. See Section 16.3, "The MEMORY Storage Engine".

Storage Engine	Description
CSV	Tables that store rows in comma-separated values format. See Section 16.4, “The CSV Storage Engine”.
ARCHIVE	The archiving storage engine. See Section 16.5, “The ARCHIVE Storage Engine”.
EXAMPLE	An example engine. See Section 16.9, “The EXAMPLE Storage Engine”.
FEDERATED	Storage engine that accesses remote tables. See Section 16.8, “The FEDERATED Storage Engine”.
HEAP	This is a synonym for MEMORY.
MERGE	A collection of MyISAM tables used as one table. Also known as MRG_MyISAM. See Section 16.7, “The MERGE Storage Engine”.
NDB	Clustered, fault-tolerant, memory-based tables, supporting transactions and foreign keys. Also known as NDBCLUSTER. See Chapter 23, *MySQL NDB Cluster 8.0*.

By default, if a storage engine is specified that is not available, the statement fails with an error. You can override this behavior by removing NO_ENGINE_SUBSTITUTION from the server SQL mode (see Section 5.1.11, “Server SQL Modes”) so that MySQL allows substitution of the specified engine with the default storage engine instead. Normally in such cases, this is InnoDB, which is the default value for the default_storage_engine system variable. When NO_ENGINE_SUBSTITUTION is disabled, a warning occurs if the storage engine specification is not honored.

- AUTOEXTEND_SIZE

  Defines the amount by which InnoDB extends the size of the tablespace when it becomes full. Introduced in MySQL 8.0.23. The setting must be a multiple of 4MB. The default setting is 0, which causes the tablespace to be extended according to the implicit default behavior. For more information, see Section 15.6.3.9, “Tablespace AUTOEXTEND_SIZE Configuration”.

- AUTO_INCREMENT

  The initial AUTO_INCREMENT value for the table. In MySQL 8.0, this works for MyISAM, MEMORY, InnoDB, and ARCHIVE tables. To set the first auto-increment value for engines that do not support the AUTO_INCREMENT table option, insert a “dummy” row with a value one less than the desired value after creating the table, and then delete the dummy row.

  For engines that support the AUTO_INCREMENT table option in CREATE TABLE statements, you can also use ALTER TABLE *tbl_name* AUTO_INCREMENT = *N* to reset the AUTO_INCREMENT value. The value cannot be set lower than the maximum value currently in the column.

- AVG_ROW_LENGTH

  An approximation of the average row length for your table. You need to set this only for large tables with variable-size rows.

  When you create a MyISAM table, MySQL uses the product of the MAX_ROWS and AVG_ROW_LENGTH options to decide how big the resulting table is. If you don't specify either option, the maximum size for MyISAM data and index files is 256TB by default. (If your operating system does not support files that large, table sizes are constrained by the file size limit.) If you want to keep down the pointer sizes to make the index smaller and faster and you don't really need big files, you can decrease the default pointer size by setting the myisam_data_pointer_size system variable. (See Section 5.1.8, “Server System Variables”.) If you want all your tables to be able

to grow above the default limit and are willing to have your tables slightly slower and larger than necessary, you can increase the default pointer size by setting this variable. Setting the value to 7 permits table sizes up to 65,536TB.

- `[DEFAULT] CHARACTER SET`

  Specifies a default character set for the table. `CHARSET` is a synonym for `CHARACTER SET`. If the character set name is `DEFAULT`, the database character set is used.

- `CHECKSUM`

  Set this to 1 if you want MySQL to maintain a live checksum for all rows (that is, a checksum that MySQL updates automatically as the table changes). This makes the table a little slower to update, but also makes it easier to find corrupted tables. The `CHECKSUM TABLE` statement reports the checksum. (`MyISAM` only.)

- `[DEFAULT] COLLATE`

  Specifies a default collation for the table.

- `COMMENT`

  A comment for the table, up to 2048 characters long.

  You can set the `InnoDB` `MERGE_THRESHOLD` value for a table using the *`table_option`* `COMMENT` clause. See Section 15.8.11, “Configuring the Merge Threshold for Index Pages”.

  Setting NDB_TABLE options. The table comment in a `CREATE TABLE` that creates an `NDB` table or an `ALTER TABLE` statement which alters one can also be used to specify one to four of the `NDB_TABLE` options `NOLOGGING`, `READ_BACKUP`, `PARTITION_BALANCE`, or `FULLY_REPLICATED` as a set of name-value pairs, separated by commas if need be, immediately following the string `NDB_TABLE=` that begins the quoted comment text. An example statement using this syntax is shown here (emphasized text):

  ```
  CREATE TABLE t1 (
      c1 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      c2 VARCHAR(100),
      c3 VARCHAR(100) )
  ENGINE=NDB
  COMMENT="NDB_TABLE=READ_BACKUP=0,PARTITION_BALANCE=FOR_RP_BY_NODE";
  ```

  Spaces are not permitted within the quoted string. The string is case-insensitive.

  The comment is displayed as part of the output of `SHOW CREATE TABLE`. The text of the comment is also available as the TABLE_COMMENT column of the MySQL Information Schema `TABLES` table.

  This comment syntax is also supported with `ALTER TABLE` statements for `NDB` tables. Keep in mind that a table comment used with `ALTER TABLE` replaces any existing comment which the table might have had previously.

  Setting the `MERGE_THRESHOLD` option in table comments is not supported for `NDB` tables (it is ignored).

  For complete syntax information and examples, see Section 13.1.20.12, “Setting NDB Comment Options”.

- `COMPRESSION`

  The compression algorithm used for page level compression for `InnoDB` tables. Supported values include `Zlib`, `LZ4`, and `None`. The `COMPRESSION` attribute was introduced with the transparent page compression feature. Page compression is only supported with `InnoDB` tables that reside in file-per-table tablespaces, and is only available on Linux and Windows platforms that support sparse files and hole punching. For more information, see Section 15.9.2, “InnoDB Page Compression”.

- `CONNECTION`

  The connection string for a `FEDERATED` table.

  

  Note

  Older versions of MySQL used a `COMMENT` option for the connection string.

- `DATA DIRECTORY`, `INDEX DIRECTORY`

  For `InnoDB`, the `DATA DIRECTORY='directory'` clause permits creating tables outside of the data directory. The `innodb_file_per_table` variable must be enabled to use the `DATA DIRECTORY` clause. The full directory path must be specified. As of MySQL 8.0.21, the directory specified must be known to `InnoDB`. For more information, see Section 15.6.1.2, "Creating Tables Externally".

  When creating `MyISAM` tables, you can use the `DATA DIRECTORY='directory'` clause, the `INDEX DIRECTORY='directory'` clause, or both. They specify where to put a `MyISAM` table's data file and index file, respectively. Unlike `InnoDB` tables, MySQL does not create subdirectories that correspond to the database name when creating a `MyISAM` table with a `DATA DIRECTORY` or `INDEX DIRECTORY` option. Files are created in the directory that is specified.

  You must have the `FILE` privilege to use the `DATA DIRECTORY` or `INDEX DIRECTORY` table option.

  

  Important

  Table-level `DATA DIRECTORY` and `INDEX DIRECTORY` options are ignored for partitioned tables. (Bug #32091)

  These options work only when you are not using the `--skip-symbolic-links` option. Your operating system must also have a working, thread-safe `realpath()` call. See Section 8.12.2.2, "Using Symbolic Links for MyISAM Tables on Unix", for more complete information.

  If a `MyISAM` table is created with no `DATA DIRECTORY` option, the `.MYD` file is created in the database directory. By default, if `MyISAM` finds an existing `.MYD` file in this case, it overwrites it. The same applies to `.MYI` files for tables created with no `INDEX DIRECTORY` option. To suppress this behavior, start the server with the `--keep_files_on_create` option, in which case `MyISAM` does not overwrite existing files and returns an error instead.

  If a `MyISAM` table is created with a `DATA DIRECTORY` or `INDEX DIRECTORY` option and an existing `.MYD` or `.MYI` file is found, `MyISAM` always returns an error, and does not overwrite a file in the specified directory.

  

  Important

  You cannot use path names that contain the MySQL data directory with `DATA DIRECTORY` or `INDEX DIRECTORY`. This includes partitioned tables and individual table partitions. (See Bug #32167.)

- `DELAY_KEY_WRITE`

  Set this to 1 if you want to delay key updates for the table until the table is closed. See the description of the `delay_key_write` system variable in Section 5.1.8, "Server System Variables". (`MyISAM` only.)

- `ENCRYPTION`

  The `ENCRYPTION` clause enables or disables page-level data encryption for an `InnoDB` table. A keyring plugin must be installed and configured before encryption can be enabled. Prior to MySQL 8.0.16, the `ENCRYPTION` clause can only be specified when creating a table in an a file-per-table

tablespace. As of MySQL 8.0.16, the ENCRYPTION clause can also be specified when creating a table in a general tablespace.

As of MySQL 8.0.16, a table inherits the default schema encryption if an ENCRYPTION clause is not specified. If the table_encryption_privilege_check variable is enabled, the TABLE_ENCRYPTION_ADMIN privilege is required to create a table with an ENCRYPTION clause setting that differs from the default schema encryption. When creating a table in a general tablespace, table and tablespace encryption must match.

As of MySQL 8.0.16, specifying an ENCRYPTION clause with a value other than 'N' or '' is not permitted when using a storage engine that does not support encryption. Previously, the clause was accepted.

For more information, see Section 15.13, “InnoDB Data-at-Rest Encryption”.

- ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE options (available as of MySQL 8.0.21) are used to specify table attributes for primary and secondary storage engines. The options are reserved for future use.

  Permitted values are a string literal containing a valid JSON document or an empty string (''). Invalid JSON is rejected.

  ```
  CREATE TABLE t1 (c1 INT) ENGINE_ATTRIBUTE='{"key":"value"}';
  ```

  ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE values can be repeated without error. In this case, the last specified value is used.

  ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE values are not checked by the server, nor are they cleared when the table's storage engine is changed.

- INSERT_METHOD

  If you want to insert data into a MERGE table, you must specify with INSERT_METHOD the table into which the row should be inserted. INSERT_METHOD is an option useful for MERGE tables only. Use a value of FIRST or LAST to have inserts go to the first or last table, or a value of NO to prevent inserts. See Section 16.7, “The MERGE Storage Engine”.

- KEY_BLOCK_SIZE

  For MyISAM tables, KEY_BLOCK_SIZE optionally specifies the size in bytes to use for index key blocks. The value is treated as a hint; a different size could be used if necessary. A KEY_BLOCK_SIZE value specified for an individual index definition overrides the table-level KEY_BLOCK_SIZE value.

  For InnoDB tables, KEY_BLOCK_SIZE specifies the page size in kilobytes to use for compressed InnoDB tables. The KEY_BLOCK_SIZE value is treated as a hint; a different size could be used by InnoDB if necessary. KEY_BLOCK_SIZE can only be less than or equal to the innodb_page_size value. A value of 0 represents the default compressed page size, which is half of the innodb_page_size value. Depending on innodb_page_size, possible KEY_BLOCK_SIZE values include 0, 1, 2, 4, 8, and 16. See Section 15.9.1, “InnoDB Table Compression” for more information.

  Oracle recommends enabling innodb_strict_mode when specifying KEY_BLOCK_SIZE for InnoDB tables. When innodb_strict_mode is enabled, specifying an invalid KEY_BLOCK_SIZE

value returns an error. If `innodb_strict_mode` is disabled, an invalid `KEY_BLOCK_SIZE` value results in a warning, and the `KEY_BLOCK_SIZE` option is ignored.

The `Create_options` column in response to `SHOW TABLE STATUS` reports the actual `KEY_BLOCK_SIZE` used by the table, as does `SHOW CREATE TABLE`.

`InnoDB` only supports `KEY_BLOCK_SIZE` at the table level.

`KEY_BLOCK_SIZE` is not supported with 32KB and 64KB `innodb_page_size` values. `InnoDB` table compression does not support these pages sizes.

`InnoDB` does not support the `KEY_BLOCK_SIZE` option when creating temporary tables.

- `MAX_ROWS`

  The maximum number of rows you plan to store in the table. This is not a hard limit, but rather a hint to the storage engine that the table must be able to store at least this many rows.

  

  Important

  The use of `MAX_ROWS` with `NDB` tables to control the number of table partitions is deprecated. It remains supported in later versions for backward compatibility, but is subject to removal in a future release. Use PARTITION_BALANCE instead; see Setting NDB_TABLE options.

  The `NDB` storage engine treats this value as a maximum. If you plan to create very large NDB Cluster tables (containing millions of rows), you should use this option to insure that `NDB` allocates sufficient number of index slots in the hash table used for storing hashes of the table's primary keys by setting `MAX_ROWS = 2 * rows`, where *`rows`* is the number of rows that you expect to insert into the table.

  The maximum `MAX_ROWS` value is 4294967295; larger values are truncated to this limit.

- `MIN_ROWS`

  The minimum number of rows you plan to store in the table. The `MEMORY` storage engine uses this option as a hint about memory use.

- `PACK_KEYS`

  Takes effect only with `MyISAM` tables. Set this option to 1 if you want to have smaller indexes. This usually makes updates slower and reads faster. Setting the option to 0 disables all packing of keys. Setting it to `DEFAULT` tells the storage engine to pack only long `CHAR`, `VARCHAR`, `BINARY`, or `VARBINARY` columns.

  If you do not use `PACK_KEYS`, the default is to pack strings, but not numbers. If you use `PACK_KEYS=1`, numbers are packed as well.

  When packing binary number keys, MySQL uses prefix compression:

  - Every key needs one extra byte to indicate how many bytes of the previous key are the same for the next key.
  - The pointer to the row is stored in high-byte-first order directly after the key, to improve compression.

  This means that if you have many equal keys on two consecutive rows, all following “same” keys usually only take two bytes (including the pointer to the row). Compare this to the ordinary case where the following keys takes `storage_size_for_key + pointer_size` (where the pointer size is usually 4). Conversely, you get a significant benefit from prefix compression only if you have many numbers that are the same. If all keys are totally different, you use one byte more per key, if

the key is not a key that can have NULL values. (In this case, the packed key length is stored in the same byte that is used to mark if a key is NULL.)

- PASSWORD

  This option is unused.

- ROW_FORMAT

  Defines the physical format in which the rows are stored.

  When creating a table with strict mode disabled, the storage engine's default row format is used if the specified row format is not supported. The actual row format of the table is reported in the Row_format column in response to SHOW TABLE STATUS. The Create_options column shows the row format that was specified in the CREATE TABLE statement, as does SHOW CREATE TABLE.

  Row format choices differ depending on the storage engine used for the table.

  For InnoDB tables:

  - The default row format is defined by innodb_default_row_format, which has a default setting of DYNAMIC. The default row format is used when the ROW_FORMAT option is not defined or when ROW_FORMAT=DEFAULT is used.

    If the ROW_FORMAT option is not defined, or if ROW_FORMAT=DEFAULT is used, operations that rebuild a table also silently change the row format of the table to the default defined by innodb_default_row_format. For more information, see Defining the Row Format of a Table.

  - For more efficient InnoDB storage of data types, especially BLOB types, use the DYNAMIC. See DYNAMIC Row Format for requirements associated with the DYNAMIC row format.

  - To enable compression for InnoDB tables, specify ROW_FORMAT=COMPRESSED. The ROW_FORMAT=COMPRESSED option is not supported when creating temporary tables. See Section 15.9, “InnoDB Table and Page Compression” for requirements associated with the COMPRESSED row format.

  - The row format used in older versions of MySQL can still be requested by specifying the REDUNDANT row format.

  - When you specify a non-default ROW_FORMAT clause, consider also enabling the innodb_strict_mode configuration option.

  - ROW_FORMAT=FIXED is not supported. If ROW_FORMAT=FIXED is specified while innodb_strict_mode is disabled, InnoDB issues a warning and assumes ROW_FORMAT=DYNAMIC. If ROW_FORMAT=FIXED is specified while innodb_strict_mode is enabled, which is the default, InnoDB returns an error.

  - For additional information about InnoDB row formats, see Section 15.10, “InnoDB Row Formats”.

  For MyISAM tables, the option value can be FIXED or DYNAMIC for static or variable-length row format. myisampack sets the type to COMPRESSED. See Section 16.2.3, “MyISAM Table Storage Formats”.

  For NDB tables, the default ROW_FORMAT is DYNAMIC.

- START TRANSACTION

  This is an internal-use table option. It was introduced in MySQL 8.0.21 to permit CREATE TABLE ... SELECT to be logged as a single, atomic transaction in the binary log when using row-based replication with a storage engine that supports atomic DDL. Only BINLOG, COMMIT, and

ROLLBACK statements are permitted after CREATE TABLE ... START TRANSACTION. For related information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

- STATS_AUTO_RECALC

  Specifies whether to automatically recalculate persistent statistics for an InnoDB table. The value DEFAULT causes the persistent statistics setting for the table to be determined by the innodb_stats_auto_recalc configuration option. The value 1 causes statistics to be recalculated when 10% of the data in the table has changed. The value 0 prevents automatic recalculation for this table; with this setting, issue an ANALYZE TABLE statement to recalculate the statistics after making substantial changes to the table. For more information about the persistent statistics feature, see Section 15.8.10.1, “Configuring Persistent Optimizer Statistics Parameters”.

- STATS_PERSISTENT

  Specifies whether to enable persistent statistics for an InnoDB table. The value DEFAULT causes the persistent statistics setting for the table to be determined by the innodb_stats_persistent configuration option. The value 1 enables persistent statistics for the table, while the value 0 turns off this feature. After enabling persistent statistics through a CREATE TABLE or ALTER TABLE statement, issue an ANALYZE TABLE statement to calculate the statistics, after loading representative data into the table. For more information about the persistent statistics feature, see Section 15.8.10.1, “Configuring Persistent Optimizer Statistics Parameters”.

- STATS_SAMPLE_PAGES

  The number of index pages to sample when estimating cardinality and other statistics for an indexed column, such as those calculated by ANALYZE TABLE. For more information, see Section 15.8.10.1, “Configuring Persistent Optimizer Statistics Parameters”.

- TABLESPACE

  The TABLESPACE clause can be used to create an InnoDB table in an existing general tablespace, a file-per-table tablespace, or the system tablespace.

```
CREATE TABLE tbl_name ... TABLESPACE [=] tablespace_name
```

  The general tablespace that you specify must exist prior to using the TABLESPACE clause. For information about general tablespaces, see Section 15.6.3.3, “General Tablespaces”.

  The *tablespace_name* is a case-sensitive identifier. It may be quoted or unquoted. The forward slash character (“/”) is not permitted. Names beginning with “innodb_” are reserved for special use.

  To create a table in the system tablespace, specify innodb_system as the tablespace name.

```
CREATE TABLE tbl_name ... TABLESPACE [=] innodb_system
```

  Using TABLESPACE [=] innodb_system, you can place a table of any uncompressed row format in the system tablespace regardless of the innodb_file_per_table setting. For example, you can add a table with ROW_FORMAT=DYNAMIC to the system tablespace using TABLESPACE [=] innodb_system.

  To create a table in a file-per-table tablespace, specify innodb_file_per_table as the tablespace name.

```
CREATE TABLE tbl_name ... TABLESPACE [=] innodb_file_per_table
```

Note

If innodb_file_per_table is enabled, you need not specify TABLESPACE=innodb_file_per_table to create an InnoDB file-per-table tablespace. InnoDB tables are created in file-per-table tablespaces by default when innodb_file_per_table is enabled.

The `DATA DIRECTORY` clause is permitted with `CREATE TABLE ... TABLESPACE=innodb_file_per_table` but is otherwise not supported for use in combination with the `TABLESPACE` clause. 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.

Note

Support for `TABLESPACE = innodb_file_per_table` and `TABLESPACE = innodb_temporary` clauses with `CREATE TEMPORARY TABLE` is deprecated as of MySQL 8.0.13; expect it to be removed in a future version of MySQL.

The `STORAGE` table option is employed only with `NDB` tables. `STORAGE` determines the type of storage used, and can be either of `DISK` or `MEMORY`.

`TABLESPACE ... STORAGE DISK` assigns a table to an NDB Cluster Disk Data tablespace. `STORAGE DISK` cannot be used in `CREATE TABLE` unless preceded by `TABLESPACE` *`tablespace_name`*.

For `STORAGE MEMORY`, the tablespace name is optional, thus, you can use `TABLESPACE` *`tablespace_name`* `STORAGE MEMORY` or simply `STORAGE MEMORY` to specify explicitly that the table is in-memory.

See Section 23.6.11, “NDB Cluster Disk Data Tables”, for more information.

- `UNION`

  Used to access a collection of identical `MyISAM` tables as one. This works only with `MERGE` tables. See Section 16.7, “The MERGE Storage Engine”.

  You must have `SELECT`, `UPDATE`, and `DELETE` privileges for the tables you map to a `MERGE` table.

  

  Note

  Formerly, all tables used had to be in the same database as the `MERGE` table itself. This restriction no longer applies.

Table Partitioning

`partition_options` can be used to control partitioning of the table created with `CREATE TABLE`.

Not all options shown in the syntax for *`partition_options`* at the beginning of this section are available for all partitioning types. Please see the listings for the following individual types for information specific to each type, and see Chapter 24, *Partitioning*, for more complete information about the workings of and uses for partitioning in MySQL, as well as additional examples of table creation and other statements relating to MySQL partitioning.

Partitions can be modified, merged, added to tables, and dropped from tables. For basic information about the MySQL statements to accomplish these tasks, see Section 13.1.9, “ALTER TABLE Statement”. For more detailed descriptions and examples, see Section 24.3, “Partition Management”.

- `PARTITION BY`

  If used, a *`partition_options`* clause begins with `PARTITION BY`. This clause contains the function that is used to determine the partition; the function returns an integer value ranging from 1 to *`num`*, where *`num`* is the number of partitions. (The maximum number of user-defined partitions which a table may contain is 1024; the number of subpartitions—discussed later in this section—is included in this maximum.)

Note

The expression (*expr*) used in a PARTITION BY clause cannot refer to any columns not in the table being created; such references are specifically not permitted and cause the statement to fail with an error. (Bug #29444)

- HASH(*expr*)

  Hashes one or more columns to create a key for placing and locating rows. *expr* is an expression using one or more table columns. This can be any valid MySQL expression (including MySQL functions) that yields a single integer value. For example, these are both valid CREATE TABLE statements using PARTITION BY HASH:

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5))
    PARTITION BY HASH(col1);

CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATETIME)
    PARTITION BY HASH ( YEAR(col3) );
```

  You may not use either VALUES LESS THAN or VALUES IN clauses with PARTITION BY HASH.

  PARTITION BY HASH uses the remainder of *expr* divided by the number of partitions (that is, the modulus). For examples and additional information, see Section 24.2.4, “HASH Partitioning”.

  The LINEAR keyword entails a somewhat different algorithm. In this case, the number of the partition in which a row is stored is calculated as the result of one or more logical AND operations. For discussion and examples of linear hashing, see Section 24.2.4.1, “LINEAR HASH Partitioning”.

- KEY(*column_list*)

  This is similar to HASH, except that MySQL supplies the hashing function so as to guarantee an even data distribution. The *column_list* argument is simply a list of 1 or more table columns (maximum: 16). This example shows a simple table partitioned by key, with 4 partitions:

```
CREATE TABLE tk (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY KEY(col3)
    PARTITIONS 4;
```

  For tables that are partitioned by key, you can employ linear partitioning by using the LINEAR keyword. This has the same effect as with tables that are partitioned by HASH. That is, the partition number is found using the & operator rather than the modulus (see Section 24.2.4.1, “LINEAR HASH Partitioning”, and Section 24.2.5, “KEY Partitioning”, for details). This example uses linear partitioning by key to distribute data between 5 partitions:

```
CREATE TABLE tk (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY LINEAR KEY(col3)
    PARTITIONS 5;
```

  The ALGORITHM={1 | 2} option is supported with [SUB]PARTITION BY [LINEAR] KEY. ALGORITHM=1 causes the server to use the same key-hashing functions as MySQL 5.1; ALGORITHM=2 means that the server employs the key-hashing functions implemented and used by default for new KEY partitioned tables in MySQL 5.5 and later. (Partitioned tables created with the key-hashing functions employed in MySQL 5.5 and later cannot be used by a MySQL 5.1 server.) Not specifying the option has the same effect as using ALGORITHM=2. This option is intended for use chiefly when upgrading or downgrading [LINEAR] KEY partitioned tables between MySQL 5.1 and later MySQL versions, or for creating tables partitioned by KEY or LINEAR KEY on a

MySQL 5.5 or later server which can be used on a MySQL 5.1 server. For more information, see Section 13.1.9.1, “ALTER TABLE Partition Operations”.

`mysqldump` writes this option encased in versioned comments.

`ALGORITHM=1` is shown when necessary in the output of `SHOW CREATE TABLE` using versioned comments in the same manner as `mysqldump`. `ALGORITHM=2` is always omitted from `SHOW CREATE TABLE` output, even if this option was specified when creating the original table.

You may not use either `VALUES LESS THAN` or `VALUES IN` clauses with `PARTITION BY KEY`.

- `RANGE(expr)`

  In this case, *`expr`* shows a range of values using a set of `VALUES LESS THAN` operators. When using range partitioning, you must define at least one partition using `VALUES LESS THAN`. You cannot use `VALUES IN` with range partitioning.

  

  Note

  For tables partitioned by `RANGE`, `VALUES LESS THAN` must be used with either an integer literal value or an expression that evaluates to a single integer value. In MySQL 8.0, you can overcome this limitation in a table that is defined using `PARTITION BY RANGE COLUMNS`, as described later in this section.

  Suppose that you have a table that you wish to partition on a column containing year values, according to the following scheme.

Partition Number:	Years Range:
0	1990 and earlier
1	1991 to 1994
2	1995 to 1998
3	1999 to 2002
4	2003 to 2005
5	2006 and later

A table implementing such a partitioning scheme can be realized by the `CREATE TABLE` statement shown here:

```
CREATE TABLE t1 (
    year_col  INT,
    some_data INT
)
PARTITION BY RANGE (year_col) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1995),
    PARTITION p2 VALUES LESS THAN (1999),
    PARTITION p3 VALUES LESS THAN (2002),
    PARTITION p4 VALUES LESS THAN (2006),
    PARTITION p5 VALUES LESS THAN MAXVALUE
);
```

`PARTITION ... VALUES LESS THAN ...` statements work in a consecutive fashion. `VALUES LESS THAN MAXVALUE` works to specify “leftover” values that are greater than the maximum value otherwise specified.

`VALUES LESS THAN` clauses work sequentially in a manner similar to that of the `case` portions of a `switch ... case` block (as found in many programming languages such as C, Java, and PHP). That is, the clauses must be arranged in such a way that the upper limit specified in each successive

`VALUES LESS THAN` is greater than that of the previous one, with the one referencing `MAXVALUE` coming last of all in the list.

- `RANGE COLUMNS(column_list)`

  This variant on `RANGE` facilitates partition pruning for queries using range conditions on multiple columns (that is, having conditions such as `WHERE a = 1 AND b < 10` or `WHERE a = 1 AND b = 10 AND c < 10`). It enables you to specify value ranges in multiple columns by using a list of columns in the `COLUMNS` clause and a set of column values in each `PARTITION ... VALUES LESS THAN (value_list)` partition definition clause. (In the simplest case, this set consists of a single column.) The maximum number of columns that can be referenced in the *`column_list`* and *`value_list`* is 16.

  The *`column_list`* used in the `COLUMNS` clause may contain only names of columns; each column in the list must be one of the following MySQL data types: the integer types; the string types; and time or date column types. Columns using `BLOB`, `TEXT`, `SET`, `ENUM`, `BIT`, or spatial data types are not permitted; columns that use floating-point number types are also not permitted. You also may not use functions or arithmetic expressions in the `COLUMNS` clause.

  The `VALUES LESS THAN` clause used in a partition definition must specify a literal value for each column that appears in the `COLUMNS()` clause; that is, the list of values used for each `VALUES LESS THAN` clause must contain the same number of values as there are columns listed in the `COLUMNS` clause. An attempt to use more or fewer values in a `VALUES LESS THAN` clause than there are in the `COLUMNS` clause causes the statement to fail with the error `Inconsistency in usage of column lists for partitioning...`. You cannot use `NULL` for any value appearing in `VALUES LESS THAN`. It is possible to use `MAXVALUE` more than once for a given column other than the first, as shown in this example:

```
CREATE TABLE rc (
    a INT NOT NULL,
    b INT NOT NULL
)
PARTITION BY RANGE COLUMNS(a,b) (
    PARTITION p0 VALUES LESS THAN (10,5),
    PARTITION p1 VALUES LESS THAN (20,10),
    PARTITION p2 VALUES LESS THAN (50,MAXVALUE),
    PARTITION p3 VALUES LESS THAN (65,MAXVALUE),
    PARTITION p4 VALUES LESS THAN (MAXVALUE,MAXVALUE)
);
```

  Each value used in a `VALUES LESS THAN` value list must match the type of the corresponding column exactly; no conversion is made. For example, you cannot use the string `'1'` for a value that matches a column that uses an integer type (you must use the numeral `1` instead), nor can you use the numeral `1` for a value that matches a column that uses a string type (in such a case, you must use a quoted string: `'1'`).

  For more information, see Section 24.2.1, “RANGE Partitioning”, and Section 24.4, “Partition Pruning”.

- `LIST(expr)`

  This is useful when assigning partitions based on a table column with a restricted set of possible values, such as a state or country code. In such a case, all rows pertaining to a certain state or country can be assigned to a single partition, or a partition can be reserved for a certain set of states or countries. It is similar to `RANGE`, except that only `VALUES IN` may be used to specify permissible values for each partition.

  `VALUES IN` is used with a list of values to be matched. For instance, you could create a partitioning scheme such as the following:

```
CREATE TABLE client_firms (
    id   INT,
    name VARCHAR(35)
```

```
)
PARTITION BY LIST (id) (
    PARTITION r0 VALUES IN (1, 5, 9, 13, 17, 21),
    PARTITION r1 VALUES IN (2, 6, 10, 14, 18, 22),
    PARTITION r2 VALUES IN (3, 7, 11, 15, 19, 23),
    PARTITION r3 VALUES IN (4, 8, 12, 16, 20, 24)
);
```

When using list partitioning, you must define at least one partition using `VALUES IN`. You cannot use `VALUES LESS THAN` with `PARTITION BY LIST`.

Note

For tables partitioned by `LIST`, the value list used with `VALUES IN` must consist of integer values only. In MySQL 8.0, you can overcome this limitation using partitioning by `LIST COLUMNS`, which is described later in this section.

- `LIST COLUMNS(column_list)`

  This variant on `LIST` facilitates partition pruning for queries using comparison conditions on multiple columns (that is, having conditions such as `WHERE a = 5 AND b = 5` or `WHERE a = 1 AND b = 10 AND c = 5`). It enables you to specify values in multiple columns by using a list of columns in the `COLUMNS` clause and a set of column values in each `PARTITION ... VALUES IN (value_list)` partition definition clause.

  The rules governing regarding data types for the column list used in `LIST COLUMNS(column_list)` and the value list used in `VALUES IN(value_list)` are the same as those for the column list used in `RANGE COLUMNS(column_list)` and the value list used in `VALUES LESS THAN(value_list)`, respectively, except that in the `VALUES IN` clause, `MAXVALUE` is not permitted, and you may use `NULL`.

  There is one important difference between the list of values used for `VALUES IN` with `PARTITION BY LIST COLUMNS` as opposed to when it is used with `PARTITION BY LIST`. When used with `PARTITION BY LIST COLUMNS`, each element in the `VALUES IN` clause must be a *set* of column values; the number of values in each set must be the same as the number of columns used in the `COLUMNS` clause, and the data types of these values must match those of the columns (and occur in the same order). In the simplest case, the set consists of a single column. The maximum number of columns that can be used in the *`column_list`* and in the elements making up the *`value_list`* is 16.

  The table defined by the following `CREATE TABLE` statement provides an example of a table using `LIST COLUMNS` partitioning:

```
CREATE TABLE lc (
    a INT NULL,
    b INT NULL
)
PARTITION BY LIST COLUMNS(a,b) (
    PARTITION p0 VALUES IN( (0,0), (NULL,NULL) ),
    PARTITION p1 VALUES IN( (0,1), (0,2), (0,3), (1,1), (1,2) ),
    PARTITION p2 VALUES IN( (1,0), (2,0), (2,1), (3,0), (3,1) ),
    PARTITION p3 VALUES IN( (1,3), (2,2), (2,3), (3,2), (3,3) )
);
```

- `PARTITIONS num`

  The number of partitions may optionally be specified with a `PARTITIONS num` clause, where *`num`* is the number of partitions. If both this clause *and* any `PARTITION` clauses are used, *`num`* must be equal to the total number of any partitions that are declared using `PARTITION` clauses.

  

  Note

  Whether or not you use a `PARTITIONS` clause in creating a table that is partitioned by `RANGE` or `LIST`, you must still include at least one `PARTITION VALUES` clause in the table definition (see below).

- `SUBPARTITION BY`

  A partition may optionally be divided into a number of subpartitions. This can be indicated by using the optional `SUBPARTITION BY` clause. Subpartitioning may be done by `HASH` or `KEY`. Either of these may be `LINEAR`. These work in the same way as previously described for the equivalent partitioning types. (It is not possible to subpartition by `LIST` or `RANGE`.)

  The number of subpartitions can be indicated using the `SUBPARTITIONS` keyword followed by an integer value.

- Rigorous checking of the value used in `PARTITIONS` or `SUBPARTITIONS` clauses is applied and this value must adhere to the following rules:

  - The value must be a positive, nonzero integer.
  - No leading zeros are permitted.
  - The value must be an integer literal, and cannot not be an expression. For example, `PARTITIONS 0.2E+01` is not permitted, even though `0.2E+01` evaluates to `2`. (Bug #15890)

- *`partition_definition`*

  Each partition may be individually defined using a *`partition_definition`* clause. The individual parts making up this clause are as follows:

  - `PARTITION partition_name`

    Specifies a logical name for the partition.

  - `VALUES`

    For range partitioning, each partition must include a `VALUES LESS THAN` clause; for list partitioning, you must specify a `VALUES IN` clause for each partition. This is used to determine which rows are to be stored in this partition. See the discussions of partitioning types in Chapter 24, *Partitioning*, for syntax examples.

  - `[STORAGE] ENGINE`

    MySQL accepts a `[STORAGE] ENGINE` option for both `PARTITION` and `SUBPARTITION`. Currently, the only way in which this option can be used is to set all partitions or all subpartitions to the same storage engine, and an attempt to set different storage engines for partitions

or subpartitions in the same table raises the error `ERROR 1469 (HY000): The mix of handlers in the partitions is not permitted in this version of MySQL`.

- `COMMENT`

  An optional `COMMENT` clause may be used to specify a string that describes the partition. Example:

  ```
  COMMENT = 'Data for the years previous to 1999'
  ```

  The maximum length for a partition comment is 1024 characters.

- `DATA DIRECTORY` and `INDEX DIRECTORY`

  `DATA DIRECTORY` and `INDEX DIRECTORY` may be used to indicate the directory where, respectively, the data and indexes for this partition are to be stored. Both the *`data_dir`* and the *`index_dir`* must be absolute system path names.

  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.

  You must have the `FILE` privilege to use the `DATA DIRECTORY` or `INDEX DIRECTORY` partition option.

  Example:

  ```
  CREATE TABLE th (id INT, name VARCHAR(30), adate DATE)
  PARTITION BY LIST(YEAR(adate))
  (
    PARTITION p1999 VALUES IN (1995, 1999, 2003)
      DATA DIRECTORY = '/var/appdata/95/data'
      INDEX DIRECTORY = '/var/appdata/95/idx',
    PARTITION p2000 VALUES IN (1996, 2000, 2004)
      DATA DIRECTORY = '/var/appdata/96/data'
      INDEX DIRECTORY = '/var/appdata/96/idx',
    PARTITION p2001 VALUES IN (1997, 2001, 2005)
      DATA DIRECTORY = '/var/appdata/97/data'
      INDEX DIRECTORY = '/var/appdata/97/idx',
    PARTITION p2002 VALUES IN (1998, 2002, 2006)
      DATA DIRECTORY = '/var/appdata/98/data'
      INDEX DIRECTORY = '/var/appdata/98/idx'
  );
  ```

  `DATA DIRECTORY` and `INDEX DIRECTORY` behave in the same way as in the `CREATE TABLE` statement's *`table_option`* clause as used for `MyISAM` tables.

  One data directory and one index directory may be specified per partition. If left unspecified, the data and indexes are stored by default in the table's database directory.

  The `DATA DIRECTORY` and `INDEX DIRECTORY` options are ignored for creating partitioned tables if `NO_DIR_IN_CREATE` is in effect.

- `MAX_ROWS` and `MIN_ROWS`

  May be used to specify, respectively, the maximum and minimum number of rows to be stored in the partition. The values for *`max_number_of_rows`* and *`min_number_of_rows`* must be positive

integers. As with the table-level options with the same names, these act only as “suggestions” to the server and are not hard limits.

- `TABLESPACE`

  May be used to designate an `InnoDB` file-per-table tablespace for the partition by specifying `TABLESPACE \`innodb_file_per_table\``. All partitions must belong to the same storage engine.

  Placing `InnoDB` table partitions in shared `InnoDB` tablespaces is not supported. Shared tablespaces include the `InnoDB` system tablespace and general tablespaces.

- *`subpartition_definition`*

  The partition definition may optionally contain one or more *`subpartition_definition`* clauses. Each of these consists at a minimum of the `SUBPARTITION` *`name`*, where *`name`* is an identifier for the subpartition. Except for the replacement of the `PARTITION` keyword with `SUBPARTITION`, the syntax for a subpartition definition is identical to that for a partition definition.

  Subpartitioning must be done by `HASH` or `KEY`, and can be done only on `RANGE` or `LIST` partitions. See Section 24.2.6, “Subpartitioning”.

Partitioning by Generated Columns

Partitioning by generated columns is permitted. For example:

```
CREATE TABLE t1 (
  s1 INT,
  s2 INT AS (EXP(s1)) STORED
)
PARTITION BY LIST (s2) (
  PARTITION p1 VALUES IN (1)
);
```

Partitioning sees a generated column as a regular column, which enables workarounds for limitations on functions that are not permitted for partitioning (see Section 24.6.3, “Partitioning Limitations Relating to Functions”). The preceding example demonstrates this technique: `EXP()` cannot be used directly in the `PARTITION BY` clause, but a generated column defined using `EXP()` is permitted.

13.1.20.1 Files Created by CREATE TABLE

For an `InnoDB` table created in a file-per-table tablespace or general tablespace, table data and associated indexes are stored in a .ibd file in the database directory. When an `InnoDB` table is created in the system tablespace, table data and indexes are stored in the ibdata* files that represent the system tablespace. The `innodb_file_per_table` option controls whether tables are created in file-per-table tablespaces or the system tablespace, by default. The `TABLESPACE` option can be used to place a table in a file-per-table tablespace, general tablespace, or the system tablespace, regardless of the `innodb_file_per_table` setting.

For `MyISAM` tables, the storage engine creates data and index files. Thus, for each `MyISAM` table *`tbl_name`*, there are two disk files.

File	Purpose
`tbl_name``.MYD`	Data file
`tbl_name``.MYI`	Index file

Chapter 16, *Alternative Storage Engines*, describes what files each storage engine creates to represent tables. If a table name contains special characters, the names for the table files contain encoded versions of those characters as described in Section 9.2.4, “Mapping of Identifiers to File Names”.

13.1.20.2 CREATE TEMPORARY TABLE Statement

You can use the TEMPORARY keyword when creating a table. A TEMPORARY table is visible only within the current session, and is dropped automatically when the session is closed. This means that two different sessions can use the same temporary table name without conflicting with each other or with an existing non-TEMPORARY table of the same name. (The existing table is hidden until the temporary table is dropped.)

InnoDB does not support compressed temporary tables. When innodb_strict_mode is enabled (the default), CREATE TEMPORARY TABLE returns an error if ROW_FORMAT=COMPRESSED or KEY_BLOCK_SIZE is specified. If innodb_strict_mode is disabled, warnings are issued and the temporary table is created using a non-compressed row format. The innodb_file_per-table option does not affect the creation of InnoDB temporary tables.

CREATE TABLE causes an implicit commit, except when used with the TEMPORARY keyword. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

TEMPORARY tables have a very loose relationship with databases (schemas). Dropping a database does not automatically drop any TEMPORARY tables created within that database.

To create a temporary table, you must have the CREATE TEMPORARY TABLES privilege. After a session has created a temporary table, the server performs no further privilege checks on the table. The creating session can perform any operation on the table, such as DROP TABLE, INSERT, UPDATE, or SELECT.

One implication of this behavior is that a session can manipulate its temporary tables even if the current user has no privilege to create them. Suppose that the current user does not have the CREATE TEMPORARY TABLES privilege but is able to execute a definer-context stored procedure that executes with the privileges of a user who does have CREATE TEMPORARY TABLES and that creates a temporary table. While the procedure executes, the session uses the privileges of the defining user. After the procedure returns, the effective privileges revert to those of the current user, which can still see the temporary table and perform any operation on it.

You cannot use CREATE TEMPORARY TABLE ... LIKE to create an empty table based on the definition of a table that resides in the mysql tablespace, InnoDB system tablespace (innodb_system), or a general tablespace. The tablespace definition for such a table includes a TABLESPACE attribute that defines the tablespace where the table resides, and the aforementioned tablespaces do not support temporary tables. To create a temporary table based on the definition of such a table, use this syntax instead:

```
CREATE TEMPORARY TABLE new_tbl SELECT * FROM orig_tbl LIMIT 0;
```

Note

Support for TABLESPACE = innodb_file_per_table and TABLESPACE = innodb_temporary clauses with CREATE TEMPORARY TABLE is deprecated as of MySQL 8.0.13; expect it be removed in a future version of MySQL.

13.1.20.3 CREATE TABLE ... LIKE Statement

Use CREATE TABLE ... LIKE to create an empty table based on the definition of another table, including any column attributes and indexes defined in the original table:

```
CREATE TABLE new_tbl LIKE orig_tbl;
```

The copy is created using the same version of the table storage format as the original table. The SELECT privilege is required on the original table.

LIKE works only for base tables, not for views.

Important

You cannot execute CREATE TABLE or CREATE TABLE ... LIKE while a LOCK TABLES statement is in effect.

> `CREATE TABLE ... LIKE` makes the same checks as `CREATE TABLE`. This means that if the current SQL mode is different from the mode in effect when the original table was created, the table definition might be considered invalid for the new mode and cause the statement to fail.

For `CREATE TABLE ... LIKE`, the destination table preserves generated column information from the original table.

For `CREATE TABLE ... LIKE`, the destination table preserves expression default values from the original table.

For `CREATE TABLE ... LIKE`, the destination table preserves `CHECK` constraints from the original table, except that all the constraint names are generated.

`CREATE TABLE ... LIKE` does not preserve any `DATA DIRECTORY` or `INDEX DIRECTORY` table options that were specified for the original table, or any foreign key definitions.

If the original table is a `TEMPORARY` table, `CREATE TABLE ... LIKE` does not preserve `TEMPORARY`. To create a `TEMPORARY` destination table, use `CREATE TEMPORARY TABLE ... LIKE`.

Tables created in the `mysql` tablespace, the `InnoDB` system tablespace (`innodb_system`), or general tablespaces include a `TABLESPACE` attribute in the table definition, which defines the tablespace where the table resides. Due to a temporary regression, `CREATE TABLE ... LIKE` preserves the `TABLESPACE` attribute and creates the table in the defined tablespace regardless of the `innodb_file_per_table` setting. To avoid the `TABLESPACE` attribute when creating an empty table based on the definition of such a table, use this syntax instead:

```
CREATE TABLE new_tbl SELECT * FROM orig_tbl LIMIT 0;
```

`CREATE TABLE ... LIKE` operations apply all `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` values to the new table.

13.1.20.4 CREATE TABLE ... SELECT Statement

You can create one table from another by adding a `SELECT` statement at the end of the `CREATE TABLE` statement:

```
CREATE TABLE new_tbl [AS] SELECT * FROM orig_tbl;
```

MySQL creates new columns for all elements in the `SELECT`. For example:

```
mysql> CREATE TABLE test (a INT NOT NULL AUTO_INCREMENT,
    ->        PRIMARY KEY (a), KEY(b))
    ->        ENGINE=InnoDB SELECT b,c FROM test2;
```

This creates an `InnoDB` table with three columns, `a`, `b`, and `c`. The `ENGINE` option is part of the `CREATE TABLE` statement, and should not be used following the `SELECT`; this would result in a syntax error. The same is true for other `CREATE TABLE` options such as `CHARSET`.

Notice that the columns from the `SELECT` statement are appended to the right side of the table, not overlapped onto it. Take the following example:

```
mysql> SELECT * FROM foo;
+---+
| n |
+---+
| 1 |
+---+

mysql> CREATE TABLE bar (m INT) SELECT n FROM foo;
Query OK, 1 row affected (0.02 sec)
```

```
Records: 1  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM bar;
+------+---+
| m    | n |
+------+---+
| NULL | 1 |
+------+---+
1 row in set (0.00 sec)
```

For each row in table `foo`, a row is inserted in `bar` with the values from `foo` and default values for the new columns.

In a table resulting from `CREATE TABLE ... SELECT`, columns named only in the `CREATE TABLE` part come first. Columns named in both parts or only in the `SELECT` part come after that. The data type of `SELECT` columns can be overridden by also specifying the column in the `CREATE TABLE` part.

If errors occur while copying data to the table, the table is automatically dropped and not created. However, prior to MySQL 8.0.21, when row-based replication is in use, a `CREATE TABLE ... SELECT` statement is recorded in the binary log as two transactions, one to create the table, and the other to insert data. When the statement applied from the binary log, a failure between the two transactions or while copying data can result in replication of an empty table. That limitation is removed in MySQL 8.0.21. On storage engines that support atomic DDL, `CREATE TABLE ... SELECT` is now recorded and applied as one transaction when row-based replication is in use. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

As of MySQL 8.0.21, on storage engines that support both atomic DDL and foreign key constraints, creation of foreign keys is not permitted in `CREATE TABLE ... SELECT` statements when row-based replication is in use. Foreign key constraints can be added later using `ALTER TABLE`.

You can precede the `SELECT` by `IGNORE` or `REPLACE` to indicate how to handle rows that duplicate unique key values. With `IGNORE`, rows that duplicate an existing row on a unique key value are discarded. With `REPLACE`, new rows replace rows that have the same unique key value. If neither `IGNORE` nor `REPLACE` is specified, duplicate unique key values result in an error. For more information, see The Effect of IGNORE on Statement Execution.

In MySQL 8.0.19 and later, you can also use a `VALUES` statement in the `SELECT` part of `CREATE TABLE ... SELECT`; the `VALUES` portion of the statement must include a table alias using an `AS` clause. To name the columns coming from `VALUES`, supply column aliases with the table alias; otherwise, the default column names `column_0`, `column_1`, `column_2`, ..., are used.

Otherwise, naming of columns in the table thus created follows the same rules as described previously in this section. Examples:

```
mysql> CREATE TABLE tv1
    >     SELECT * FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v;
mysql> TABLE tv1;
+----------+----------+----------+
| column_0 | column_1 | column_2 |
+----------+----------+----------+
|        1 |        3 |        5 |
|        2 |        4 |        6 |
+----------+----------+----------+

mysql> CREATE TABLE tv2
    >     SELECT * FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v(x,y,z);
mysql> TABLE tv2;
+---+---+---+
| x | y | z |
+---+---+---+
| 1 | 3 | 5 |
| 2 | 4 | 6 |
+---+---+---+

mysql> CREATE TABLE tv3 (a INT, b INT, c INT)
```

```
    >     SELECT * FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v(x,y,z);
mysql> TABLE tv3;
+------+------+------+----------+----------+----------+
| a    | b    | c    |        x |        y |        z |
+------+------+------+----------+----------+----------+
| NULL | NULL | NULL |        1 |        3 |        5 |
| NULL | NULL | NULL |        2 |        4 |        6 |
+------+------+------+----------+----------+----------+

mysql> CREATE TABLE tv4 (a INT, b INT, c INT)
    >     SELECT * FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v(x,y,z);
mysql> TABLE tv4;
+------+------+------+---+---+---+
| a    | b    | c    | x | y | z |
+------+------+------+---+---+---+
| NULL | NULL | NULL | 1 | 3 | 5 |
| NULL | NULL | NULL | 2 | 4 | 6 |
+------+------+------+---+---+---+

mysql> CREATE TABLE tv5 (a INT, b INT, c INT)
    >     SELECT * FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v(a,b,c);
mysql> TABLE tv5;
+------+------+------+
| a    | b    | c    |
+------+------+------+
|    1 |    3 |    5 |
|    2 |    4 |    6 |
+------+------+------+
```

When selecting all columns and using the default column names, you can omit `SELECT *`, so the statement just used to create table `tv1` can also be written as shown here:

```
mysql> CREATE TABLE tv1 VALUES ROW(1,3,5), ROW(2,4,6);
mysql> TABLE tv1;
+----------+----------+----------+
| column_0 | column_1 | column_2 |
+----------+----------+----------+
|        1 |        3 |        5 |
|        2 |        4 |        6 |
+----------+----------+----------+
```

When using `VALUES` as the source of the `SELECT`, all columns are always selected into the new table, and individual columns cannot be selected as they can be when selecting from a named table; each of the following statements produces an error (`ER_OPERAND_COLUMNS`):

```
CREATE TABLE tvx
    SELECT (x,z) FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v(x,y,z);

CREATE TABLE tvx (a INT, c INT)
    SELECT (x,z) FROM (VALUES ROW(1,3,5), ROW(2,4,6)) AS v(x,y,z);
```

Similarly, you can use a `TABLE` statement in place of the `SELECT`. This follows the same rules as with `VALUES`; all columns of the source table and their names in the source table are always inserted into the new table. Examples:

```
mysql> TABLE t1;
+----+----+
| a  | b  |
+----+----+
|  1 |  2 |
|  6 |  7 |
| 10 | -4 |
| 14 |  6 |
+----+----+

mysql> CREATE TABLE tt1 TABLE t1;
mysql> TABLE tt1;
+----+----+
| a  | b  |
+----+----+
```

```
|  1 |  2 |
|  6 |  7 |
| 10 | -4 |
| 14 |  6 |
+----+----+

mysql> CREATE TABLE tt2 (x INT) TABLE t1;
mysql> TABLE tt2;
+------+----+----+
| x    | a  | b  |
+------+----+----+
| NULL |  1 |  2 |
| NULL |  6 |  7 |
| NULL | 10 | -4 |
| NULL | 14 |  6 |
+------+----+----+
```

Because the ordering of the rows in the underlying `SELECT` statements cannot always be determined, `CREATE TABLE ... IGNORE SELECT` and `CREATE TABLE ... REPLACE SELECT` statements are flagged as unsafe for statement-based replication. Such statements produce a warning in the error log when using statement-based mode and are written to the binary log using the row-based format when using `MIXED` mode. See also Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

`CREATE TABLE ... SELECT` does not automatically create any indexes for you. This is done intentionally to make the statement as flexible as possible. If you want to have indexes in the created table, you should specify these before the `SELECT` statement:

```
mysql> CREATE TABLE bar (UNIQUE (n)) SELECT n FROM foo;
```

For `CREATE TABLE ... SELECT`, the destination table does not preserve information about whether columns in the selected-from table are generated columns. The `SELECT` part of the statement cannot assign values to generated columns in the destination table.

For `CREATE TABLE ... SELECT`, the destination table does preserve expression default values from the original table.

Some conversion of data types might occur. For example, the `AUTO_INCREMENT` attribute is not preserved, and `VARCHAR` columns can become `CHAR` columns. Retrained attributes are `NULL` (or `NOT NULL`) and, for those columns that have them, `CHARACTER SET`, `COLLATION`, `COMMENT`, and the `DEFAULT` clause.

When creating a table with `CREATE TABLE ... SELECT`, make sure to alias any function calls or expressions in the query. If you do not, the `CREATE` statement might fail or result in undesirable column names.

```
CREATE TABLE artists_and_works
  SELECT artist.name, COUNT(work.artist_id) AS number_of_works
  FROM artist LEFT JOIN work ON artist.id = work.artist_id
  GROUP BY artist.id;
```

You can also explicitly specify the data type for a column in the created table:

```
CREATE TABLE foo (a TINYINT NOT NULL) SELECT b+1 AS a FROM bar;
```

For `CREATE TABLE ... SELECT`, if `IF NOT EXISTS` is given and the target table exists, nothing is inserted into the destination table, and the statement is not logged.

To ensure that the binary log can be used to re-create the original tables, MySQL does not permit concurrent inserts during `CREATE TABLE ... SELECT`. However, prior to MySQL 8.0.21, when a `CREATE TABLE ... SELECT` operation is applied from the binary log when row-based replication is in use, concurrent inserts are permitted on the replicated table while copying data. That limitation is removed in MySQL 8.0.21 on storage engines that support atomic DDL. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

You cannot use `FOR UPDATE` as part of the `SELECT` in a statement such as `CREATE TABLE new_table SELECT ... FROM old_table ...`. If you attempt to do so, the statement fails.

`CREATE TABLE ... SELECT` operations apply `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` values to columns only. Table and index `ENGINE_ATTRIBUTE` and `SECONDARY_ENGINE_ATTRIBUTE` values are not applied to the new table unless specified explicitly.

13.1.20.5 FOREIGN KEY Constraints

MySQL supports foreign keys, which permit cross-referencing related data across tables, and foreign key constraints, which help keep the related data consistent.

A foreign key relationship involves a parent table that holds the initial column values, and a child table with column values that reference the parent column values. A foreign key constraint is defined on the child table.

The essential syntax for a defining a foreign key constraint in a `CREATE TABLE` or `ALTER TABLE` statement includes the following:

```
[CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (col_name, ...)
    REFERENCES tbl_name (col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]

reference_option:
    RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT
```

Foreign key constraint usage is described under the following topics in this section:

- Identifiers
- Conditions and Restrictions
- Referential Actions
- Foreign Key Constraint Examples
- Adding Foreign Key Constraints
- Dropping Foreign Key Constraints
- Foreign Key Checks
- Locking
- Foreign Key Definitions and Metadata
- Foreign Key Errors

Identifiers

Foreign key constraint naming is governed by the following rules:

- The `CONSTRAINT symbol` value is used, if defined.
- If the `CONSTRAINT symbol` clause is not defined, or a symbol is not included following the `CONSTRAINT` keyword, a constraint name name is generated automatically.

  Prior to MySQL 8.0.16, if the `CONSTRAINT symbol` clause was not defined, or a symbol was not included following the `CONSTRAINT` keyword, both `InnoDB` and `NDB` storage engines would use the `FOREIGN_KEY index_name` if defined. In MySQL 8.0.16 and higher, the `FOREIGN_KEY index_name` is ignored.

- The `CONSTRAINT symbol` value, if defined, must be unique in the database. A duplicate `symbol` results in an error similar to: `ERROR 1005 (HY000): Can't create table 'test.fk1' (errno: 121)`.

- NDB Cluster stores foreign names using the same lettercase with which they are created. Prior to version 8.0.20, when processing `SELECT` and other SQL statements, `NDB` compared the names of foreign keys in such statements with the names as stored in a case-sensitive fashion when `lower_case_table_names` was equal to 0. In NDB 8.0.20 and later, this value no longer has any effect on how such comparisons are made, and they are always done without regard to lettercase. (Bug #30512043)

Table and column identifiers in a `FOREIGN KEY ... REFERENCES` clause can be quoted within backticks (`). Alternatively, double quotation marks (") can be used if the `ANSI_QUOTES` SQL mode is enabled. The `lower_case_table_names` system variable setting is also taken into account.

Conditions and Restrictions

Foreign key constraints are subject to the following conditions and restrictions:

- Parent and child tables must use the same storage engine, and they cannot be defined as temporary tables.

- Creating a foreign key constraint requires the `REFERENCES` privilege on the parent table.

- Corresponding columns in the foreign key and the referenced key must have similar data types. *The size and sign of fixed precision types such as `INTEGER` and `DECIMAL` must be the same*. The length of string types need not be the same. For nonbinary (character) string columns, the character set and collation must be the same.

- MySQL supports foreign key references between one column and another within a table. (A column cannot have a foreign key reference to itself.) In these cases, a “child table record” refers to a dependent record within the same table.

- MySQL requires indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. In the referencing table, there must be an index where the foreign key columns are listed as the *first* columns in the same order. Such an index is created on the referencing table automatically if it does not exist. This index might be silently dropped later if you create another index that can be used to enforce the foreign key constraint. `index_name`, if given, is used as described previously.

- `InnoDB` permits a foreign key to reference any index column or group of columns. However, in the referenced table, there must be an index where the referenced columns are the *first* columns in the same order. Hidden columns that `InnoDB` adds to an index are also considered (see Section 15.6.2.1, “Clustered and Secondary Indexes”).

  `NDB` requires an explicit unique key (or primary key) on any column referenced as a foreign key. `InnoDB` does not, which is an extension of standard SQL.

- Index prefixes on foreign key columns are not supported. Consequently, `BLOB` and `TEXT` columns cannot be included in a foreign key because indexes on those columns must always include a prefix length.

- `InnoDB` does not currently support foreign keys for tables with user-defined partitioning. This includes both parent and child tables.

  This restriction does not apply for `NDB` tables that are partitioned by `KEY` or `LINEAR KEY` (the only user partitioning types supported by the `NDB` storage engine); these may have foreign key references or be the targets of such references.

- A table in a foreign key relationship cannot be altered to use another storage engine. To change the storage engine, you must drop any foreign key constraints first.

- A foreign key constraint cannot reference a virtual generated column.

For information about how the MySQL implementation of foreign key constraints differs from the SQL standard, see Section 1.6.2.3, “FOREIGN KEY Constraint Differences”.

Referential Actions

When an `UPDATE` or `DELETE` operation affects a key value in the parent table that has matching rows in the child table, the result depends on the *referential action* specified by `ON UPDATE` and `ON DELETE` subclauses of the `FOREIGN KEY` clause. Referential actions include:

- `CASCADE`: Delete or update the row from the parent table and automatically delete or update the matching rows in the child table. Both `ON DELETE CASCADE` and `ON UPDATE CASCADE` are supported. Between two tables, do not define several `ON UPDATE CASCADE` clauses that act on the same column in the parent table or in the child table.

  If a `FOREIGN KEY` clause is defined on both tables in a foreign key relationship, making both tables a parent and child, an `ON UPDATE CASCADE` or `ON DELETE CASCADE` subclause defined for one `FOREIGN KEY` clause must be defined for the other in order for cascading operations to succeed. If an `ON UPDATE CASCADE` or `ON DELETE CASCADE` subclause is only defined for one `FOREIGN KEY` clause, cascading operations fail with an error.

  

  Note

  Cascaded foreign key actions do not activate triggers.

- `SET NULL`: Delete or update the row from the parent table and set the foreign key column or columns in the child table to `NULL`. Both `ON DELETE SET NULL` and `ON UPDATE SET NULL` clauses are supported.

  If you specify a `SET NULL` action, *make sure that you have not declared the columns in the child table as `NOT NULL`*.

- `RESTRICT`: Rejects the delete or update operation for the parent table. Specifying `RESTRICT` (or `NO ACTION`) is the same as omitting the `ON DELETE` or `ON UPDATE` clause.

- `NO ACTION`: A keyword from standard SQL. In MySQL, equivalent to `RESTRICT`. The MySQL Server rejects the delete or update operation for the parent table if there is a related foreign key value in the referenced table. Some database systems have deferred checks, and `NO ACTION` is a deferred check. In MySQL, foreign key constraints are checked immediately, so `NO ACTION` is the same as `RESTRICT`.

- `SET DEFAULT`: This action is recognized by the MySQL parser, but both `InnoDB` and `NDB` reject table definitions containing `ON DELETE SET DEFAULT` or `ON UPDATE SET DEFAULT` clauses.

For storage engines that support foreign keys, MySQL rejects any `INSERT` or `UPDATE` operation that attempts to create a foreign key value in a child table if there is no matching candidate key value in the parent table.

For an `ON DELETE` or `ON UPDATE` that is not specified, the default action is always `NO ACTION`.

As the default, an `ON DELETE NO ACTION` or `ON UPDATE NO ACTION` clause that is specified explicitly does not appear in `SHOW CREATE TABLE` output or in tables dumped with `mysqldump`. `RESTRICT`, which is an equivalent non-default keyword, appears in `SHOW CREATE TABLE` output and in tables dumped with `mysqldump`.

For `NDB` tables, `ON UPDATE CASCADE` is not supported where the reference is to the parent table's primary key.

As of NDB 8.0.16: For `NDB` tables, `ON DELETE CASCADE` is not supported where the child table contains one or more columns of any of the `TEXT` or `BLOB` types. (Bug #89511, Bug #27484882)

`InnoDB` performs cascading operations using a depth-first search algorithm on the records of the index that corresponds to the foreign key constraint.

A foreign key constraint on a stored generated column cannot use `CASCADE`, `SET NULL`, or `SET DEFAULT` as `ON UPDATE` referential actions, nor can it use `SET NULL` or `SET DEFAULT` as `ON DELETE` referential actions.

A foreign key constraint on the base column of a stored generated column cannot use `CASCADE`, `SET NULL`, or `SET DEFAULT` as `ON UPDATE` or `ON DELETE` referential actions.

Foreign Key Constraint Examples

This simple example relates `parent` and `child` tables through a single-column foreign key:

```
CREATE TABLE parent (
    id INT NOT NULL,
    PRIMARY KEY (id)
) ENGINE=INNODB;

CREATE TABLE child (
    id INT,
    parent_id INT,
    INDEX par_ind (parent_id),
    FOREIGN KEY (parent_id)
        REFERENCES parent(id)
        ON DELETE CASCADE
) ENGINE=INNODB;
```

This is a more complex example in which a `product_order` table has foreign keys for two other tables. One foreign key references a two-column index in the `product` table. The other references a single-column index in the `customer` table:

```
CREATE TABLE product (
    category INT NOT NULL, id INT NOT NULL,
    price DECIMAL,
    PRIMARY KEY(category, id)
)   ENGINE=INNODB;

CREATE TABLE customer (
    id INT NOT NULL,
    PRIMARY KEY (id)
)   ENGINE=INNODB;

CREATE TABLE product_order (
    no INT NOT NULL AUTO_INCREMENT,
    product_category INT NOT NULL,
    product_id INT NOT NULL,
    customer_id INT NOT NULL,

    PRIMARY KEY(no),
    INDEX (product_category, product_id),
    INDEX (customer_id),

    FOREIGN KEY (product_category, product_id)
      REFERENCES product(category, id)
      ON UPDATE CASCADE ON DELETE RESTRICT,

    FOREIGN KEY (customer_id)
      REFERENCES customer(id)
)   ENGINE=INNODB;
```

Adding Foreign Key Constraints

You can add a foreign key constraint to an existing table using the following `ALTER TABLE` syntax:

```
ALTER TABLE tbl_name
    ADD [CONSTRAINT [symbol]] FOREIGN KEY
```

```
    [index_name] (col_name, ...)
    REFERENCES tbl_name (col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

The foreign key can be self referential (referring to the same table). When you add a foreign key constraint to a table using `ALTER TABLE`, *remember to first create an index on the column(s) referenced by the foreign key.*

Dropping Foreign Key Constraints

You can drop a foreign key constraint using the following `ALTER TABLE` syntax:

```
ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol;
```

If the `FOREIGN KEY` clause defined a `CONSTRAINT` name when you created the constraint, you can refer to that name to drop the foreign key constraint. Otherwise, a constraint name was generated internally, and you must use that value. To determine the foreign key constraint name, use `SHOW CREATE TABLE`:

```
mysql> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
       Table: child
Create Table: CREATE TABLE `child` (
  `id` int DEFAULT NULL,
  `parent_id` int DEFAULT NULL,
  KEY `par_ind` (`parent_id`),
  CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`)
  REFERENCES `parent` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci

mysql> ALTER TABLE child DROP FOREIGN KEY `child_ibfk_1`;
```

Adding and dropping a foreign key in the same `ALTER TABLE` statement is supported for `ALTER TABLE ... ALGORITHM=INPLACE`. It is not supported for `ALTER TABLE ... ALGORITHM=COPY`.

Foreign Key Checks

In MySQL, InnoDB and NDB tables support checking of foreign key constraints. Foreign key checking is controlled by the `foreign_key_checks` variable, which is enabled by default. Typically, you leave this variable enabled during normal operation to enforce referential integrity. The `foreign_key_checks` variable has the same effect on `NDB` tables as it does for `InnoDB` tables.

The `foreign_key_checks` variable is dynamic and supports both global and session scopes. For information about using system variables, see Section 5.1.9, “Using System Variables”.

Disabling foreign key checking is useful when:

- Dropping a table that is referenced by a foreign key constraint. A referenced table can only be dropped after `foreign_key_checks` is disabled. When you drop a table, constraints defined on the table are also dropped.
- Reloading tables in different order than required by their foreign key relationships. For example, `mysqldump` produces correct definitions of tables in the dump file, including foreign key constraints for child tables. To make it easier to reload dump files for tables with foreign key relationships, `mysqldump` automatically includes a statement in the dump output that disables `foreign_key_checks`. This enables you to import the tables in any order in case the dump file contains tables that are not correctly ordered for foreign keys. Disabling `foreign_key_checks` also speeds up the import operation by avoiding foreign key checks.
- Executing `LOAD DATA` operations, to avoid foreign key checking.
- Performing an `ALTER TABLE` operation on a table that has a foreign key relationship.

When `foreign_key_checks` is disabled, foreign key constraints are ignored, with the following exceptions:

- Recreating a table that was previously dropped returns an error if the table definition does not conform to the foreign key constraints that reference the table. The table must have the correct column names and types. It must also have indexes on the referenced keys. If these requirements are not satisfied, MySQL returns Error 1005 that refers to errno: 150 in the error message, which means that a foreign key constraint was not correctly formed.
- Altering a table returns an error (errno: 150) if a foreign key definition is incorrectly formed for the altered table.
- Dropping an index required by a foreign key constraint. The foreign key constraint must be removed before dropping the index.
- Creating a foreign key constraint where a column references a nonmatching column type.

Disabling `foreign_key_checks` has these additional implications:

- It is permitted to drop a database that contains tables with foreign keys that are referenced by tables outside the database.
- It is permitted to drop a table with foreign keys referenced by other tables.
- Enabling `foreign_key_checks` does not trigger a scan of table data, which means that rows added to a table while `foreign_key_checks` is disabled are not checked for consistency when `foreign_key_checks` is re-enabled.

Locking

MySQL extends metadata locks, as necessary, to tables that are related by a foreign key constraint. Extending metadata locks prevents conflicting DML and DDL operations from executing concurrently on related tables. This feature also enables updates to foreign key metadata when a parent table is modified. In earlier MySQL releases, foreign key metadata, which is owned by the child table, could not be updated safely.

If a table is locked explicitly with `LOCK TABLES`, any tables related by a foreign key constraint are opened and locked implicitly. For foreign key checks, a shared read-only lock (`LOCK TABLES READ`) is taken on related tables. For cascading updates, a shared-nothing write lock (`LOCK TABLES WRITE`) is taken on related tables that are involved in the operation.

Foreign Key Definitions and Metadata

To view a foreign key definition, use `SHOW CREATE TABLE`:

```
mysql> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
       Table: child
Create Table: CREATE TABLE `child` (
  `id` int DEFAULT NULL,
  `parent_id` int DEFAULT NULL,
  KEY `par_ind` (`parent_id`),
  CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`)
  REFERENCES `parent` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

You can obtain information about foreign keys from the Information Schema `KEY_COLUMN_USAGE` table. An example of a query against this table is shown here:

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME
       FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
       WHERE REFERENCED_TABLE_SCHEMA IS NOT NULL;
+--------------+------------+-------------+-----------------+
| TABLE_SCHEMA | TABLE_NAME | COLUMN_NAME | CONSTRAINT_NAME |
```

```
+--------------+------------+-------------+-----------------+
| test         | child      | parent_id   | child_ibfk_1    |
+--------------+------------+-------------+-----------------+
```

You can obtain information specific to `InnoDB` foreign keys from the `INNODB_FOREIGN` and `INNODB_FOREIGN_COLS` tables. Example queries are show here:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_FOREIGN \G
*************************** 1. row ***************************
      ID: test/child_ibfk_1
FOR_NAME: test/child
REF_NAME: test/parent
  N_COLS: 1
    TYPE: 1

mysql> SELECT * FROM INFORMATION_SCHEMA.INNODB_FOREIGN_COLS \G
*************************** 1. row ***************************
          ID: test/child_ibfk_1
FOR_COL_NAME: parent_id
REF_COL_NAME: id
         POS: 0
```

Foreign Key Errors

In the event of a foreign key error involving `InnoDB` tables (usually Error 150 in the MySQL Server), information about the latest foreign key error can be obtained by checking `SHOW ENGINE INNODB STATUS` output.

```
mysql> SHOW ENGINE INNODB STATUS\G
...
------------------------
LATEST FOREIGN KEY ERROR
------------------------
2018-04-12 14:57:24 0x7f97a9c91700 Transaction:
TRANSACTION 7717, ACTIVE 0 sec inserting
mysql tables in use 1, locked 1
4 lock struct(s), heap size 1136, 3 row lock(s), undo log entries 3
MySQL thread id 8, OS thread handle 140289365317376, query id 14 localhost root update
INSERT INTO child VALUES (NULL, 1), (NULL, 2), (NULL, 3), (NULL, 4), (NULL, 5), (NULL, 6)
Foreign key constraint fails for table `test`.`child`:
,
  CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `parent` (`id`) ON DELETE
  CASCADE ON UPDATE CASCADE
Trying to add in child table, in index par_ind tuple:
DATA TUPLE: 2 fields;
 0: len 4; hex 80000003; asc     ;;
 1: len 4; hex 80000003; asc     ;;

But in parent table `test`.`parent`, in index PRIMARY,
the closest match we can find is record:
PHYSICAL RECORD: n_fields 3; compact format; info bits 0
 0: len 4; hex 80000004; asc     ;;
 1: len 6; hex 000000001e19; asc       ;;
 2: len 7; hex 81000001110137; asc       7;;
...
```

Warning

If a user has table-level privileges for all parent tables, `ER_NO_REFERENCED_ROW_2` and `ER_ROW_IS_REFERENCED_2` error messages for foreign key operations expose information about parent tables. If a user does not have table-level privileges for all parent tables, more generic error messages are displayed instead (`ER_NO_REFERENCED_ROW` and `ER_ROW_IS_REFERENCED`).

An exception is that, for stored programs defined to execute with `DEFINER` privileges, the user against which privileges are assessed is the user in the program `DEFINER` clause, not the invoking user. If that user has table-level

parent table privileges, parent table information is still displayed. In this case, it is the responsibility of the stored program creator to hide the information by including appropriate condition handlers.

13.1.20.6 CHECK Constraints

Prior to MySQL 8.0.16, `CREATE TABLE` permits only the following limited version of table `CHECK` constraint syntax, which is parsed and ignored:

```
CHECK (expr)
```

As of MySQL 8.0.16, `CREATE TABLE` permits the core features of table and column `CHECK` constraints, for all storage engines. `CREATE TABLE` permits the following `CHECK` constraint syntax, for both table constraints and column constraints:

```
[CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED]
```

The optional *`symbol`* specifies a name for the constraint. If omitted, MySQL generates a name from the table name, a literal `_chk_`, and an ordinal number (1, 2, 3, ...). Constraint names have a maximum length of 64 characters. They are case-sensitive, but not accent-sensitive.

`expr` specifies the constraint condition as a boolean expression that must evaluate to `TRUE` or `UNKNOWN` (for `NULL` values) for each row of the table. If the condition evaluates to `FALSE`, it fails and a constraint violation occurs. The effect of a violation depends on the statement being executed, as described later in this section.

The optional enforcement clause indicates whether the constraint is enforced:

- If omitted or specified as `ENFORCED`, the constraint is created and enforced.
- If specified as `NOT ENFORCED`, the constraint is created but not enforced.

A `CHECK` constraint is specified as either a table constraint or column constraint:

- A table constraint does not appear within a column definition and can refer to any table column or columns. Forward references are permitted to columns appearing later in the table definition.
- A column constraint appears within a column definition and can refer only to that column.

Consider this table definition:

```
CREATE TABLE t1
(
  CHECK (c1 <> c2),
  c1 INT CHECK (c1 > 10),
  c2 INT CONSTRAINT c2_positive CHECK (c2 > 0),
  c3 INT CHECK (c3 < 100),
  CONSTRAINT c1_nonzero CHECK (c1 <> 0),
  CHECK (c1 > c3)
);
```

The definition includes table constraints and column constraints, in named and unnamed formats:

- The first constraint is a table constraint: It occurs outside any column definition, so it can (and does) refer to multiple table columns. This constraint contains forward references to columns not defined yet. No constraint name is specified, so MySQL generates a name.
- The next three constraints are column constraints: Each occurs within a column definition, and thus can refer only to the column being defined. One of the constraints is named explicitly. MySQL generates a name for each of the other two.
- The last two constraints are table constraints. One of them is named explicitly. MySQL generates a name for the other one.

As mentioned, MySQL generates a name for any CHECK constraint specified without one. To see the names generated for the preceding table definition, use SHOW CREATE TABLE:

```
mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `c1` int(11) DEFAULT NULL,
  `c2` int(11) DEFAULT NULL,
  `c3` int(11) DEFAULT NULL,
  CONSTRAINT `c1_nonzero` CHECK ((`c1` <> 0)),
  CONSTRAINT `c2_positive` CHECK ((`c2` > 0)),
  CONSTRAINT `t1_chk_1` CHECK ((`c1` <> `c2`)),
  CONSTRAINT `t1_chk_2` CHECK ((`c1` > 10)),
  CONSTRAINT `t1_chk_3` CHECK ((`c3` < 100)),
  CONSTRAINT `t1_chk_4` CHECK ((`c1` > `c3`))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

The SQL standard specifies that all types of constraints (primary key, unique index, foreign key, check) belong to the same namespace. In MySQL, each constraint type has its own namespace per schema (database). Consequently, CHECK constraint names must be unique per schema; no two tables in the same schema can share a CHECK constraint name. (Exception: A TEMPORARY table hides a non-TEMPORARY table of the same name, so it can have the same CHECK constraint names as well.)

Beginning generated constraint names with the table name helps ensure schema uniqueness because table names also must be unique within the schema.

CHECK condition expressions must adhere to the following rules. An error occurs if an expression contains disallowed constructs.

- Nongenerated and generated columns are permitted, except columns with the AUTO_INCREMENT attribute and columns in other tables.
- Literals, deterministic built-in functions, and operators are permitted. A function is deterministic if, given the same data in tables, multiple invocations produce the same result, independently of the connected user. Examples of functions that are nondeterministic and fail this definition: CONNECTION_ID(), CURRENT_USER(), NOW().
- Stored functions and loadable functions are not permitted.
- Stored procedure and function parameters are not permitted.
- Variables (system variables, user-defined variables, and stored program local variables) are not permitted.
- Subqueries are not permitted.

Foreign key referential actions (ON UPDATE, ON DELETE) are prohibited on columns used in CHECK constraints. Likewise, CHECK constraints are prohibited on columns used in foreign key referential actions.

CHECK constraints are evaluated for INSERT, UPDATE, REPLACE, LOAD DATA, and LOAD XML statements and an error occurs if a constraint evaluates to FALSE. If an error occurs, handling of changes already applied differs for transactional and nontransactional storage engines, and also depends on whether strict SQL mode is in effect, as described in Strict SQL Mode.

CHECK constraints are evaluated for INSERT IGNORE, UPDATE IGNORE, LOAD DATA ... IGNORE, and LOAD XML ... IGNORE statements and a warning occurs if a constraint evaluates to FALSE. The insert or update for any offending row is skipped.

If the constraint expression evaluates to a data type that differs from the declared column type, implicit coercion to the declared type occurs according to the usual MySQL type-conversion rules. See Section 12.3, “Type Conversion in Expression Evaluation”. If type conversion fails or results in a loss of precision, an error occurs.

Note

Constraint expression evaluation uses the SQL mode in effect at evaluation time. If any component of the expression depends on the SQL mode, different results may occur for different uses of the table unless the SQL mode is the same during all uses.

The Information Schema `CHECK_CONSTRAINTS` table provides information about `CHECK` constraints defined on tables. See Section 26.3.5, "The INFORMATION_SCHEMA CHECK_CONSTRAINTS Table".

13.1.20.7 Silent Column Specification Changes

In some cases, MySQL silently changes column specifications from those given in a `CREATE TABLE` or `ALTER TABLE` statement. These might be changes to a data type, to attributes associated with a data type, or to an index specification.

All changes are subject to the internal row-size limit of 65,535 bytes, which may cause some attempts at data type changes to fail. See Section 8.4.7, "Limits on Table Column Count and Row Size".

- Columns that are part of a `PRIMARY KEY` are made `NOT NULL` even if not declared that way.
- Trailing spaces are automatically deleted from `ENUM` and `SET` member values when the table is created.
- MySQL maps certain data types used by other SQL database vendors to MySQL types. See Section 11.9, "Using Data Types from Other Database Engines".
- If you include a `USING` clause to specify an index type that is not permitted for a given storage engine, but there is another index type available that the engine can use without affecting query results, the engine uses the available type.
- If strict SQL mode is not enabled, a `VARCHAR` column with a length specification greater than 65535 is converted to `TEXT`, and a `VARBINARY` column with a length specification greater than 65535 is converted to `BLOB`. Otherwise, an error occurs in either of these cases.
- Specifying the `CHARACTER SET binary` attribute for a character data type causes the column to be created as the corresponding binary data type: `CHAR` becomes `BINARY`, `VARCHAR` becomes `VARBINARY`, and `TEXT` becomes `BLOB`. For the `ENUM` and `SET` data types, this does not occur; they are created as declared. Suppose that you specify a table using this definition:

```
CREATE TABLE t
(
  c1 VARCHAR(10) CHARACTER SET binary,
  c2 TEXT CHARACTER SET binary,
  c3 ENUM('a','b','c') CHARACTER SET binary
);
```

The resulting table has this definition:

```
CREATE TABLE t
(
  c1 VARBINARY(10),
  c2 BLOB,
  c3 ENUM('a','b','c') CHARACTER SET binary
);
```

To see whether MySQL used a data type other than the one you specified, issue a `DESCRIBE` or `SHOW CREATE TABLE` statement after creating or altering the table.

Certain other data type changes can occur if you compress a table using `myisampack`. See Section 16.2.3.3, "Compressed Table Characteristics".

13.1.20.8 CREATE TABLE and Generated Columns

`CREATE TABLE` supports the specification of generated columns. Values of a generated column are computed from an expression included in the column definition.

Generated columns are also supported by the `NDB` storage engine.

The following simple example shows a table that stores the lengths of the sides of right triangles in the `sidea` and `sideb` columns, and computes the length of the hypotenuse in `sidec` (the square root of the sums of the squares of the other sides):

```
CREATE TABLE triangle (
  sidea DOUBLE,
  sideb DOUBLE,
  sidec DOUBLE AS (SQRT(sidea * sidea + sideb * sideb))
);
INSERT INTO triangle (sidea, sideb) VALUES(1,1),(3,4),(6,8);
```

Selecting from the table yields this result:

```
mysql> SELECT * FROM triangle;
+-------+-------+--------------------+
| sidea | sideb | sidec              |
+-------+-------+--------------------+
|     1 |     1 | 1.4142135623730951 |
|     3 |     4 |                  5 |
|     6 |     8 |                 10 |
+-------+-------+--------------------+
```

Any application that uses the `triangle` table has access to the hypotenuse values without having to specify the expression that calculates them.

Generated column definitions have this syntax:

```
col_name data_type [GENERATED ALWAYS] AS (expr)
  [VIRTUAL | STORED] [NOT NULL | NULL]
  [UNIQUE [KEY]] [[PRIMARY] KEY]
  [COMMENT 'string']
```

`AS (expr)` indicates that the column is generated and defines the expression used to compute column values. `AS` may be preceded by `GENERATED ALWAYS` to make the generated nature of the column more explicit. Constructs that are permitted or prohibited in the expression are discussed later.

The `VIRTUAL` or `STORED` keyword indicates how column values are stored, which has implications for column use:

- `VIRTUAL`: Column values are not stored, but are evaluated when rows are read, immediately after any `BEFORE` triggers. A virtual column takes no storage.

  `InnoDB` supports secondary indexes on virtual columns. See Section 13.1.20.9, “Secondary Indexes and Generated Columns”.

- `STORED`: Column values are evaluated and stored when rows are inserted or updated. A stored column does require storage space and can be indexed.

The default is `VIRTUAL` if neither keyword is specified.

It is permitted to mix `VIRTUAL` and `STORED` columns within a table.

Other attributes may be given to indicate whether the column is indexed or can be `NULL`, or provide a comment.

Generated column expressions must adhere to the following rules. An error occurs if an expression contains disallowed constructs.

- Literals, deterministic built-in functions, and operators are permitted. A function is deterministic if, given the same data in tables, multiple invocations produce the same result, independently

of the connected user. Examples of functions that are nondeterministic and fail this definition: `CONNECTION_ID()`, `CURRENT_USER()`, `NOW()`.

- Stored functions and loadable functions are not permitted.
- Stored procedure and function parameters are not permitted.
- Variables (system variables, user-defined variables, and stored program local variables) are not permitted.
- Subqueries are not permitted.
- A generated column definition can refer to other generated columns, but only those occurring earlier in the table definition. A generated column definition can refer to any base (nongenerated) column in the table whether its definition occurs earlier or later.
- The `AUTO_INCREMENT` attribute cannot be used in a generated column definition.
- An `AUTO_INCREMENT` column cannot be used as a base column in a generated column definition.
- If expression evaluation causes truncation or provides incorrect input to a function, the `CREATE TABLE` statement terminates with an error and the DDL operation is rejected.

If the expression evaluates to a data type that differs from the declared column type, implicit coercion to the declared type occurs according to the usual MySQL type-conversion rules. See Section 12.3, “Type Conversion in Expression Evaluation”.

If a generated column uses the `TIMESTAMP` data type, the setting for `explicit_defaults_for_timestamp` is ignored. In such cases, if this variable is disabled then `NULL` is not converted to `CURRENT_TIMESTAMP`. In MySQL 8.0.22 and later, if the column is also declared as `NOT NULL`, attempting to insert `NULL` is explicitly rejected with `ER_BAD_NULL_ERROR`.

Note

Expression evaluation uses the SQL mode in effect at evaluation time. If any component of the expression depends on the SQL mode, different results may occur for different uses of the table unless the SQL mode is the same during all uses.

For `CREATE TABLE ... LIKE`, the destination table preserves generated column information from the original table.

For `CREATE TABLE ... SELECT`, the destination table does not preserve information about whether columns in the selected-from table are generated columns. The `SELECT` part of the statement cannot assign values to generated columns in the destination table.

Partitioning by generated columns is permitted. See Table Partitioning.

A foreign key constraint on a stored generated column cannot use `CASCADE`, `SET NULL`, or `SET DEFAULT` as `ON UPDATE` referential actions, nor can it use `SET NULL` or `SET DEFAULT` as `ON DELETE` referential actions.

A foreign key constraint on the base column of a stored generated column cannot use `CASCADE`, `SET NULL`, or `SET DEFAULT` as `ON UPDATE` or `ON DELETE` referential actions.

A foreign key constraint cannot reference a virtual generated column.

Triggers cannot use `NEW.col_name` or use `OLD.col_name` to refer to generated columns.

For `INSERT`, `REPLACE`, and `UPDATE`, if a generated column is inserted into, replaced, or updated explicitly, the only permitted value is `DEFAULT`.

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

Generated columns have several use cases, such as these:

- Virtual generated columns can be used as a way to simplify and unify queries. A complicated condition can be defined as a generated column and referred to from multiple queries on the table to ensure that all of them use exactly the same condition.
- Stored generated columns can be used as a materialized cache for complicated conditions that are costly to calculate on the fly.
- Generated columns can simulate functional indexes: Use a generated column to define a functional expression and index it. This can be useful for working with columns of types that cannot be indexed directly, such as `JSON` columns; see Indexing a Generated Column to Provide a JSON Column Index, for a detailed example.

  For stored generated columns, the disadvantage of this approach is that values are stored twice; once as the value of the generated column and once in the index.
- If a generated column is indexed, the optimizer recognizes query expressions that match the column definition and uses indexes from the column as appropriate during query execution, even if a query does not refer to the column directly by name. For details, see Section 8.3.11, “Optimizer Use of Generated Column Indexes”.

Example:

Suppose that a table `t1` contains `first_name` and `last_name` columns and that applications frequently construct the full name using an expression like this:

```
SELECT CONCAT(first_name,' ',last_name) AS full_name FROM t1;
```

One way to avoid writing out the expression is to create a view `v1` on `t1`, which simplifies applications by enabling them to select `full_name` directly without using an expression:

```
CREATE VIEW v1 AS
SELECT *, CONCAT(first_name,' ',last_name) AS full_name FROM t1;

SELECT full_name FROM v1;
```

A generated column also enables applications to select `full_name` directly without the need to define a view:

```
CREATE TABLE t1 (
  first_name VARCHAR(10),
  last_name VARCHAR(10),
  full_name VARCHAR(255) AS (CONCAT(first_name,' ',last_name))
);

SELECT full_name FROM t1;
```

13.1.20.9 Secondary Indexes and Generated Columns

`InnoDB` supports secondary indexes on virtual generated columns. Other index types are not supported. A secondary index defined on a virtual column is sometimes referred to as a “virtual index”.

A secondary index may be created on one or more virtual columns or on a combination of virtual columns and regular columns or stored generated columns. Secondary indexes that include virtual columns may be defined as `UNIQUE`.

When a secondary index is created on a virtual generated column, generated column values are materialized in the records of the index. If the index is a covering index (one that includes all the

columns retrieved by a query), generated column values are retrieved from materialized values in the index structure instead of computed “on the fly”.

There are additional write costs to consider when using a secondary index on a virtual column due to computation performed when materializing virtual column values in secondary index records during `INSERT` and `UPDATE` operations. Even with additional write costs, secondary indexes on virtual columns may be preferable to generated *stored* columns, which are materialized in the clustered index, resulting in larger tables that require more disk space and memory. If a secondary index is not defined on a virtual column, there are additional costs for reads, as virtual column values must be computed each time the column's row is examined.

Values of an indexed virtual column are MVCC-logged to avoid unnecessary recomputation of generated column values during rollback or during a purge operation. The data length of logged values is limited by the index key limit of 767 bytes for `COMPACT` and `REDUNDANT` row formats, and 3072 bytes for `DYNAMIC` and `COMPRESSED` row formats.

Adding or dropping a secondary index on a virtual column is an in-place operation.

Indexing a Generated Column to Provide a JSON Column Index

As noted elsewhere, `JSON` columns cannot be indexed directly. To create an index that references such a column indirectly, you can define a generated column that extracts the information that should be indexed, then create an index on the generated column, as shown in this example:

```
mysql> CREATE TABLE jemp (
    ->     c JSON,
    ->     g INT GENERATED ALWAYS AS (c->"$.id"),
    ->     INDEX i (g)
    -> );
Query OK, 0 rows affected (0.28 sec)

mysql> INSERT INTO jemp (c) VALUES
     >   ('{"id": "1", "name": "Fred"}'), ('{"id": "2", "name": "Wilma"}'),
     >   ('{"id": "3", "name": "Barney"}'), ('{"id": "4", "name": "Betty"}');
Query OK, 4 rows affected (0.04 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT c->>"$.name" AS name
     >     FROM jemp WHERE g > 2;
+--------+
| name   |
+--------+
| Barney |
| Betty  |
+--------+
2 rows in set (0.00 sec)

mysql> EXPLAIN SELECT c->>"$.name" AS name
     >    FROM jemp WHERE g > 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: jemp
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 5
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: Using where
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
```

```
   Code: 1003
Message: /* select#1 */ select json_unquote(json_extract(`test`.`jemp`.`c`,'$.name'))
AS `name` from `test`.`jemp` where (`test`.`jemp`.`g` > 2)
1 row in set (0.00 sec)
```

(We have wrapped the output from the last statement in this example to fit the viewing area.)

When you use `EXPLAIN` on a `SELECT` or other SQL statement containing one or more expressions that use the `->` or `->>` operator, these expressions are translated into their equivalents using `JSON_EXTRACT()` and (if needed) `JSON_UNQUOTE()` instead, as shown here in the output from `SHOW WARNINGS` immediately following this `EXPLAIN` statement:

```
mysql> EXPLAIN SELECT c->>"$.name"
     > FROM jemp WHERE g > 2 ORDER BY c->"$.name"\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: jemp
   partitions: NULL
         type: range
possible_keys: i
          key: i
      key_len: 5
          ref: NULL
         rows: 2
     filtered: 100.00
        Extra: Using where; Using filesort
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select json_unquote(json_extract(`test`.`jemp`.`c`,'$.name')) AS
`c->>"$.name"` from `test`.`jemp` where (`test`.`jemp`.`g` > 2) order by
json_extract(`test`.`jemp`.`c`,'$.name')
1 row in set (0.00 sec)
```

See the descriptions of the `->` and `->>` operators, as well as those of the `JSON_EXTRACT()` and `JSON_UNQUOTE()` functions, for additional information and examples.

This technique also can be used to provide indexes that indirectly reference columns of other types that cannot be indexed directly, such as `GEOMETRY` columns.

In MySQL 8.0.21 and later, it is also possible to create an index on a `JSON` column using the `JSON_VALUE()` function with an expression that can be used to optimize queries employing the expression. See the description of that function for more information and examples.

JSON columns and indirect indexing in NDB Cluster

It is also possible to use indirect indexing of JSON columns in MySQL NDB Cluster, subject to the following conditions:

1. `NDB` handles a `JSON` column value internally as a `BLOB`. This means that any `NDB` table having one or more JSON columns must have a primary key, else it cannot be recorded in the binary log.

2. The `NDB` storage engine does not support indexing of virtual columns. Since the default for generated columns is `VIRTUAL`, you must specify explicitly the generated column to which to apply the indirect index as `STORED`.

The **`CREATE TABLE`** statement used to create the table `jempn` shown here is a version of the `jemp` table shown previously, with modifications making it compatible with `NDB`:

```
CREATE TABLE jempn (
  a BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
```

```
  c JSON DEFAULT NULL,
  g INT GENERATED ALWAYS AS (c->"$.id") STORED,
  INDEX i (g)
) ENGINE=NDB;
```

We can populate this table using the following `INSERT` statement:

```
INSERT INTO jempn (c) VALUES
  ('{"id": "1", "name": "Fred"}'),
  ('{"id": "2", "name": "Wilma"}'),
  ('{"id": "3", "name": "Barney"}'),
  ('{"id": "4", "name": "Betty"}');
```

Now `NDB` can use index `i`, as shown here:

```
mysql> EXPLAIN SELECT c->>"$.name" AS name
    ->           FROM jempn WHERE g > 2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: jempn
   partitions: p0,p1,p2,p3
         type: range
possible_keys: i
          key: i
      key_len: 5
          ref: NULL
         rows: 3
     filtered: 100.00
        Extra: Using pushed condition (`test`.`jempn`.`g` > 2)
1 row in set, 1 warning (0.01 sec)

mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select
json_unquote(json_extract(`test`.`jempn`.`c`,'$.name')) AS `name` from
`test`.`jempn` where (`test`.`jempn`.`g` > 2)
1 row in set (0.00 sec)
```

You should keep in mind that a stored generated column, as well as any index on such a column, uses `DataMemory`.

13.1.20.10 Invisible Columns

MySQL supports invisible columns as of MySQL 8.0.23. An invisible column is normally hidden to queries, but can be accessed if explicitly referenced. Prior to MySQL 8.0.23, all columns are visible.

As an illustration of when invisible columns may be useful, suppose that an application uses `SELECT *` queries to access a table, and must continue to work without modification even if the table is altered to add a new column that the application does not expect to be there. In a `SELECT *` query, the `*` evaluates to all table columns, except those that are invisible, so the solution is to add the new column as an invisible column. The column remains “hidden” from `SELECT *` queries, and the application continues to work as previously. A newer version of the application can refer to the invisible column if necessary by explicitly referencing it.

The following sections detail how MySQL treats invisible columns.

- DDL Statements and Invisible Columns
- DML Statements and Invisible Columns
- Invisible Column Metadata
- The Binary Log and Invisible Columns

DDL Statements and Invisible Columns

Columns are visible by default. To explicitly specify visibility for a new column, use a `VISIBLE` or `INVISIBLE` keyword as part of the column definition for `CREATE TABLE` or `ALTER TABLE`:

```
CREATE TABLE t1 (
  i INT,
  j DATE INVISIBLE
) ENGINE = InnoDB;
ALTER TABLE t1 ADD COLUMN k INT INVISIBLE;
```

To alter the visibility of an existing column, use a `VISIBLE` or `INVISIBLE` keyword with one of the `ALTER TABLE` column-modification clauses:

```
ALTER TABLE t1 CHANGE COLUMN j j DATE VISIBLE;
ALTER TABLE t1 MODIFY COLUMN j DATE INVISIBLE;
ALTER TABLE t1 ALTER COLUMN j SET VISIBLE;
```

A table must have at least one visible column. Attempting to make all columns invisible produces an error.

Invisible columns support the usual column attributes: `NULL`, `NOT NULL`, `AUTO_INCREMENT`, and so forth.

Generated columns can be invisible.

Index definitions can name invisible columns, including definitions for `PRIMARY KEY` and `UNIQUE` indexes. Although a table must have at least one visible column, an index definition need not have any visible columns.

An invisible column dropped from a table is dropped in the usual way from any index definition that names the column.

Foreign key constraints can be defined on invisible columns, and foreign key constraints can reference invisible columns.

`CHECK` constraints can be defined on invisible columns. For new or modified rows, violation of a `CHECK` constraint on an invisible column produces an error.

`CREATE TABLE ... LIKE` includes invisible columns, and they are invisible in the new table.

`CREATE TABLE ... SELECT` does not include invisible columns, unless they are explicitly referenced in the `SELECT` part. However, even if explicitly referenced, a column that is invisible in the existing table is visible in the new table:

```
mysql> CREATE TABLE t1 (col1 INT, col2 INT INVISIBLE);
mysql> CREATE TABLE t2 AS SELECT col1, col2 FROM t1;
mysql> SHOW CREATE TABLE t2\G
*************************** 1. row ***************************
       Table: t2
Create Table: CREATE TABLE `t2` (
  `col1` int DEFAULT NULL,
  `col2` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

If invisibility should be preserved, provide a definition for the invisible column in the `CREATE TABLE` part of the `CREATE TABLE ... SELECT` statement:

```
mysql> CREATE TABLE t1 (col1 INT, col2 INT INVISIBLE);
mysql> CREATE TABLE t2 (col2 INT INVISIBLE) AS SELECT col1, col2 FROM t1;
mysql> SHOW CREATE TABLE t2\G
*************************** 1. row ***************************
       Table: t2
Create Table: CREATE TABLE `t2` (
  `col1` int DEFAULT NULL,
```

```
  `col2` int DEFAULT NULL /*!80023 INVISIBLE */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Views can refer to invisible columns by explicitly referencing them in the `SELECT` statement that defines the view. Changing a column's visibility subsequent to defining a view that references the column does not change view behavior.

DML Statements and Invisible Columns

For `SELECT` statements, an invisible column is not part of the result set unless explicitly referenced in the select list. In a select list, the `*` and *`tbl_name.*`* shorthands do not include invisible columns. Natural joins do not include invisible columns.

Consider the following statement sequence:

```
mysql> CREATE TABLE t1 (col1 INT, col2 INT INVISIBLE);
mysql> INSERT INTO t1 (col1, col2) VALUES(1, 2), (3, 4);

mysql> SELECT * FROM t1;
+------+
| col1 |
+------+
|    1 |
|    3 |
+------+

mysql> SELECT col1, col2 FROM t1;
+------+------+
| col1 | col2 |
+------+------+
|    1 |    2 |
|    3 |    4 |
+------+------+
```

The first `SELECT` does not reference the invisible column `col2` in the select list (because `*` does not include invisible columns), so `col2` does not appear in the statement result. The second `SELECT` explicitly references `col2`, so the column appears in the result.

The statement `TABLE t1` produces the same output as the first `SELECT` statement. Since there is no way to specify columns in a `TABLE` statement, `TABLE` never displays invisible columns.

For statements that create new rows, an invisible column is assigned its implicit default value unless explicitly referenced and assigned a value. For information about implicit defaults, see Implicit Default Handling.

For `INSERT` (and `REPLACE`, for non-replaced rows), implicit default assignment occurs with a missing column list, an empty column list, or a nonempty column list that does not include the invisible column:

```
CREATE TABLE t1 (col1 INT, col2 INT INVISIBLE);
INSERT INTO t1 VALUES(...);
INSERT INTO t1 () VALUES(...);
INSERT INTO t1 (col1) VALUES(...);
```

For the first two `INSERT` statements, the `VALUES()` list must provide a value for each visible column and no invisible column. For the third `INSERT` statement, the `VALUES()` list must provide the same number of values as the number of named columns; the same is true when you use `VALUES ROW()` rather than `VALUES()`.

For `LOAD DATA` and `LOAD XML`, implicit default assignment occurs with a missing column list or a nonempty column list that does not include the invisible column. Input rows should not include a value for the invisible column.

To assign a value other than the implicit default for the preceding statements, explicitly name the invisible column in the column list and provide a value for it.

`INSERT INTO ... SELECT *` and `REPLACE INTO ... SELECT *` do not include invisible columns because `*` does not include invisible columns. Implicit default assignment occurs as described previously.

For statements that insert or ignore new rows, or that replace or modify existing rows, based on values in a `PRIMARY KEY` or `UNIQUE` index, MySQL treats invisible columns the same as visible columns: Invisible columns participate in key value comparisons. Specifically, if a new row has the same value as an existing row for a unique key value, these behaviors occur whether the index columns are visible or invisible:

- With the `IGNORE` modifier, `INSERT`, `LOAD DATA`, and `LOAD XML` ignore the new row.
- `REPLACE` replaces the existing row with the new row. With the `REPLACE` modifier, `LOAD DATA` and `LOAD XML` do the same.
- `INSERT ... ON DUPLICATE KEY UPDATE` updates the existing row.

To update invisible columns for `UPDATE` statements, name them and assign a value, just as for visible columns.

Invisible Column Metadata

Information about whether a column is visible or invisible is available from the `EXTRA` column of the Information Schema `COLUMNS` table or `SHOW COLUMNS` output. For example:

```
mysql> SELECT TABLE_NAME, COLUMN_NAME, EXTRA
       FROM INFORMATION_SCHEMA.COLUMNS
       WHERE TABLE_SCHEMA = 'test' AND TABLE_NAME = 't1';
+------------+-------------+-----------+
| TABLE_NAME | COLUMN_NAME | EXTRA     |
+------------+-------------+-----------+
| t1         | i           |           |
| t1         | j           |           |
| t1         | k           | INVISIBLE |
+------------+-------------+-----------+
```

Columns are visible by default, so in that case, `EXTRA` displays no visibility information. For invisible columns, `EXTRA` displays `INVISIBLE`.

`SHOW CREATE TABLE` displays invisible columns in the table definition, with the `INVISIBLE` keyword in a version-specific comment:

```
mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `i` int DEFAULT NULL,
  `j` int DEFAULT NULL,
  `k` int DEFAULT NULL /*!80023 INVISIBLE */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

`mysqldump` and `mysqlpump` use `SHOW CREATE TABLE`, so they include invisible columns in dumped table definitions. They also include invisible column values in dumped data.

Reloading a dump file into an older version of MySQL that does not support invisible columns causes the version-specific comment to be ignored, which creates any invisible columns as visible.

The Binary Log and Invisible Columns

MySQL treats invisible columns as follows with respect to events in the binary log:

- Table-creation events include the `INVISIBLE` attribute for invisible columns.
- Invisible columns are treated like visible columns in row events. They are included if needed according to the `binlog_row_image` system variable setting.

- When row events are applied, invisible columns are treated like visible columns in row events. In particular, the algorithm and index to use are chosen according to the `slave_rows_search_algorithms` system variable setting.
- Invisible columns are treated like visible columns when computing writesets. In particular, writesets include indexes defined on invisible columns.
- The `mysqlbinlog` command includes visibility in column metadata.

13.1.20.11 Generated Invisible Primary Keys

Beginning with MySQL 8.0.30, MySQL supports generated invisible primary keys for any `InnoDB` table that is created without an explicit primary key. When the `sql_generate_invisible_primary_key` server system variable is set to `ON`, the MySQL server automatically adds a generated invisible primary key (GIPK) to any such table.

By default, the value of `sql_generate_invisible_primary_key` is `OFF`, meaning that the automatic addition of GIPKs is disabled. To illustrate how this affects table creation, we begin by creating two identical tables, neither having a primary key, the only difference being that the first (table `auto_0`) is created with `sql_generate_invisible_primary_key` set to `OFF`, and the second (`auto_1`) after setting it to `ON`, as shown here:

```
mysql> SELECT @@sql_generate_invisible_primary_key;
+-------------------------------------+
| @@sql_generate_invisible_primary_key |
+-------------------------------------+
|                                   0 |
+-------------------------------------+
1 row in set (0.00 sec)

mysql> CREATE TABLE auto_0 (c1 VARCHAR(50), c2 INT);
Query OK, 0 rows affected (0.02 sec)

mysql> SET sql_generate_invisible_primary_key=ON;
Query OK, 0 rows affected (0.00 sec)

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

mysql> CREATE TABLE auto_1 (c1 VARCHAR(50), c2 INT);
Query OK, 0 rows affected (0.04 sec)
```

Compare the output of these `SHOW CREATE TABLE` statements to see the difference in how the tables were actually created:

```
mysql> SHOW CREATE TABLE auto_0\G
*************************** 1. row ***************************
       Table: auto_0
Create Table: CREATE TABLE `auto_0` (
  `c1` varchar(50) DEFAULT NULL,
  `c2` int DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

mysql> SHOW CREATE TABLE auto_1\G
*************************** 1. row ***************************
       Table: auto_1
Create Table: CREATE TABLE `auto_1` (
  `my_row_id` bigint unsigned NOT NULL AUTO_INCREMENT /*!80023 INVISIBLE */,
  `c1` varchar(50) DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_row_id`)
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

Since `auto_1` had no primary key specified by the `CREATE TABLE` statement used to create it, setting `sql_generate_invisible_primary_key = ON` causes MySQL to add both the invisible column `my_row_id` to this table and a primary key on that column. Since `sql_generate_invisible_primary_key` was `OFF` at the time that `auto_0` was created, no such additions were performed on that table.

When a primary key is added to a table by the server, the column and key name is always `my_row_id`. For this reason, when when enabling generated invisible primary keys in this way, you cannot create a table having a column named `my_row_id` unless the table creation statement also specifies an explicit primary key. (You are not required to name the column or key `my_row_id` in such cases.)

`my_row_id` is an invisible column, which means it is not shown in the output of `SELECT *` or `TABLE`; the column must be selected explicitly by name. See Section 13.1.20.10, “Invisible Columns”.

When GIPKs are enabled, a generated primary key cannot be altered other than to switch it between `VISIBLE` and `INVISIBLE`. To make the generated invisible primary key on `auto_1` visible, execute this `ALTER TABLE` statement:

```
mysql> ALTER TABLE auto_1 ALTER COLUMN my_row_id SET VISIBLE;
Query OK, 0 rows affected (0.02 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SHOW CREATE TABLE auto_1\G
*************************** 1. row ***************************
       Table: auto_1
Create Table: CREATE TABLE `auto_1` (
  `my_row_id` bigint unsigned NOT NULL AUTO_INCREMENT,
  `c1` varchar(50) DEFAULT NULL,
  `c2` int DEFAULT NULL,
  PRIMARY KEY (`my_row_id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.01 sec)
```

To make this generated primary key invisible again, issue `ALTER TABLE auto_1 ALTER COLUMN my_row_id SET INVISIBLE`.

A generated invisible primary key is always invisible by default.

Whenever GIPKs are enabled, you cannot drop a generated primary key if either of the following 2 conditions would result:

- The table is left with no primary key.
- The primary key is dropped, but not the primary key column.

The effects of `sql_generate_invisible_primary_key` apply to tables using the `InnoDB` storage engine only. You can use an `ALTER TABLE` statement to change the storage engine used by a table that has a generated invisible primary key; in this case, the primary key and column remain in place, but the table and key no longer receive any special treatment.

By default, GIPKs are shown in the output of `SHOW CREATE TABLE`, `SHOW COLUMNS`, and `SHOW INDEX`, and are visible in the Information Schema `COLUMNS` and `STATISTICS` tables. You can cause generated invisible primary keys to be hidden instead in such cases by setting the `show_gipk_in_create_table_and_information_schema` system variable to `OFF`. By default, this variable is `ON`, as shown here:

```
mysql> SELECT @@show_gipk_in_create_table_and_information_schema;
+-------------------------------------------------+
| @@show_gipk_in_create_table_and_information_schema |
+-------------------------------------------------+
|                                               1 |
```

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

As can be seen from the following query against the `COLUMNS` table, `my_row_id` is visible among the columns of `auto_1`:

```
mysql> SELECT COLUMN_NAME, ORDINAL_POSITION, DATA_TYPE, COLUMN_KEY
    -> FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_NAME = "auto_1";
+-------------+------------------+-----------+------------+
| COLUMN_NAME | ORDINAL_POSITION | DATA_TYPE | COLUMN_KEY |
+-------------+------------------+-----------+------------+
| my_row_id   |                1 | bigint    | PRI        |
| c1          |                2 | varchar   |            |
| c2          |                3 | int       |            |
+-------------+------------------+-----------+------------+
3 rows in set (0.01 sec)
```

After `show_gipk_in_create_table_and_information_schema` is set to `OFF`, `my_row_id` can no longer be seen in the `COLUMNS` table, as shown here:

```
mysql> SET show_gipk_in_create_table_and_information_schema = OFF;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT @@show_gipk_in_create_table_and_information_schema;
+-------------------------------------------------+
| @@show_gipk_in_create_table_and_information_schema |
+-------------------------------------------------+
|                                               0 |
+-------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT COLUMN_NAME, ORDINAL_POSITION, DATA_TYPE, COLUMN_KEY
    -> FROM INFORMATION_SCHEMA.COLUMNS
    -> WHERE TABLE_NAME = "auto_1";
+-------------+------------------+-----------+------------+
| COLUMN_NAME | ORDINAL_POSITION | DATA_TYPE | COLUMN_KEY |
+-------------+------------------+-----------+------------+
| c1          |                2 | varchar   |            |
| c2          |                3 | int       |            |
+-------------+------------------+-----------+------------+
2 rows in set (0.00 sec)
```

The setting for `sql_generate_invisible_primary_key` is not replicated, and is ignored by replication applier threads. This means that the setting of this variable on the source has no effect on the replica. In MySQL 8.0.32 and later, you can cause the replica to add a GIPK for tables replicated without primary keys on a given replication channel using `REQUIRE_TABLE_PRIMARY_KEY_CHECK = GENERATE` as part of a `CHANGE REPLICATION SOURCE TO` statement.

GIPKs work with row-based replication of `CREATE TABLE ... SELECT`; the information written to the binary log for this statement in such cases includes the GIPK definition, and thus is replicated correctly. Statement-based replication of `CREATE TABLE ... SELECT` is not supported with `sql_generate_invisible_primary_key = ON`.

When creating or importing backups of installations where GIPKs are in use, it is possible to exclude generated invisible primary key columns and values. The `--skip-generated-invisible-primary-key` option for `mysqldump` causes GIPK information to be excluded in the program's output. If you are importing a dump file that contains generated invisible primary keys and values, you can also use `--skip-generated-invisible-primary-key` with `mysqlpump` to cause these to be suppressed (and thus not imported).

13.1.20.12 Setting NDB Comment Options

- NDB_COLUMN Options
- NDB_TABLE Options

It is possible to set a number of options specific to NDB Cluster in the table comment or column comments of an NDB table. Table-level options for controlling read from any replica and partition balance can be embedded in a table comment using NDB_TABLE.

NDB_COLUMN can be used in a column comment to set the size of the blob parts table column used for storing parts of blob values by NDB to its maximum. This works for BLOB, MEDIUMBLOB, LONGBLOB, TEXT, MEDIUMTEXT, LONGTEXT, and JSON columns. Beginning with NDB 8.0.30, a column comment can also be used to control the inline size of a blob column. NDB_COLUMN comments do not support TINYBLOB or TINYTEXT columns, since these have an inline part (only) of fixed size, and no separate parts to store elsewhere.

NDB_TABLE can be used in a table comment to set options relating to partition balance and whether the table is fully replicated, among others.

The remainder of this section describes these options and their use.

NDB_COLUMN Options

In NDB Cluster, a column comment in a CREATE TABLE or ALTER TABLE statement can also be used to specify an NDB_COLUMN option. Beginning with version 8.0.30, NDB supports two column comment options BLOB_INLINE_SIZE and MAX_BLOB_PART_SIZE. (Prior to NDB 8.0.30, only MAX_BLOB_PART_SIZE is supported.) Syntax for this option is shown here:

```
COMMENT 'NDB_COLUMN=speclist'

speclist := spec[,spec]

spec :=
    BLOB_INLINE_SIZE=value
  | MAX_BLOB_PART_SIZE[={0|1}]
```

BLOB_INLINE_SIZE specifies the number of bytes to be stored inline by the column; its expected value is an integer in the range 1 - 29980, inclusive. Setting a value greater than 29980 raises an error; setting a value less than 1 is allowed, but causes the default inline size for the column type to be used.

You should be aware that the maximum value for this option is actually the maximum number of bytes that can be stored in one row of an NDB table; every column in the row contributes to this total.

You should also keep in mind, especially when working with TEXT columns, that the value set by MAX_BLOB_PART_SIZE or BLOB_INLINE_SIZE represents column size in bytes. It does not indicate the number of characters, which varies according to the character set and collation used by the column.

To see the effects of this option, first create a table with two BLOB columns, one (b1) with no extra options, and another (b2) with a setting for BLOB_INLINE_SIZE, as shown here:

```
mysql> CREATE TABLE t1 (
    ->    a INT NOT NULL PRIMARY KEY,
    ->    b1 BLOB,
    ->    b2 BLOB COMMENT 'NDB_COLUMN=BLOB_INLINE_SIZE=8000'
    ->  ) ENGINE NDB;
Query OK, 0 rows affected (0.32 sec)
```

You can see the BLOB_INLINE_SIZE settings for the BLOB columns by querying the ndbinfo.blobs table, like this:

```
mysql> SELECT
    ->   column_name AS 'Column Name',
    ->   inline_size AS 'Inline Size',
    ->   part_size AS 'Blob Part Size'
    -> FROM ndbinfo.blobs
    -> WHERE table_name = 't1';
+-------------+-------------+----------------+
| Column Name | Inline Size | Blob Part Size |
+-------------+-------------+----------------+
```

```
| b1          |         256 |           2000 |
| b2          |        8000 |           2000 |
+-------------+-------------+----------------+
2 rows in set (0.01 sec)
```

You can also check the output from the `ndb_desc` utility, as shown here, with the relevant lines displayed using emphasized text:

```
$> ndb_desc -d test t1
-- t --
Version: 1
Fragment type: HashMapPartition
K Value: 6
Min load factor: 78
Max load factor: 80
Temporary table: no
Number of attributes: 3
Number of primary keys: 1
Length of frm data: 945
Max Rows: 0
Row Checksum: 1
Row GCI: 1
SingleUserMode: 0
ForceVarPart: 1
PartitionCount: 2
FragmentCount: 2
PartitionBalance: FOR_RP_BY_LDM
ExtraRowGciBits: 0
ExtraRowAuthorBits: 0
TableStatus: Retrieved
Table options: readbackup
HashMap: DEFAULT-HASHMAP-3840-2
-- Attributes --
a Int PRIMARY KEY DISTRIBUTION KEY AT=FIXED ST=MEMORY
b1 Blob(256,2000,0) NULL AT=MEDIUM_VAR ST=MEMORY BV=2 BT=NDB$BLOB_64_1
b2 Blob(8000,2000,0) NULL AT=MEDIUM_VAR ST=MEMORY BV=2 BT=NDB$BLOB_64_2
-- Indexes --
PRIMARY KEY(a) - UniqueHashIndex
PRIMARY(a) - OrderedIndex

NDBT_ProgramExit: 0 - OK
```

For `MAX_BLOB_PART_SIZE`, the `=` sign and the value following it are optional. Using any value other than 0 or 1 results in a syntax error.

The effect of using `MAX_BLOB_PART_SIZE` in a column comment is to set the blob part size of a `TEXT` or `BLOB` column to the maximum number of bytes supported for this by `NDB` (13948). This option can be applied to any blob column type supported by MySQL except `TINYBLOB` or `TINYTEXT` (`BLOB`, `MEDIUMBLOB`, `LONGBLOB`, `TEXT`, `MEDIUMTEXT`, `LONGTEXT`). Unlike `BLOB_INLINE_SIZE`, `MAX_BLOB_PART_SIZE` has no effect on `JSON` columns.

To see the effects of this option, we first run the following SQL statement in the `mysql` client to create a table with two `BLOB` columns, one (`c1`) with no extra options, and another (`c2`) with `MAX_BLOB_PART_SIZE`:

```
mysql> CREATE TABLE test.t2 (
    ->   p INT PRIMARY KEY,
    ->   c1 BLOB,
    ->   c2 BLOB COMMENT 'NDB_COLUMN=MAX_BLOB_PART_SIZE'
    -> ) ENGINE NDB;
Query OK, 0 rows affected (0.32 sec)
```

From the system shell, run the `ndb_desc` utility to obtain information about the table just created, as shown in this example:

```
$> ndb_desc -d test t2
-- t --
Version: 1
Fragment type: HashMapPartition
```

```
K Value: 6
Min load factor: 78
Max load factor: 80
Temporary table: no
Number of attributes: 3
Number of primary keys: 1
Length of frm data: 324
Row Checksum: 1
Row GCI: 1
SingleUserMode: 0
ForceVarPart: 1
FragmentCount: 2
ExtraRowGciBits: 0
ExtraRowAuthorBits: 0
TableStatus: Retrieved
HashMap: DEFAULT-HASHMAP-3840-2
-- Attributes --
p Int PRIMARY KEY DISTRIBUTION KEY AT=FIXED ST=MEMORY
c1 Blob(256,2000,0) NULL AT=MEDIUM_VAR ST=MEMORY BV=2 BT=NDB$BLOB_22_1
c2 Blob(256,13948,0) NULL AT=MEDIUM_VAR ST=MEMORY BV=2 BT=NDB$BLOB_22_2
-- Indexes --
PRIMARY KEY(p) - UniqueHashIndex
PRIMARY(p) - OrderedIndex
```

Column information in the output is listed under `Attributes`; for columns `c1` and `c2` it is displayed here in emphasized text. For `c1`, the blob part size is 2000, the default value; for `c2`, it is 13948, as set by `MAX_BLOB_PART_SIZE`.

You can also query the `ndbinfo.blobs` table to see this, as shown here:

```
mysql> SELECT
    ->   column_name AS 'Column Name',
    ->   inline_size AS 'Inline Size',
    ->   part_size AS 'Blob Part Size'
    -> FROM ndbinfo.blobs
    -> WHERE table_name = 't2';
+-------------+-------------+----------------+
| Column Name | Inline Size | Blob Part Size |
+-------------+-------------+----------------+
| c1          |         256 |           2000 |
| c2          |         256 |          13948 |
+-------------+-------------+----------------+
2 rows in set (0.00 sec)
```

You can change the blob part size for a given blob column of an `NDB` table using an `ALTER TABLE` statement such as this one, and verifying the changes afterwards using `SHOW CREATE TABLE`:

```
mysql> ALTER TABLE test.t2
    ->    DROP COLUMN c1,
    ->     ADD COLUMN c1 BLOB COMMENT 'NDB_COLUMN=MAX_BLOB_PART_SIZE',
    ->     CHANGE COLUMN c2 c2 BLOB AFTER c1;
Query OK, 0 rows affected (0.47 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SHOW CREATE TABLE test.t2\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t2` (
  `p` int(11) NOT NULL,
  `c1` blob COMMENT 'NDB_COLUMN=MAX_BLOB_PART_SIZE',
  `c2` blob,
  PRIMARY KEY (`p`)
) ENGINE=ndbcluster DEFAULT CHARSET=latin1
1 row in set (0.00 sec)

mysql> EXIT
Bye
```

The output of `ndb_desc` shows that the blob part sizes of the columns have been changed as expected:

```
$> ndb_desc -d test t2
-- t --
Version: 16777220
Fragment type: HashMapPartition
K Value: 6
Min load factor: 78
Max load factor: 80
Temporary table: no
Number of attributes: 3
Number of primary keys: 1
Length of frm data: 324
Row Checksum: 1
Row GCI: 1
SingleUserMode: 0
ForceVarPart: 1
FragmentCount: 2
ExtraRowGciBits: 0
ExtraRowAuthorBits: 0
TableStatus: Retrieved
HashMap: DEFAULT-HASHMAP-3840-2
-- Attributes --
p Int PRIMARY KEY DISTRIBUTION KEY AT=FIXED ST=MEMORY
c1 Blob(256,13948,0) NULL AT=MEDIUM_VAR ST=MEMORY BV=2 BT=NDB$BLOB_26_1
c2 Blob(256,2000,0) NULL AT=MEDIUM_VAR ST=MEMORY BV=2 BT=NDB$BLOB_26_2
-- Indexes --
PRIMARY KEY(p) - UniqueHashIndex
PRIMARY(p) - OrderedIndex

NDBT_ProgramExit: 0 - OK
```

You can also see the change by running the query against `ndbinfo.blobs` again:

```
mysql> SELECT
    ->   column_name AS 'Column Name',
    ->   inline_size AS 'Inline Size',
    ->   part_size AS 'Blob Part Size'
    -> FROM ndbinfo.blobs
    -> WHERE table_name = 't2';
+-------------+-------------+----------------+
| Column Name | Inline Size | Blob Part Size |
+-------------+-------------+----------------+
| c1          |         256 |          13948 |
| c2          |         256 |           2000 |
+-------------+-------------+----------------+
2 rows in set (0.00 sec)
```

It is possible to set both `BLOB_INLINE_SIZE` and `MAX_BLOB_PART_SIZE` for a blob column, as shown in this `CREATE TABLE` statement:

```
mysql> CREATE TABLE test.t3 (
    ->   p INT NOT NULL PRIMARY KEY,
    ->   c1 JSON,
    ->   c2 JSON COMMENT 'NDB_COLUMN=BLOB_INLINE_SIZE=5000,MAX_BLOB_PART_SIZE'
    -> ) ENGINE NDB;
Query OK, 0 rows affected (0.28 sec)
```

Querying the `blobs` table shows us that the statement worked as expected:

```
mysql> SELECT
    ->   column_name AS 'Column Name',
    ->   inline_size AS 'Inline Size',
    ->   part_size AS 'Blob Part Size'
    -> FROM ndbinfo.blobs
    -> WHERE table_name = 't3';
+-------------+-------------+----------------+
| Column Name | Inline Size | Blob Part Size |
+-------------+-------------+----------------+
| c1          |        4000 |           8100 |
| c2          |        5000 |           8100 |
+-------------+-------------+----------------+
2 rows in set (0.00 sec)
```

You can also verify that the statement worked by checking the output of `ndb_desc`.

Changing a column's blob part size must be done using a copying `ALTER TABLE`; this operation cannot be performed online (see Section 23.6.12, “Online Operations with ALTER TABLE in NDB Cluster”).

For more information about how `NDB` stores columns of blob types, see String Type Storage Requirements.

NDB_TABLE Options

In MySQL NDB Cluster, the table comment in a `CREATE TABLE` or `ALTER TABLE` statement can also be used to specify an `NDB_TABLE` option, which consists of one or more name-value pairs, separated by commas if need be, following the string `NDB_TABLE=`. Complete syntax for names and values syntax is shown here:

```
COMMENT="NDB_TABLE=ndb_table_option[,ndb_table_option[,...]]"

ndb_table_option: {
    NOLOGGING={1 | 0}
  | READ_BACKUP={1 | 0}
  | PARTITION_BALANCE={FOR_RP_BY_NODE | FOR_RA_BY_NODE | FOR_RP_BY_LDM
                      | FOR_RA_BY_LDM | FOR_RA_BY_LDM_X_2
                      | FOR_RA_BY_LDM_X_3 | FOR_RA_BY_LDM_X_4}
  | FULLY_REPLICATED={1 | 0}
}
```

Spaces are not permitted within the quoted string. The string is case-insensitive.

The four `NDB` table options that can be set as part of a comment in this way are described in more detail in the next few paragraphs.

`NOLOGGING`: Using 1 corresponds to having `ndb_table_no_logging` enabled, but has no actual effect. Provided as a placeholder, mostly for completeness of `ALTER TABLE` statements.

`READ_BACKUP`: Setting this option to 1 has the same effect as though `ndb_read_backup` were enabled; enables reading from any replica. Doing so greatly improves the performance of reads from the table at a relatively small cost to write performance. Beginning with NDB 8.0.19, 1 is the default for `READ_BACKUP`, and the default for `ndb_read_backup` is `ON` (previously, read from any replica was disabled by default).

You can set `READ_BACKUP` for an existing table online, using an `ALTER TABLE` statement similar to one of those shown here:

```
ALTER TABLE ... ALGORITHM=INPLACE, COMMENT="NDB_TABLE=READ_BACKUP=1";

ALTER TABLE ... ALGORITHM=INPLACE, COMMENT="NDB_TABLE=READ_BACKUP=0";
```

For more information about the `ALGORITHM` option for `ALTER TABLE`, see Section 23.6.12, “Online Operations with ALTER TABLE in NDB Cluster”.

`PARTITION_BALANCE`: Provides additional control over assignment and placement of partitions. The following four schemes are supported:

1. `FOR_RP_BY_NODE`: One partition per node.

   Only one LDM on each node stores a primary partition. Each partition is stored in the same LDM (same ID) on all nodes.

2. `FOR_RA_BY_NODE`: One partition per node group.

   Each node stores a single partition, which can be either a primary replica or a backup replica. Each partition is stored in the same LDM on all nodes.

3. FOR_RP_BY_LDM: One partition for each LDM on each node; the default.

   This is the setting used if READ_BACKUP is set to 1.

4. FOR_RA_BY_LDM: One partition per LDM in each node group.

   These partitions can be primary or backup partitions.

5. FOR_RA_BY_LDM_X_2: Two partitions per LDM in each node group.

   These partitions can be primary or backup partitions.

6. FOR_RA_BY_LDM_X_3: Three partitions per LDM in each node group.

   These partitions can be primary or backup partitions.

7. FOR_RA_BY_LDM_X_4: Four partitions per LDM in each node group.

   These partitions can be primary or backup partitions.

PARTITION_BALANCE is the preferred interface for setting the number of partitions per table. Using MAX_ROWS to force the number of partitions is deprecated but continues to be supported for backward compatibility; it is subject to removal in a future release of MySQL NDB Cluster. (Bug #81759, Bug #23544301)

FULLY_REPLICATED controls whether the table is fully replicated, that is, whether each data node has a complete copy of the table. To enable full replication of the table, use FULLY_REPLICATED=1.

This setting can also be controlled using the ndb_fully_replicated system variable. Setting it to ON enables the option by default for all new NDB tables; the default is OFF. The ndb_data_node_neighbour system variable is also used for fully replicated tables, to ensure that when a fully replicated table is accessed, we access the data node which is local to this MySQL Server.

An example of a CREATE TABLE statement using such a comment when creating an NDB table is shown here:

```
mysql> CREATE TABLE t1 (
    >     c1 INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    >     c2 VARCHAR(100),
    >     c3 VARCHAR(100) )
    > ENGINE=NDB
    >
COMMENT="NDB_TABLE=READ_BACKUP=0,PARTITION_BALANCE=FOR_RP_BY_NODE";
```

The comment is displayed as part of the output of SHOW CREATE TABLE. The text of the comment is also available from querying the MySQL Information Schema TABLES table, as in this example:

```
mysql> SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT
    > FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME="t1"\G
*************************** 1. row ***************************
   TABLE_NAME: t1
 TABLE_SCHEMA: test
TABLE_COMMENT: NDB_TABLE=READ_BACKUP=0,PARTITION_BALANCE=FOR_RP_BY_NODE
1 row in set (0.01 sec)
```

This comment syntax is also supported with ALTER TABLE statements for NDB tables, as shown here:

```
mysql> ALTER TABLE t1 COMMENT="NDB_TABLE=PARTITION_BALANCE=FOR_RA_BY_NODE";
Query OK, 0 rows affected (0.40 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Beginning with NDB 8.0.21, the TABLE_COMMENT column displays the comment that is required to re-create the table as it is following the ALTER TABLE statement, like this:

```
mysql> SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT
```

```
    ->     FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME="t1"\G
*************************** 1. row ***************************
   TABLE_NAME: t1
 TABLE_SCHEMA: test
TABLE_COMMENT: NDB_TABLE=READ_BACKUP=0,PARTITION_BALANCE=FOR_RP_BY_NODE
1 row in set (0.01 sec)
```

```
mysql> SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT
     > FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME="t1";
+------------+--------------+------------------------------------------------+
| TABLE_NAME | TABLE_SCHEMA | TABLE_COMMENT                                  |
+------------+--------------+------------------------------------------------+
| t1         | c            | NDB_TABLE=PARTITION_BALANCE=FOR_RA_BY_NODE     |
| t1         | d            |                                                |
+------------+--------------+------------------------------------------------+
2 rows in set (0.01 sec)
```

Keep in mind that a table comment used with `ALTER TABLE` replaces any existing comment which the table might have.

```
mysql> ALTER TABLE t1 COMMENT="NDB_TABLE=PARTITION_BALANCE=FOR_RA_BY_NODE";
Query OK, 0 rows affected (0.40 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SELECT TABLE_NAME, TABLE_SCHEMA, TABLE_COMMENT
     > FROM INFORMATION_SCHEMA.TABLES WHERE TABLE_NAME="t1";
+------------+--------------+------------------------------------------------+
| TABLE_NAME | TABLE_SCHEMA | TABLE_COMMENT                                  |
+------------+--------------+------------------------------------------------+
| t1         | c            | NDB_TABLE=PARTITION_BALANCE=FOR_RA_BY_NODE     |
| t1         | d            |                                                |
+------------+--------------+------------------------------------------------+
2 rows in set (0.01 sec)
```

Prior to NDB 8.0.21, the table comment used with `ALTER TABLE` replaced any existing comment which the table might have had. This meant that (for example) the `READ_BACKUP` value was not carried over to the new comment set by the `ALTER TABLE` statement, and that any unspecified values reverted to their defaults. (BUG#30428829) There was thus no longer any way using SQL to retrieve the value previously set for the comment. To keep comment values from reverting to their defaults, it was necessary to preserve any such values from the existing comment string and include them in the comment passed to `ALTER TABLE`.

You can also see the value of the `PARTITION_BALANCE` option in the output of `ndb_desc`. `ndb_desc` also shows whether the `READ_BACKUP` and `FULLY_REPLICATED` options are set for the table. See the description of this program for more information.

13.1.21 CREATE TABLESPACE Statement

```
CREATE [UNDO] TABLESPACE tablespace_name

  InnoDB and NDB:
    [ADD DATAFILE 'file_name']
    [AUTOEXTEND_SIZE [=] value]

  InnoDB only:
    [FILE_BLOCK_SIZE = value]
    [ENCRYPTION [=] {'Y' | 'N'}]

  NDB only:
    USE LOGFILE GROUP logfile_group
    [EXTENT_SIZE [=] extent_size]
    [INITIAL_SIZE [=] initial_size]
    [MAX_SIZE [=] max_size]
    [NODEGROUP [=] nodegroup_id]
    [WAIT]
    [COMMENT [=] 'string']

  InnoDB and NDB:
    [ENGINE [=] engine_name]
```

```
Reserved for future use:
  [ENGINE_ATTRIBUTE [=] 'string']
```

This statement is used to create a tablespace. The precise syntax and semantics depend on the storage engine used. In standard MySQL releases, this is always an `InnoDB` tablespace. MySQL NDB Cluster also supports tablespaces using the `NDB` storage engine.

- Considerations for InnoDB
- Considerations for NDB Cluster
- Options
- Notes
- InnoDB Examples
- NDB Example

Considerations for InnoDB

`CREATE TABLESPACE` syntax is used to create general tablespaces or undo tablespaces. The `UNDO` keyword, introduced in MySQL 8.0.14, must be specified to create an undo tablespace.

A general tablespace is a shared tablespace. It can hold multiple tables, and supports all table row formats. General tablespaces can be created in a location relative to or independent of the data directory.

After creating an `InnoDB` general tablespace, use `CREATE TABLE *tbl_name* ... TABLESPACE [=] *tablespace_name*` or `ALTER TABLE *tbl_name* TABLESPACE [=] *tablespace_name*` to add tables to the tablespace. For more information, see Section 15.6.3.3, “General Tablespaces”.

Undo tablespaces contain undo logs. Undo tablespaces can be created in a chosen location by specifying a fully qualified data file path. For more information, see Section 15.6.3.4, “Undo Tablespaces”.

Considerations for NDB Cluster

This statement is used to create a tablespace, which can contain one or more data files, providing storage space for NDB Cluster Disk Data tables (see Section 23.6.11, “NDB Cluster Disk Data Tables”). One data file is created and added to the tablespace using this statement. Additional data files may be added to the tablespace by using the `ALTER TABLESPACE` statement (see Section 13.1.10, “ALTER TABLESPACE Statement”).

Note

All NDB Cluster Disk Data objects share the same namespace. This means that *each Disk Data object* must be uniquely named (and not merely each Disk Data object of a given type). For example, you cannot have a tablespace and a log file group with the same name, or a tablespace and a data file with the same name.

A log file group of one or more `UNDO` log files must be assigned to the tablespace to be created with the `USE LOGFILE GROUP` clause. *`logfile_group`* must be an existing log file group created with `CREATE LOGFILE GROUP` (see Section 13.1.16, “CREATE LOGFILE GROUP Statement”). Multiple tablespaces may use the same log file group for `UNDO` logging.

When setting `EXTENT_SIZE` or `INITIAL_SIZE`, you may optionally follow the number with a one-letter abbreviation for an order of magnitude, similar to those used in `my.cnf`. Generally, this is one of the letters `M` (for megabytes) or `G` (for gigabytes).

`INITIAL_SIZE` and `EXTENT_SIZE` are subject to rounding as follows:

- `EXTENT_SIZE` is rounded up to the nearest whole multiple of 32K.
- `INITIAL_SIZE` is rounded *down* to the nearest whole multiple of 32K; this result is rounded up to the nearest whole multiple of `EXTENT_SIZE` (after any rounding).

Note

`NDB` reserves 4% of a tablespace for data node restart operations. This reserved space cannot be used for data storage.

The rounding just described is done explicitly, and a warning is issued by the MySQL Server when any such rounding is performed. The rounded values are also used by the NDB kernel for calculating `INFORMATION_SCHEMA.FILES` column values and other purposes. However, to avoid an unexpected result, we suggest that you always use whole multiples of 32K in specifying these options.

When `CREATE TABLESPACE` is used with `ENGINE [=] NDB`, a tablespace and associated data file are created on each Cluster data node. You can verify that the data files were created and obtain information about them by querying the Information Schema `FILES` table. (See the example later in this section.)

(See Section 26.3.15, “The INFORMATION_SCHEMA FILES Table”.)

Options

- `ADD DATAFILE`: Defines the name of a tablespace data file. This option is always required when creating an `NDB` tablespace; for `InnoDB` in MySQL 8.0.14 and later, it is required only when creating an undo tablespace. The *`file_name`*, including any specified path, must be quoted with single or double quotation marks. File names (not counting the file extension) and directory names must be at least one byte in length. Zero length file names and directory names are not supported.

  Because there are considerable differences in how `InnoDB` and `NDB` treat data files, the two storage engines are covered separately in the discussion that follows.

  InnoDB data files. An `InnoDB` tablespace supports only a single data file, whose name must include a `.ibd` extension.

  To place an `InnoDB` general tablespace data file in a location outside of the data directory, include a fully qualified path or a path relative to the data directory. Only a fully qualified path is permitted for undo tablespaces. If you do not specify a path, a general tablespace is created in the data directory. An undo tablespace created without specifying a path is created in the directory defined by the `innodb_undo_directory` variable. If the `innodb_undo_directory` variable is undefined, undo tablespaces are created in the data directory.

  To avoid conflicts with implicitly created file-per-table tablespaces, creating an `InnoDB` general tablespace in a subdirectory under the data directory is not supported. When creating a general tablespace or undo tablespace outside of the data directory, the directory must exist and must be known to `InnoDB` prior to creating the tablespace. To make a directory known to `InnoDB`, add it to the `innodb_directories` value or to one of the variables whose values are appended to the `innodb_directories` value. `innodb_directories` is a read-only variable. Configuring it requires restarting the server.

  If the `ADD DATAFILE` clause is not specified when creating an `InnoDB` tablespace, a tablespace data file with a unique file name is created implicitly. The unique file name is a 128 bit UUID formatted into five groups of hexadecimal numbers separated by dashes (*`aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee`*). A file extension is added if required by the storage engine. An `.ibd` file extension is added for `InnoDB` general tablespace data files. In a replication environment, the data file name created on the replication source server is not the same as the data file name created on the replica.

As of MySQL 8.0.17, the `ADD DATAFILE` clause does not permit circular directory references when creating an `InnoDB` tablespace. For example, the circular directory reference (`/../`) in the following statement is not permitted:

```
CREATE TABLESPACE ts1 ADD DATAFILE ts1.ibd 'any_directory/../ts1.ibd';
```

An exception to this restriction exists on Linux, where a circular directory reference is permitted if the preceding directory is a symbolic link. For example, the data file path in the example above is permitted if *`any_directory`* is a symbolic link. (It is still permitted for data file paths to begin with '`../`'.)

NDB data files. An `NDB` tablespace supports multiple data files which can have any legal file names; more data files can be added to an NDB Cluster tablespace following its creation by using an `ALTER TABLESPACE` statement.

An `NDB` tablespace data file is created by default in the data node file system directory—that is, the directory named `ndb_nodeid_fs/TS` under the data node's data directory (`DataDir`), where *`nodeid`* is the data node's `NodeId`. To place the data file in a location other than the default, include an absolute directory path or a path relative to the default location. If the directory specified does not exist, `NDB` attempts to create it; the system user account under which the data node process is running must have the appropriate permissions to do so.

Note

When determining the path used for a data file, `NDB` does not expand the `~` (tilde) character.

When multiple data nodes are run on the same physical host, the following considerations apply:

- You cannot specify an absolute path when creating a data file.
- It is not possible to create tablespace data files outside the data node file system directory, unless each data node has a separate data directory.
- If each data node has its own data directory, data files can be created anywhere within this directory.
- If each data node has its own data directory, it may also be possible to create a data file outside the node's data directory using a relative path, as long as this path resolves to a unique location on the host file system for each data node running on that host.

- `FILE_BLOCK_SIZE`: This option—which is specific to `InnoDB` general tablespaces, and is ignored by `NDB`—defines the block size for the tablespace data file. Values can be specified in bytes or kilobytes. For example, an 8 kilobyte file block size can be specified as 8192 or 8K. If you do not specify this option, `FILE_BLOCK_SIZE` defaults to the `innodb_page_size` value. `FILE_BLOCK_SIZE` is required when you intend to use the tablespace for storing compressed `InnoDB` tables (`ROW_FORMAT=COMPRESSED`). In this case, you must define the tablespace `FILE_BLOCK_SIZE` when creating the tablespace.

  If `FILE_BLOCK_SIZE` is equal the `innodb_page_size` value, the tablespace can contain only tables having an uncompressed row format (`COMPACT`, `REDUNDANT`, and `DYNAMIC`). Tables with a `COMPRESSED` row format have a different physical page size than uncompressed tables. Therefore, compressed tables cannot coexist in the same tablespace as uncompressed tables.

  For a general tablespace to contain compressed tables, `FILE_BLOCK_SIZE` must be specified, and the `FILE_BLOCK_SIZE` value must be a valid compressed page size in relation to the `innodb_page_size` value. Also, the physical page size of the compressed table (`KEY_BLOCK_SIZE`) must be equal to `FILE_BLOCK_SIZE/1024`. For example, if `innodb_page_size=16K`, and `FILE_BLOCK_SIZE=8K`, the `KEY_BLOCK_SIZE` of the table must be 8. For more information, see Section 15.6.3.3, “General Tablespaces”.

- `USE LOGFILE GROUP`: Required for `NDB`, this is the name of a log file group previously created using `CREATE LOGFILE GROUP`. Not supported for `InnoDB`, where it fails with an error.

- `EXTENT_SIZE`: This option is specific to NDB, and is not supported by InnoDB, where it fails with an error. `EXTENT_SIZE` sets the size, in bytes, of the extents used by any files belonging to the tablespace. The default value is 1M. The minimum size is 32K, and theoretical maximum is 2G, although the practical maximum size depends on a number of factors. In most cases, changing the extent size does not have any measurable effect on performance, and the default value is recommended for all but the most unusual situations.

  An *extent* is a unit of disk space allocation. One extent is filled with as much data as that extent can contain before another extent is used. In theory, up to 65,535 (64K) extents may used per data file; however, the recommended maximum is 32,768 (32K). The recommended maximum size for a single data file is 32G—that is, 32K extents × 1 MB per extent. In addition, once an extent is allocated to a given partition, it cannot be used to store data from a different partition; an extent cannot store data from more than one partition. This means, for example that a tablespace having a single datafile whose `INITIAL_SIZE` (described in the following item) is 256 MB and whose `EXTENT_SIZE` is 128M has just two extents, and so can be used to store data from at most two different disk data table partitions.

  You can see how many extents remain free in a given data file by querying the Information Schema `FILES` table, and so derive an estimate for how much space remains free in the file. For further discussion and examples, see Section 26.3.15, “The INFORMATION_SCHEMA FILES Table”.

- `INITIAL_SIZE`: This option is specific to `NDB`, and is not supported by `InnoDB`, where it fails with an error.

  The `INITIAL_SIZE` parameter sets the total size in bytes of the data file that was specific using `ADD DATAFILE`. Once this file has been created, its size cannot be changed; however, you can add more data files to the tablespace using `ALTER TABLESPACE ... ADD DATAFILE`.

  `INITIAL_SIZE` is optional; its default value is 134217728 (128 MB).

  On 32-bit systems, the maximum supported value for `INITIAL_SIZE` is 4294967296 (4 GB).

- `AUTOEXTEND_SIZE`: Ignored by MySQL prior to MySQL 8.0.23; From MySQL 8.0.23, defines the amount by which `InnoDB` extends the size of the tablespace when it becomes full. The setting must be a multiple of 4MB. The default setting is 0, which causes the tablespace to be extended according to the implicit default behavior. For more information, see Section 15.6.3.9, “Tablespace AUTOEXTEND_SIZE Configuration”.

  Has no effect in any release of MySQL NDB Cluster 8.0, regardless of the storage engine used.

- `MAX_SIZE`: Currently ignored by MySQL; reserved for possible future use. Has no effect in any release of MySQL 8.0 or MySQL NDB Cluster 8.0, regardless of the storage engine used.

- `NODEGROUP`: Currently ignored by MySQL; reserved for possible future use. Has no effect in any release of MySQL 8.0 or MySQL NDB Cluster 8.0, regardless of the storage engine used.

- `WAIT`: Currently ignored by MySQL; reserved for possible future use. Has no effect in any release of MySQL 8.0 or MySQL NDB Cluster 8.0, regardless of the storage engine used.

- `COMMENT`: Currently ignored by MySQL; reserved for possible future use. Has no effect in any release of MySQL 8.0 or MySQL NDB Cluster 8.0, regardless of the storage engine used.

- The `ENCRYPTION` clause enables or disables page-level data encryption for an `InnoDB` general tablespace. Encryption support for general tablespaces was introduced in MySQL 8.0.13.

  As of MySQL 8.0.16, if the `ENCRYPTION` clause is not specified, the `default_table_encryption` setting controls whether encryption is enabled. The `ENCRYPTION` clause overrides the `default_table_encryption` setting. However, if the `table_encryption_privilege_check`

variable is enabled, the TABLE_ENCRYPTION_ADMIN privilege is required to use an ENCRYPTION clause setting that differs from the default_table_encryption setting.

A keyring plugin must be installed and configured before an encryption-enabled tablespace can be created.

When a general tablespace is encrypted, all tables residing in the tablespace are encrypted. Likewise, a table created in an encrypted tablespace is encrypted.

For more information, see Section 15.13, “InnoDB Data-at-Rest Encryption”

- ENGINE: Defines the storage engine which uses the tablespace, where *engine_name* is the name of the storage engine. Currently, only the InnoDB storage engine is supported by standard MySQL 8.0 releases. MySQL NDB Cluster supports both NDB and InnoDB tablespaces. The value of the default_storage_engine system variable is used for ENGINE if the option is not specified.
- The ENGINE_ATTRIBUTE option (available as of MySQL 8.0.21) is used to specify tablespace attributes for primary storage engines. The option is reserved for future use.

  Permitted values are a string literal containing a valid JSON document or an empty string (''). Invalid JSON is rejected.

  ```
  CREATE TABLESPACE ts1 ENGINE_ATTRIBUTE='{"key":"value"}';
  ```

  ENGINE_ATTRIBUTE values can be repeated without error. In this case, the last specified value is used.

  ENGINE_ATTRIBUTE values are not checked by the server, nor are they cleared when the table's storage engine is changed.

Notes

- For the rules covering the naming of MySQL tablespaces, see Section 9.2, “Schema Object Names”. In addition to these rules, the slash character (“/”) is not permitted, nor can you use names beginning with innodb_, as this prefix is reserved for system use.
- Creation of temporary general tablespaces is not supported.
- General tablespaces do not support temporary tables.
- The TABLESPACE option may be used with CREATE TABLE or ALTER TABLE to assign an InnoDB table partition or subpartition to a file-per-table tablespace. All partitions must belong to the same storage engine. Assigning table partitions to shared InnoDB tablespaces is not supported. Shared tablespaces include the InnoDB system tablespace and general tablespaces.
- General tablespaces support the addition of tables of any row format using CREATE TABLE ... TABLESPACE. innodb_file_per_table does not need to be enabled.
- innodb_strict_mode is not applicable to general tablespaces. Tablespace management rules are strictly enforced independently of innodb_strict_mode. If CREATE TABLESPACE parameters are incorrect or incompatible, the operation fails regardless of the innodb_strict_mode setting. When a table is added to a general tablespace using CREATE TABLE ... TABLESPACE or ALTER TABLE ... TABLESPACE, innodb_strict_mode is ignored but the statement is evaluated as if innodb_strict_mode is enabled.
- Use DROP TABLESPACE to remove a tablespace. All tables must be dropped from a tablespace using DROP TABLE prior to dropping the tablespace. Before dropping an NDB Cluster tablespace you must also remove all its data files using one or more ALTER TABLESPACE ... DROP DATAFILE statements. See Section 23.6.11.1, “NDB Cluster Disk Data Objects”.
- All parts of an InnoDB table added to an InnoDB general tablespace reside in the general tablespace, including indexes and BLOB pages.

For an NDB table assigned to a tablespace, only those columns which are not indexed are stored on disk, and actually use the tablespace data files. Indexes and indexed columns for all NDB tables are always kept in memory.

- Similar to the system tablespace, truncating or dropping tables stored in a general tablespace creates free space internally in the general tablespace .ibd data file which can only be used for new InnoDB data. Space is not released back to the operating system as it is for file-per-table tablespaces.
- A general tablespace is not associated with any database or schema.
- ALTER TABLE ... DISCARD TABLESPACE and ALTER TABLE ...IMPORT TABLESPACE are not supported for tables that belong to a general tablespace.
- The server uses tablespace-level metadata locking for DDL that references general tablespaces. By comparison, the server uses table-level metadata locking for DDL that references file-per-table tablespaces.
- A generated or existing tablespace cannot be changed to a general tablespace.
- There is no conflict between general tablespace names and file-per-table tablespace names. The "/" character, which is present in file-per-table tablespace names, is not permitted in general tablespace names.
- mysqldump and mysqlpump do not dump InnoDB CREATE TABLESPACE statements.

InnoDB Examples

This example demonstrates creating a general tablespace and adding three uncompressed tables of different row formats.

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE 'ts1.ibd' ENGINE=INNODB;

mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1 ROW_FORMAT=REDUNDANT;

mysql> CREATE TABLE t2 (c1 INT PRIMARY KEY) TABLESPACE ts1 ROW_FORMAT=COMPACT;

mysql> CREATE TABLE t3 (c1 INT PRIMARY KEY) TABLESPACE ts1 ROW_FORMAT=DYNAMIC;
```

This example demonstrates creating a general tablespace and adding a compressed table. The example assumes a default innodb_page_size value of 16K. The FILE_BLOCK_SIZE of 8192 requires that the compressed table have a KEY_BLOCK_SIZE of 8.

```
mysql> CREATE TABLESPACE `ts2` ADD DATAFILE 'ts2.ibd' FILE_BLOCK_SIZE = 8192 Engine=InnoDB;

mysql> CREATE TABLE t4 (c1 INT PRIMARY KEY) TABLESPACE ts2 ROW_FORMAT=COMPRESSED KEY_BLOCK_SIZE=8;
```

This example demonstrates creating a general tablespace without specifying the ADD DATAFILE clause, which is optional as of MySQL 8.0.14.

```
mysql> CREATE TABLESPACE `ts3` ENGINE=INNODB;
```

This example demonstrates creating an undo tablespace.

```
mysql> CREATE UNDO TABLESPACE undo_003 ADD DATAFILE 'undo_003.ibu';
```

NDB Example

Suppose that you wish to create an NDB Cluster Disk Data tablespace named myts using a datafile named mydata-1.dat. An NDB tablespace always requires the use of a log file group consisting of one or more undo log files. For this example, we first create a log file group named mylg that contains

one undo long file named `myundo-1.dat`, using the `CREATE LOGFILE GROUP` statement shown here:

```
mysql> CREATE LOGFILE GROUP myg1
    ->     ADD UNDOFILE 'myundo-1.dat'
    ->     ENGINE=NDB;
Query OK, 0 rows affected (3.29 sec)
```

Now you can create the tablespace previously described using the following statement:

```
mysql> CREATE TABLESPACE myts
    ->     ADD DATAFILE 'mydata-1.dat'
    ->     USE LOGFILE GROUP mylg
    ->     ENGINE=NDB;
Query OK, 0 rows affected (2.98 sec)
```

You can now create a Disk Data table using a `CREATE TABLE` statement with the `TABLESPACE` and `STORAGE DISK` options, similar to what is shown here:

```
mysql> CREATE TABLE mytable (
    ->     id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->     lname VARCHAR(50) NOT NULL,
    ->     fname VARCHAR(50) NOT NULL,
    ->     dob DATE NOT NULL,
    ->     joined DATE NOT NULL,
    ->     INDEX(last_name, first_name)
    -> )
    ->     TABLESPACE myts STORAGE DISK
    ->     ENGINE=NDB;
Query OK, 0 rows affected (1.41 sec)
```

It is important to note that only the `dob` and `joined` columns from `mytable` are actually stored on disk, due to the fact that the `id`, `lname`, and `fname` columns are all indexed.

As mentioned previously, when `CREATE TABLESPACE` is used with `ENGINE [=] NDB`, a tablespace and associated data file are created on each NDB Cluster data node. You can verify that the data files were created and obtain information about them by querying the Information Schema `FILES` table, as shown here:

```
mysql> SELECT FILE_NAME, FILE_TYPE, LOGFILE_GROUP_NAME, STATUS, EXTRA
    ->     FROM INFORMATION_SCHEMA.FILES
    ->     WHERE TABLESPACE_NAME = 'myts';

+--------------+------------+--------------------+--------+----------------+
| file_name    | file_type  | logfile_group_name | status | extra          |
+--------------+------------+--------------------+--------+----------------+
| mydata-1.dat | DATAFILE   | mylg               | NORMAL | CLUSTER_NODE=5 |
| mydata-1.dat | DATAFILE   | mylg               | NORMAL | CLUSTER_NODE=6 |
| NULL         | TABLESPACE | mylg               | NORMAL | NULL           |
+--------------+------------+--------------------+--------+----------------+
3 rows in set (0.01 sec)
```

For additional information and examples, see Section 23.6.11.1, “NDB Cluster Disk Data Objects”.

13.1.22 CREATE TRIGGER Statement

```
CREATE
    [DEFINER = user]
    TRIGGER [IF NOT EXISTS] trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }
```

```
trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

This statement creates a new trigger. A trigger is a named database object that is associated with a table, and that activates when a particular event occurs for the table. The trigger becomes associated with the table named *`tbl_name`*, which must refer to a permanent table. You cannot associate a trigger with a `TEMPORARY` table or a view.

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

`IF NOT EXISTS` prevents an error from occurring if a trigger having the same name, on the same table, exists in the same schema. This option is supported with `CREATE TRIGGER` beginning with MySQL 8.0.29.

This section describes `CREATE TRIGGER` syntax. For additional discussion, see Section 25.3.1, “Trigger Syntax and Examples”.

`CREATE TRIGGER` requires the `TRIGGER` privilege for the table associated with the trigger. If the `DEFINER` clause is present, the privileges required depend on the *`user`* value, as discussed in Section 25.6, “Stored Object Access Control”. If binary logging is enabled, `CREATE TRIGGER` might require the `SUPER` privilege, as discussed in Section 25.7, “Stored Program Binary Logging”.

The `DEFINER` clause determines the security context to be used when checking access privileges at trigger activation time, as described later in this section.

`trigger_time` is the trigger action time. It can be `BEFORE` or `AFTER` to indicate that the trigger activates before or after each row to be modified.

Basic column value checks occur prior to trigger activation, so you cannot use `BEFORE` triggers to convert values inappropriate for the column type to valid values.

`trigger_event` indicates the kind of operation that activates the trigger. These *`trigger_event`* values are permitted:

- `INSERT`: The trigger activates whenever a new row is inserted into the table (for example, through `INSERT`, `LOAD DATA`, and `REPLACE` statements).
- `UPDATE`: The trigger activates whenever a row is modified (for example, through `UPDATE` statements).
- `DELETE`: The trigger activates whenever a row is deleted from the table (for example, through `DELETE` and `REPLACE` statements). `DROP TABLE` and `TRUNCATE TABLE` statements on the table do *not* activate this trigger, because they do not use `DELETE`. Dropping a partition does not activate `DELETE` triggers, either.

The *`trigger_event`* does not represent a literal type of SQL statement that activates the trigger so much as it represents a type of table operation. For example, an `INSERT` trigger activates not only for `INSERT` statements but also `LOAD DATA` statements because both statements insert rows into a table.

A potentially confusing example of this is the `INSERT INTO ... ON DUPLICATE KEY UPDATE ...` syntax: a `BEFORE INSERT` trigger activates for every row, followed by either an `AFTER INSERT` trigger or both the `BEFORE UPDATE` and `AFTER UPDATE` triggers, depending on whether there was a duplicate key for the row.

Note

Cascaded foreign key actions do not activate triggers.

It is possible to define multiple triggers for a given table that have the same trigger event and action time. For example, you can have two `BEFORE UPDATE` triggers for a table. By default, triggers that

have the same trigger event and action time activate in the order they were created. To affect trigger order, specify a *`trigger_order`* clause that indicates `FOLLOWS` or `PRECEDES` and the name of an existing trigger that also has the same trigger event and action time. With `FOLLOWS`, the new trigger activates after the existing trigger. With `PRECEDES`, the new trigger activates before the existing trigger.

`trigger_body` is the statement to execute when the trigger activates. To execute multiple statements, use the `BEGIN ... END` compound statement construct. This also enables you to use the same statements that are permitted within stored routines. See Section 13.6.1, “BEGIN ... END Compound Statement”. Some statements are not permitted in triggers; see Section 25.8, “Restrictions on Stored Programs”.

Within the trigger body, you can refer to columns in the subject table (the table associated with the trigger) by using the aliases `OLD` and `NEW`. `OLD.col_name` refers to a column of an existing row before it is updated or deleted. `NEW.col_name` refers to the column of a new row to be inserted or an existing row after it is updated.

Triggers cannot use `NEW.col_name` or use `OLD.col_name` to refer to generated columns. For information about generated columns, see Section 13.1.20.8, “CREATE TABLE and Generated Columns”.

MySQL stores the `sql_mode` system variable setting in effect when a trigger is created, and always executes the trigger body with this setting in force, *regardless of the current server SQL mode when the trigger begins executing*.

The `DEFINER` clause specifies the MySQL account to be used when checking access privileges at trigger activation time. If the `DEFINER` clause is present, the *`user`* value should be a MySQL account specified as *`'user_name'@'host_name'`*, `CURRENT_USER`, or `CURRENT_USER()`. The permitted *`user`* values depend on the privileges you hold, as discussed in Section 25.6, “Stored Object Access Control”. Also see that section for additional information about trigger security.

If the `DEFINER` clause is omitted, the default definer is the user who executes the `CREATE TRIGGER` statement. This is the same as specifying `DEFINER = CURRENT_USER` explicitly.

MySQL takes the `DEFINER` user into account when checking trigger privileges as follows:

- At `CREATE TRIGGER` time, the user who issues the statement must have the `TRIGGER` privilege.
- At trigger activation time, privileges are checked against the `DEFINER` user. This user must have these privileges:
  - The `TRIGGER` privilege for the subject table.
  - The `SELECT` privilege for the subject table if references to table columns occur using `OLD.col_name` or `NEW.col_name` in the trigger body.
  - The `UPDATE` privilege for the subject table if table columns are targets of `SET NEW.col_name = value` assignments in the trigger body.
  - Whatever other privileges normally are required for the statements executed by the trigger.

Within a trigger body, the `CURRENT_USER` function returns the account used to check privileges at trigger activation time. This is the `DEFINER` user, not the user whose actions caused the trigger to be activated. For information about user auditing within triggers, see Section 6.2.23, “SQL-Based Account Activity Auditing”.

If you use `LOCK TABLES` to lock a table that has triggers, the tables used within the trigger are also locked, as described in LOCK TABLES and Triggers.

For additional discussion of trigger use, see Section 25.3.1, “Trigger Syntax and Examples”.

13.1.23 CREATE VIEW Statement

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = user]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW view_name [(column_list)]
    AS select_statement
    [WITH [CASCADED | LOCAL] CHECK OPTION]
```

The `CREATE VIEW` statement creates a new view, or replaces an existing view if the `OR REPLACE` clause is given. If the view does not exist, `CREATE OR REPLACE VIEW` is the same as `CREATE VIEW`. If the view does exist, `CREATE OR REPLACE VIEW` replaces it.

For information about restrictions on view use, see Section 25.9, “Restrictions on Views”.

The *`select_statement`* is a `SELECT` statement that provides the definition of the view. (Selecting from the view selects, in effect, using the `SELECT` statement.) The *`select_statement`* can select from base tables, other views. Beginning with MySQL 8.0.19, the `SELECT` statement can use a `VALUES` statement as its source, or can be replaced with a `TABLE` statement, as with `CREATE TABLE ... SELECT`.

The view definition is “frozen” at creation time and is not affected by subsequent changes to the definitions of the underlying tables. For example, if a view is defined as `SELECT *` on a table, new columns added to the table later do not become part of the view, and columns dropped from the table result in an error when selecting from the view.

The `ALGORITHM` clause affects how MySQL processes the view. The `DEFINER` and `SQL SECURITY` clauses specify the security context to be used when checking access privileges at view invocation time. The `WITH CHECK OPTION` clause can be given to constrain inserts or updates to rows in tables referenced by the view. These clauses are described later in this section.

The `CREATE VIEW` statement requires the `CREATE VIEW` privilege for the view, and some privilege for each column selected by the `SELECT` statement. For columns used elsewhere in the `SELECT` statement, you must have the `SELECT` privilege. If the `OR REPLACE` clause is present, you must also have the `DROP` privilege for the view. If the `DEFINER` clause is present, the privileges required depend on the *`user`* value, as discussed in Section 25.6, “Stored Object Access Control”.

When a view is referenced, privilege checking occurs as described later in this section.

A view belongs to a database. By default, a new view is created in the default database. To create the view explicitly in a given database, use *`db_name.view_name`* syntax to qualify the view name with the database name:

```
CREATE VIEW test.v AS SELECT * FROM t;
```

Unqualified table or view names in the `SELECT` statement are also interpreted with respect to the default database. A view can refer to tables or views in other databases by qualifying the table or view name with the appropriate database name.

Within a database, base tables and views share the same namespace, so a base table and a view cannot have the same name.

Columns retrieved by the `SELECT` statement can be simple references to table columns, or expressions that use functions, constant values, operators, and so forth.

A view must have unique column names with no duplicates, just like a base table. By default, the names of the columns retrieved by the `SELECT` statement are used for the view column names. To define explicit names for the view columns, specify the optional *`column_list`* clause as a list of comma-separated identifiers. The number of names in *`column_list`* must be the same as the number of columns retrieved by the `SELECT` statement.

A view can be created from many kinds of SELECT statements. It can refer to base tables or other views. It can use joins, UNION, and subqueries. The SELECT need not even refer to any tables:

```
CREATE VIEW v_today (today) AS SELECT CURRENT_DATE;
```

The following example defines a view that selects two columns from another table as well as an expression calculated from those columns:

```
mysql> CREATE TABLE t (qty INT, price INT);
mysql> INSERT INTO t VALUES(3, 50);
mysql> CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
mysql> SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

A view definition is subject to the following restrictions:

- The SELECT statement cannot refer to system variables or user-defined variables.
- Within a stored program, the SELECT statement cannot refer to program parameters or local variables.
- The SELECT statement cannot refer to prepared statement parameters.
- Any table or view referred to in the definition must exist. If, after the view has been created, a table or view that the definition refers to is dropped, use of the view results in an error. To check a view definition for problems of this kind, use the CHECK TABLE statement.
- The definition cannot refer to a TEMPORARY table, and you cannot create a TEMPORARY view.
- You cannot associate a trigger with a view.
- Aliases for column names in the SELECT statement are checked against the maximum column length of 64 characters (not the maximum alias length of 256 characters).

ORDER BY is permitted in a view definition, but it is ignored if you select from a view using a statement that has its own ORDER BY.

For other options or clauses in the definition, they are added to the options or clauses of the statement that references the view, but the effect is undefined. For example, if a view definition includes a LIMIT clause, and you select from the view using a statement that has its own LIMIT clause, it is undefined which limit applies. This same principle applies to options such as ALL, DISTINCT, or SQL_SMALL_RESULT that follow the SELECT keyword, and to clauses such as INTO, FOR UPDATE, FOR SHARE, LOCK IN SHARE MODE, and PROCEDURE.

The results obtained from a view may be affected if you change the query processing environment by changing system variables:

```
mysql> CREATE VIEW v (mycol) AS SELECT 'abc';
Query OK, 0 rows affected (0.01 sec)

mysql> SET sql_mode = '';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT "mycol" FROM v;
+-------+
| mycol |
+-------+
| mycol |
+-------+
1 row in set (0.01 sec)
```

```
mysql> SET sql_mode = 'ANSI_QUOTES';
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT "mycol" FROM v;
+-------+
| mycol |
+-------+
| abc   |
+-------+
1 row in set (0.00 sec)
```

The `DEFINER` and `SQL SECURITY` clauses determine which MySQL account to use when checking access privileges for the view when a statement is executed that references the view. The valid `SQL SECURITY` characteristic values are `DEFINER` (the default) and `INVOKER`. These indicate that the required privileges must be held by the user who defined or invoked the view, respectively.

If the `DEFINER` clause is present, the *`user`* value should be a MySQL account specified as *`'user_name'@'host_name'`*, `CURRENT_USER`, or `CURRENT_USER()`. The permitted *`user`* values depend on the privileges you hold, as discussed in Section 25.6, “Stored Object Access Control”. Also see that section for additional information about view security.

If the `DEFINER` clause is omitted, the default definer is the user who executes the `CREATE VIEW` statement. This is the same as specifying `DEFINER = CURRENT_USER` explicitly.

Within a view definition, the `CURRENT_USER` function returns the view's `DEFINER` value by default. For views defined with the `SQL SECURITY INVOKER` characteristic, `CURRENT_USER` returns the account for the view's invoker. For information about user auditing within views, see Section 6.2.23, “SQL-Based Account Activity Auditing”.

Within a stored routine that is defined with the `SQL SECURITY DEFINER` characteristic, `CURRENT_USER` returns the routine's `DEFINER` value. This also affects a view defined within such a routine, if the view definition contains a `DEFINER` value of `CURRENT_USER`.

MySQL checks view privileges like this:

- At view definition time, the view creator must have the privileges needed to use the top-level objects accessed by the view. For example, if the view definition refers to table columns, the creator must have some privilege for each column in the select list of the definition, and the `SELECT` privilege for each column used elsewhere in the definition. If the definition refers to a stored function, only the privileges needed to invoke the function can be checked. The privileges required at function invocation time can be checked only as it executes: For different invocations, different execution paths within the function might be taken.
- The user who references a view must have appropriate privileges to access it (`SELECT` to select from it, `INSERT` to insert into it, and so forth.)
- When a view has been referenced, privileges for objects accessed by the view are checked against the privileges held by the view `DEFINER` account or invoker, depending on whether the `SQL SECURITY` characteristic is `DEFINER` or `INVOKER`, respectively.
- If reference to a view causes execution of a stored function, privilege checking for statements executed within the function depend on whether the function `SQL SECURITY` characteristic is `DEFINER` or `INVOKER`. If the security characteristic is `DEFINER`, the function runs with the privileges of the `DEFINER` account. If the characteristic is `INVOKER`, the function runs with the privileges determined by the view's `SQL SECURITY` characteristic.

Example: A view might depend on a stored function, and that function might invoke other stored routines. For example, the following view invokes a stored function `f()`:

```
CREATE VIEW v AS SELECT * FROM t WHERE t.id = f(t.name);
```

Suppose that `f()` contains a statement such as this:

```
IF name IS NULL then
  CALL p1();
ELSE
  CALL p2();
END IF;
```

The privileges required for executing statements within `f()` need to be checked when `f()` executes. This might mean that privileges are needed for `p1()` or `p2()`, depending on the execution path within `f()`. Those privileges must be checked at runtime, and the user who must possess the privileges is determined by the `SQL SECURITY` values of the view `v` and the function `f()`.

The `DEFINER` and `SQL SECURITY` clauses for views are extensions to standard SQL. In standard SQL, views are handled using the rules for `SQL SECURITY DEFINER`. The standard says that the definer of the view, which is the same as the owner of the view's schema, gets applicable privileges on the view (for example, `SELECT`) and may grant them. MySQL has no concept of a schema "owner", so MySQL adds a clause to identify the definer. The `DEFINER` clause is an extension where the intent is to have what the standard has; that is, a permanent record of who defined the view. This is why the default `DEFINER` value is the account of the view creator.

The optional `ALGORITHM` clause is a MySQL extension to standard SQL. It affects how MySQL processes the view. `ALGORITHM` takes three values: `MERGE`, `TEMPTABLE`, or `UNDEFINED`. For more information, see Section 25.5.2, "View Processing Algorithms", as well as Section 8.2.2.4, "Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization".

Some views are updatable. That is, you can use them in statements such as `UPDATE`, `DELETE`, or `INSERT` to update the contents of the underlying table. For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view nonupdatable.

A generated column in a view is considered updatable because it is possible to assign to it. However, if such a column is updated explicitly, the only permitted value is `DEFAULT`. For information about generated columns, see Section 13.1.20.8, "CREATE TABLE and Generated Columns".

The `WITH CHECK OPTION` clause can be given for an updatable view to prevent inserts or updates to rows except those for which the `WHERE` clause in the *`select_statement`* is true.

In a `WITH CHECK OPTION` clause for an updatable view, the `LOCAL` and `CASCADED` keywords determine the scope of check testing when the view is defined in terms of another view. The `LOCAL` keyword restricts the `CHECK OPTION` only to the view being defined. `CASCADED` causes the checks for underlying views to be evaluated as well. When neither keyword is given, the default is `CASCADED`.

For more information about updatable views and the `WITH CHECK OPTION` clause, see Section 25.5.3, "Updatable and Insertable Views", and Section 25.5.4, "The View WITH CHECK OPTION Clause".

13.1.24 DROP DATABASE Statement

```
DROP {DATABASE | SCHEMA} [IF EXISTS] db_name
```

`DROP DATABASE` drops all tables in the database and deletes the database. Be *very* careful with this statement! To use `DROP DATABASE`, you need the `DROP` privilege on the database. `DROP SCHEMA` is a synonym for `DROP DATABASE`.

Important

When a database is dropped, privileges granted specifically for the database are *not* automatically dropped. They must be dropped manually. See Section 13.7.1.6, "GRANT Statement".

`IF EXISTS` is used to prevent an error from occurring if the database does not exist.

If the default database is dropped, the default database is unset (the `DATABASE()` function returns `NULL`).

If you use `DROP DATABASE` on a symbolically linked database, both the link and the original database are deleted.

`DROP DATABASE` returns the number of tables that were removed.

The `DROP DATABASE` statement removes from the given database directory those files and directories that MySQL itself may create during normal operation. This includes all files with the extensions shown in the following list:

- `.BAK`
- `.DAT`
- `.HSH`
- `.MRG`
- `.MYD`
- `.MYI`
- `.cfg`
- `.db`
- `.ibd`
- `.ndb`

If other files or directories remain in the database directory after MySQL removes those just listed, the database directory cannot be removed. In this case, you must remove any remaining files or directories manually and issue the `DROP DATABASE` statement again.

Dropping a database does not remove any `TEMPORARY` tables that were created in that database. `TEMPORARY` tables are automatically removed when the session that created them ends. See Section 13.1.20.2, “CREATE TEMPORARY TABLE Statement”.

You can also drop databases with `mysqladmin`. See Section 4.5.2, “mysqladmin — A MySQL Server Administration Program”.

13.1.25 DROP EVENT Statement

```
DROP EVENT [IF EXISTS] event_name
```

This statement drops the event named *`event_name`*. The event immediately ceases being active, and is deleted completely from the server.

If the event does not exist, the error `ERROR 1517 (HY000): Unknown event 'event_name'` results. You can override this and cause the statement to generate a warning for nonexistent events instead using `IF EXISTS`.

This statement requires the `EVENT` privilege for the schema to which the event to be dropped belongs.

13.1.26 DROP FUNCTION Statement

The `DROP FUNCTION` statement is used to drop stored functions and loadable functions:

- For information about dropping stored functions, see Section 13.1.29, “DROP PROCEDURE and DROP FUNCTION Statements”.

- For information about dropping loadable functions, see Section 13.7.4.2, “DROP FUNCTION Statement for Loadable Functions”.

13.1.27 DROP INDEX Statement

```
DROP INDEX index_name ON tbl_name
    [algorithm_option | lock_option] ...

algorithm_option:
    ALGORITHM [=] {DEFAULT | INPLACE | COPY}

lock_option:
    LOCK [=] {DEFAULT | NONE | SHARED | EXCLUSIVE}
```

`DROP INDEX` drops the index named *`index_name`* from the table *`tbl_name`*. This statement is mapped to an `ALTER TABLE` statement to drop the index. See Section 13.1.9, “ALTER TABLE Statement”.

To drop a primary key, the index name is always `PRIMARY`, which must be specified as a quoted identifier because `PRIMARY` is a reserved word:

```
DROP INDEX `PRIMARY` ON t;
```

Indexes on variable-width columns of `NDB` tables are dropped online; that is, without any table copying. The table is not locked against access from other NDB Cluster API nodes, although it is locked against other operations on the *same* API node for the duration of the operation. This is done automatically by the server whenever it determines that it is possible to do so; you do not have to use any special SQL syntax or server options to cause it to happen.

`ALGORITHM` and `LOCK` clauses may be given to influence the table copying method and level of concurrency for reading and writing the table while its indexes are being modified. They have the same meaning as for the `ALTER TABLE` statement. For more information, see Section 13.1.9, “ALTER TABLE Statement”

MySQL NDB Cluster supports online operations using the same `ALGORITHM=INPLACE` syntax supported in the standard MySQL Server. See Section 23.6.12, “Online Operations with ALTER TABLE in NDB Cluster”, for more information.

13.1.28 DROP LOGFILE GROUP Statement

```
DROP LOGFILE GROUP logfile_group
    ENGINE [=] engine_name
```

This statement drops the log file group named *`logfile_group`*. The log file group must already exist or an error results. (For information on creating log file groups, see Section 13.1.16, “CREATE LOGFILE GROUP Statement”.)

Important

Before dropping a log file group, you must drop all tablespaces that use that log file group for `UNDO` logging.

The required `ENGINE` clause provides the name of the storage engine used by the log file group to be dropped. Currently, the only permitted values for *`engine_name`* are `NDB` and `NDBCLUSTER`.

`DROP LOGFILE GROUP` is useful only with Disk Data storage for NDB Cluster. See Section 23.6.11, “NDB Cluster Disk Data Tables”.

13.1.29 DROP PROCEDURE and DROP FUNCTION Statements

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

These statements are used to drop a stored routine (a stored procedure or function). That is, the specified routine is removed from the server. (`DROP FUNCTION` is also used to drop loadable functions; see Section 13.7.4.2, “DROP FUNCTION Statement for Loadable Functions”.)

To drop a stored routine, you must have the `ALTER ROUTINE` privilege for it. (If the `automatic_sp_privileges` system variable is enabled, that privilege and `EXECUTE` are granted automatically to the routine creator when the routine is created and dropped from the creator when the routine is dropped. See Section 25.2.2, “Stored Routines and MySQL Privileges”.)

In addition, if the definer of the routine has the `SYSTEM_USER` privilege, the user dropping it must also have this privilege. This is enforced in MySQL 8.0.16 and later.

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

`DROP FUNCTION` is also used to drop loadable functions (see Section 13.7.4.2, “DROP FUNCTION Statement for Loadable Functions”).

13.1.30 DROP SERVER Statement

```
DROP SERVER [ IF EXISTS ] server_name
```

Drops the server definition for the server named *`server_name`*. The corresponding row in the `mysql.servers` table is deleted. This statement requires the `SUPER` privilege.

Dropping a server for a table does not affect any `FEDERATED` tables that used this connection information when they were created. See Section 13.1.18, “CREATE SERVER Statement”.

`DROP SERVER` causes an implicit commit. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

`DROP SERVER` is not written to the binary log, regardless of the logging format that is in use.

13.1.31 DROP SPATIAL REFERENCE SYSTEM Statement

```
DROP SPATIAL REFERENCE SYSTEM
    [IF EXISTS]
    srid

srid: 32-bit unsigned integer
```

This statement removes a spatial reference system (SRS) definition from the data dictionary. It requires the `SUPER` privilege.

Example:

```
DROP SPATIAL REFERENCE SYSTEM 4120;
```

If no SRS definition with the SRID value exists, an error occurs unless `IF EXISTS` is specified. In that case, a warning occurs rather than an error.

If the SRID value is used by some column in an existing table, an error occurs. For example:

```
mysql> DROP SPATIAL REFERENCE SYSTEM 4326;
ERROR 3716 (SR005): Can't modify SRID 4326. There is at
least one column depending on it.
```

To identify which column or columns use the SRID, use this query:

```
SELECT * FROM INFORMATION_SCHEMA.ST_GEOMETRY_COLUMNS WHERE SRS_ID=4326;
```

SRID values must be in the range of 32-bit unsigned integers, with these restrictions:

- SRID 0 is a valid SRID but cannot be used with `DROP SPATIAL REFERENCE SYSTEM`.
- If the value is in a reserved SRID range, a warning occurs. Reserved ranges are [0, 32767] (reserved by EPSG), [60,000,000, 69,999,999] (reserved by EPSG), and [2,000,000,000, 2,147,483,647] (reserved by MySQL). EPSG stands for the European Petroleum Survey Group.
- Users should not drop SRSs with SRIDs in the reserved ranges. If system-installed SRSs are dropped, the SRS definitions may be recreated for MySQL upgrades.

13.1.32 DROP TABLE Statement

```
DROP [TEMPORARY] TABLE [IF EXISTS]
    tbl_name [, tbl_name] ...
    [RESTRICT | CASCADE]
```

`DROP TABLE` removes one or more tables. You must have the `DROP` privilege for each table.

Be careful with this statement! For each table, it removes the table definition and all table data. If the table is partitioned, the statement removes the table definition, all its partitions, all data stored in those partitions, and all partition definitions associated with the dropped table.

Dropping a table also drops any triggers for the table.

`DROP TABLE` causes an implicit commit, except when used with the `TEMPORARY` keyword. See Section 13.3.3, “Statements That Cause an Implicit Commit”.

Important

When a table is dropped, privileges granted specifically for the table are *not* automatically dropped. They must be dropped manually. See Section 13.7.1.6, “GRANT Statement”.

If any tables named in the argument list do not exist, `DROP TABLE` behavior depends on whether the `IF EXISTS` clause is given:

- Without `IF EXISTS`, the statement fails with an error indicating which nonexisting tables it was unable to drop, and no changes are made.
- With `IF EXISTS`, no error occurs for nonexisting tables. The statement drops all named tables that do exist, and generates a `NOTE` diagnostic for each nonexistent table. These notes can be displayed with `SHOW WARNINGS`. See Section 13.7.7.42, “SHOW WARNINGS Statement”.

`IF EXISTS` can also be useful for dropping tables in unusual circumstances under which there is an entry in the data dictionary but no table managed by the storage engine. (For example, if an abnormal server exit occurs after removal of the table from the storage engine but before removal of the data dictionary entry.)

The `TEMPORARY` keyword has the following effects:

- The statement drops only `TEMPORARY` tables.
- The statement does not cause an implicit commit.
- No access rights are checked. A `TEMPORARY` table is visible only with the session that created it, so no check is necessary.

Including the `TEMPORARY` keyword is a good way to prevent accidentally dropping non-`TEMPORARY` tables.

The `RESTRICT` and `CASCADE` keywords do nothing. They are permitted to make porting easier from other database systems.

`DROP TABLE` is not supported with all `innodb_force_recovery` settings. See Section 15.21.3, “Forcing InnoDB Recovery”.

13.1.33 DROP TABLESPACE Statement

```
DROP [UNDO] TABLESPACE tablespace_name
    [ENGINE [=] engine_name]
```

This statement drops a tablespace that was previously created using `CREATE TABLESPACE`. It is supported by the `NDB` and `InnoDB` storage engines.

The `UNDO` keyword, introduced in MySQL 8.0.14, must be specified to drop an undo tablespace. Only undo tablespaces created using `CREATE UNDO TABLESPACE` syntax can be dropped. An undo tablespace must be in an `empty` state before it can be dropped. For more information, see Section 15.6.3.4, “Undo Tablespaces”.

`ENGINE` sets the storage engine that uses the tablespace, where *`engine_name`* is the name of the storage engine. Currently, the values `InnoDB` and `NDB` are supported. If not set, the value of `default_storage_engine` is used. If it is not the same as the storage engine used to create the tablespace, the `DROP TABLESPACE` statement fails.

`tablespace_name` is a case-sensitive identifier in MySQL.

For an `InnoDB` general tablespace, all tables must be dropped from the tablespace prior to a `DROP TABLESPACE` operation. If the tablespace is not empty, `DROP TABLESPACE` returns an error.

An `NDB` tablespace to be dropped must not contain any data files; in other words, before you can drop an `NDB` tablespace, you must first drop each of its data files using `ALTER TABLESPACE ... DROP DATAFILE`.

Notes

- A general `InnoDB` tablespace is not deleted automatically when the last table in the tablespace is dropped. The tablespace must be dropped explicitly using `DROP TABLESPACE` *`tablespace_name`*.
- A `DROP DATABASE` operation can drop tables that belong to a general tablespace but it cannot drop the tablespace, even if the operation drops all tables that belong to the tablespace. The tablespace must be dropped explicitly using `DROP TABLESPACE` *`tablespace_name`*.
- Similar to the system tablespace, truncating or dropping tables stored in a general tablespace creates free space internally in the general tablespace .ibd data file which can only be used for new `InnoDB` data. Space is not released back to the operating system as it is for file-per-table tablespaces.

InnoDB Examples

This example demonstrates how to drop an `InnoDB` general tablespace. The general tablespace `ts1` is created with a single table. Before dropping the tablespace, the table must be dropped.

```
mysql> CREATE TABLESPACE `ts1` ADD DATAFILE 'ts1.ibd' Engine=InnoDB;

mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY) TABLESPACE ts1 Engine=InnoDB;

mysql> DROP TABLE t1;

mysql> DROP TABLESPACE ts1;
```

This example demonstrates dropping an undo tablespace. An undo tablespace must be in an `empty` state before it can be dropped. For more information, see Section 15.6.3.4, “Undo Tablespaces”.

```
mysql> DROP UNDO TABLESPACE undo_003;
```

NDB Example

This example shows how to drop an NDB tablespace myts having a data file named mydata-1.dat after first creating the tablespace, and assumes the existence of a log file group named mylg (see Section 13.1.16, “CREATE LOGFILE GROUP Statement”).

```
mysql> CREATE TABLESPACE myts
    ->     ADD DATAFILE 'mydata-1.dat'
    ->     USE LOGFILE GROUP mylg
    ->     ENGINE=NDB;
```

You must remove all data files from the tablespace using ALTER TABLESPACE, as shown here, before it can be dropped:

```
mysql> ALTER TABLESPACE myts
    ->     DROP DATAFILE 'mydata-1.dat'
    ->     ENGINE=NDB;

mysql> DROP TABLESPACE myts;
```

13.1.34 DROP TRIGGER Statement

```
DROP TRIGGER [IF EXISTS] [schema_name.]trigger_name
```

This statement drops a trigger. The schema (database) name is optional. If the schema is omitted, the trigger is dropped from the default schema. DROP TRIGGER requires the TRIGGER privilege for the table associated with the trigger.

Use IF EXISTS to prevent an error from occurring for a trigger that does not exist. A NOTE is generated for a nonexistent trigger when using IF EXISTS. See Section 13.7.7.42, “SHOW WARNINGS Statement”.

Triggers for a table are also dropped if you drop the table.

13.1.35 DROP VIEW Statement

```
DROP VIEW [IF EXISTS]
    view_name [, view_name] ...
    [RESTRICT | CASCADE]
```

DROP VIEW removes one or more views. You must have the DROP privilege for each view.

If any views named in the argument list do not exist, the statement fails with an error indicating by name which nonexisting views it was unable to drop, and no changes are made.

Note

In MySQL 5.7 and earlier, DROP VIEW returns an error if any views named in the argument list do not exist, but also drops all views in the list that do exist. Due to the change in behavior in MySQL 8.0, a partially completed DROP VIEW operation on a MySQL 5.7 replication source server fails when replicated on a MySQL 8.0 replica. To avoid this failure scenario, use IF EXISTS syntax in DROP VIEW statements to prevent an error from occurring for views that do not exist. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

The IF EXISTS clause prevents an error from occurring for views that don't exist. When this clause is given, a NOTE is generated for each nonexistent view. See Section 13.7.7.42, “SHOW WARNINGS Statement”.

RESTRICT and CASCADE, if given, are parsed and ignored.

13.1.36 RENAME TABLE Statement

```
RENAME TABLE
    tbl_name TO new_tbl_name
    [, tbl_name2 TO new_tbl_name2] ...
```

RENAME TABLE renames one or more tables. You must have ALTER and DROP privileges for the original table, and CREATE and INSERT privileges for the new table.

For example, to rename a table named old_table to new_table, use this statement:

```
RENAME TABLE old_table TO new_table;
```

That statement is equivalent to the following ALTER TABLE statement:

```
ALTER TABLE old_table RENAME new_table;
```

RENAME TABLE, unlike ALTER TABLE, can rename multiple tables within a single statement:

```
RENAME TABLE old_table1 TO new_table1,
             old_table2 TO new_table2,
             old_table3 TO new_table3;
```

Renaming operations are performed left to right. Thus, to swap two table names, do this (assuming that a table with the intermediary name tmp_table does not already exist):

```
RENAME TABLE old_table TO tmp_table,
             new_table TO old_table,
             tmp_table TO new_table;
```

Metadata locks on tables are acquired in name order, which in some cases can make a difference in operation outcome when multiple transactions execute concurrently. See Section 8.11.4, “Metadata Locking”.

As of MySQL 8.0.13, you can rename tables locked with a LOCK TABLES statement, provided that they are locked with a WRITE lock or are the product of renaming WRITE-locked tables from earlier steps in a multiple-table rename operation. For example, this is permitted:

```
LOCK TABLE old_table1 WRITE;
RENAME TABLE old_table1 TO new_table1,
             new_table1 TO new_table2;
```

This is not permitted:

```
LOCK TABLE old_table1 READ;
RENAME TABLE old_table1 TO new_table1,
             new_table1 TO new_table2;
```

Prior to MySQL 8.0.13, to execute RENAME TABLE, there must be no tables locked with LOCK TABLES.

With the transaction table locking conditions satisfied, the rename operation is done atomically; no other session can access any of the tables while the rename is in progress.

If any errors occur during a RENAME TABLE, the statement fails and no changes are made.

You can use RENAME TABLE to move a table from one database to another:

```
RENAME TABLE current_db.tbl_name TO other_db.tbl_name;
```

Using this method to move all tables from one database to a different one in effect renames the database (an operation for which MySQL has no single statement), except that the original database continues to exist, albeit with no tables.

Like RENAME TABLE, ALTER TABLE ... RENAME can also be used to move a table to a different database. Regardless of the statement used, if the rename operation would move the table to a

database located on a different file system, the success of the outcome is platform specific and depends on the underlying operating system calls used to move table files.

If a table has triggers, attempts to rename the table into a different database fail with a `Trigger in wrong schema` (`ER_TRG_IN_WRONG_SCHEMA`) error.

An unencrypted table can be moved to an encryption-enabled database and vice versa. However, if the `table_encryption_privilege_check` variable is enabled, the `TABLE_ENCRYPTION_ADMIN` privilege is required if the table encryption setting differs from the default database encryption.

To rename `TEMPORARY` tables, `RENAME TABLE` does not work. Use `ALTER TABLE` instead.

`RENAME TABLE` works for views, except that views cannot be renamed into a different database.

Any privileges granted specifically for a renamed table or view are not migrated to the new name. They must be changed manually.

`RENAME TABLE` *`tbl_name`* `TO` *`new_tbl_name`* changes internally generated foreign key constraint names and user-defined foreign key constraint names that begin with the string "*`tbl_name`*_ibfk_" to reflect the new table name. `InnoDB` interprets foreign key constraint names that begin with the string "*`tbl_name`*_ibfk_" as internally generated names.

Foreign key constraint names that point to the renamed table are automatically updated unless there is a conflict, in which case the statement fails with an error. A conflict occurs if the renamed constraint name already exists. In such cases, you must drop and re-create the foreign keys for them to function properly.

`RENAME TABLE` *`tbl_name`* `TO` *`new_tbl_name`* changes internally generated and user-defined `CHECK` constraint names that begin with the string "*`tbl_name`*_chk_" to reflect the new table name. MySQL interprets `CHECK` constraint names that begin with the string "*`tbl_name`*_chk_" as internally generated names. Example:

```
mysql> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `i1` int(11) DEFAULT NULL,
  `i2` int(11) DEFAULT NULL,
  CONSTRAINT `t1_chk_1` CHECK ((`i1` > 0)),
  CONSTRAINT `t1_chk_2` CHECK ((`i2` < 0))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.02 sec)

mysql> RENAME TABLE t1 TO t3;
Query OK, 0 rows affected (0.03 sec)

mysql> SHOW CREATE TABLE t3\G
*************************** 1. row ***************************
       Table: t3
Create Table: CREATE TABLE `t3` (
  `i1` int(11) DEFAULT NULL,
  `i2` int(11) DEFAULT NULL,
  CONSTRAINT `t3_chk_1` CHECK ((`i1` > 0)),
  CONSTRAINT `t3_chk_2` CHECK ((`i2` < 0))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.01 sec)
```

13.1.37 TRUNCATE TABLE Statement

```
TRUNCATE [TABLE] tbl_name
```

`TRUNCATE TABLE` empties a table completely. It requires the `DROP` privilege. Logically, `TRUNCATE TABLE` is similar to a `DELETE` statement that deletes all rows, or a sequence of `DROP TABLE` and `CREATE TABLE` statements.

To achieve high performance, `TRUNCATE TABLE` bypasses the DML method of deleting data. Thus, it does not cause `ON DELETE` triggers to fire, it cannot be performed for `InnoDB` tables with parent-child foreign key relationships, and it cannot be rolled back like a DML operation. However, `TRUNCATE TABLE` operations on tables that use an atomic DDL-supported storage engine are either fully committed or rolled back if the server halts during their operation. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

Although `TRUNCATE TABLE` is similar to `DELETE`, it is classified as a DDL statement rather than a DML statement. It differs from `DELETE` in the following ways:

- Truncate operations drop and re-create the table, which is much faster than deleting rows one by one, particularly for large tables.
- Truncate operations cause an implicit commit, and so cannot be rolled back. See Section 13.3.3, “Statements That Cause an Implicit Commit”.
- Truncation operations cannot be performed if the session holds an active table lock.
- `TRUNCATE TABLE` fails for an `InnoDB` table or `NDB` table if there are any `FOREIGN KEY` constraints from other tables that reference the table. Foreign key constraints between columns of the same table are permitted.
- Truncation operations do not return a meaningful value for the number of deleted rows. The usual result is “0 rows affected,” which should be interpreted as “no information.”
- As long as the table definition is valid, the table can be re-created as an empty table with `TRUNCATE TABLE`, even if the data or index files have become corrupted.
- Any `AUTO_INCREMENT` value is reset to its start value. This is true even for `MyISAM` and `InnoDB`, which normally do not reuse sequence values.
- When used with partitioned tables, `TRUNCATE TABLE` preserves the partitioning; that is, the data and index files are dropped and re-created, while the partition definitions are unaffected.
- The `TRUNCATE TABLE` statement does not invoke `ON DELETE` triggers.
- Truncating a corrupted `InnoDB` table is supported.

`TRUNCATE TABLE` is treated for purposes of binary logging and replication as DDL rather than DML, and is always logged as a statement.

`TRUNCATE TABLE` for a table closes all handlers for the table that were opened with `HANDLER OPEN`.

In MySQL 5.7 and earlier, on a system with a large buffer pool and `innodb_adaptive_hash_index` enabled, a `TRUNCATE TABLE` operation could cause a temporary drop in system performance due to an LRU scan that occurred when removing the table's adaptive hash index entries (Bug #68184). The remapping of `TRUNCATE TABLE` to `DROP TABLE` and `CREATE TABLE` in MySQL 8.0 avoids the problematic LRU scan.

`TRUNCATE TABLE` can be used with Performance Schema summary tables, but the effect is to reset the summary columns to 0 or `NULL`, not to remove rows. See Section 27.12.20, “Performance Schema Summary Tables”.

Truncating an `InnoDB` table that resides in a file-per-table tablespace drops the existing tablespace and creates a new one. As of MySQL 8.0.21, if the tablespace was created with an earlier version and resides in an unknown directory, `InnoDB` creates the new tablespace in the default location and writes the following warning to the error log: `The DATA DIRECTORY location must be in a known directory. The DATA DIRECTORY location will be ignored and the file will be put into the default datadir location`. Known directories are those defined by the `datadir`, `innodb_data_home_dir`, and `innodb_directories` variables. To have `TRUNCATE TABLE` create the tablespace in its current location, add the directory to the `innodb_directories` setting before running `TRUNCATE TABLE`.

13.2 Data Manipulation Statements

13.2.1 CALL Statement

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

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

Stored procedures that take no arguments can be invoked without parentheses. That is, `CALL p()` and `CALL p` are equivalent.

`CALL` can pass back values to its caller using parameters that are declared as `OUT` or `INOUT` parameters. When the procedure returns, a client program can also obtain the number of rows affected for the final statement executed within the routine: At the SQL level, call the `ROW_COUNT()` function; from the C API, call the `mysql_affected_rows()` function.

For information about the effect of unhandled conditions on procedure parameters, see Section 13.6.7.8, “Condition Handling and OUT or INOUT Parameters”.

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

```
DELIMITER //

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

DELIMITER ;
```

Before calling the procedure, initialize the variable to be passed as the `INOUT` parameter. After calling the procedure, you can see that the values of the two variables are set or modified:

```
mysql> SET @increment = 10;
mysql> CALL p(@version, @increment);
mysql> SELECT @version, @increment;
+----------+------------+
| @version | @increment |
+----------+------------+
| 8.0.32   |         11 |
+----------+------------+
```

In prepared `CALL` statements used with `PREPARE` and `EXECUTE`, placeholders can be used for `IN` parameters, `OUT`, and `INOUT` parameters. These types of parameters can be used as follows:

```
mysql> SET @increment = 10;
mysql> PREPARE s FROM 'CALL p(?, ?)';
mysql> EXECUTE s USING @version, @increment;
mysql> SELECT @version, @increment;
+----------+------------+
| @version | @increment |
+----------+------------+
| 8.0.32   |         11 |
+----------+------------+
```

To write C programs that use the `CALL` SQL statement to execute stored procedures that produce result sets, the `CLIENT_MULTI_RESULTS` flag must be enabled. This is because each `CALL` returns a result to indicate the call status, in addition to any result sets that might be returned by statements executed within the procedure. `CLIENT_MULTI_RESULTS` must also be enabled if `CALL` is used to execute any stored procedure that contains prepared statements. It cannot be determined when such a procedure is loaded whether those statements produce result sets, so it is necessary to assume that they do so.

`CLIENT_MULTI_RESULTS` can be enabled when you call `mysql_real_connect()`, either explicitly by passing the `CLIENT_MULTI_RESULTS` flag itself, or implicitly by passing `CLIENT_MULTI_STATEMENTS` (which also enables `CLIENT_MULTI_RESULTS`). `CLIENT_MULTI_RESULTS` is enabled by default.

To process the result of a `CALL` statement executed using `mysql_query()` or `mysql_real_query()`, use a loop that calls `mysql_next_result()` to determine whether there are more results. For an example, see Multiple Statement Execution Support.

C programs can use the prepared-statement interface to execute `CALL` statements and access `OUT` and `INOUT` parameters. This is done by processing the result of a `CALL` statement using a loop that calls `mysql_stmt_next_result()` to determine whether there are more results. For an example, see Prepared CALL Statement Support. Languages that provide a MySQL interface can use prepared `CALL` statements to directly retrieve `OUT` and `INOUT` procedure parameters.

Metadata changes to objects referred to by stored programs are detected and cause automatic reparsing of the affected statements when the program is next executed. For more information, see Section 8.10.3, “Caching of Prepared Statements and Stored Programs”.

13.2.2 DELETE Statement

`DELETE` is a DML statement that removes rows from a table.

A `DELETE` statement can start with a `WITH` clause to define common table expressions accessible within the `DELETE`. See Section 13.2.20, “WITH (Common Table Expressions)”.

Single-Table Syntax

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name [[AS] tbl_alias]
    [PARTITION (partition_name [, partition_name] ...)]
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]
```

The `DELETE` statement deletes rows from *`tbl_name`* and returns the number of deleted rows. To check the number of deleted rows, call the `ROW_COUNT()` function described in Section 12.16, “Information Functions”.

Main Clauses

The conditions in the optional `WHERE` clause identify which rows to delete. With no `WHERE` clause, all rows are deleted.

`where_condition` is an expression that evaluates to true for each row to be deleted. It is specified as described in Section 13.2.13, “SELECT Statement”.

If the `ORDER BY` clause is specified, the rows are deleted in the order that is specified. The `LIMIT` clause places a limit on the number of rows that can be deleted. These clauses apply to single-table deletes, but not multi-table deletes.

Multiple-Table Syntax

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
```

```
    tbl_name[.*] [, tbl_name[.*]] ...
    FROM table_references
    [WHERE where_condition]

DELETE [LOW_PRIORITY] [QUICK] [IGNORE]
    FROM tbl_name[.*] [, tbl_name[.*]] ...
    USING table_references
    [WHERE where_condition]
```

Privileges

You need the `DELETE` privilege on a table to delete rows from it. You need only the `SELECT` privilege for any columns that are only read, such as those named in the `WHERE` clause.

Performance

When you do not need to know the number of deleted rows, the `TRUNCATE TABLE` statement is a faster way to empty a table than a `DELETE` statement with no `WHERE` clause. Unlike `DELETE`, `TRUNCATE TABLE` cannot be used within a transaction or if you have a lock on the table. See Section 13.1.37, “TRUNCATE TABLE Statement” and Section 13.3.6, “LOCK TABLES and UNLOCK TABLES Statements”.

The speed of delete operations may also be affected by factors discussed in Section 8.2.5.3, “Optimizing DELETE Statements”.

To ensure that a given `DELETE` statement does not take too much time, the MySQL-specific `LIMIT row_count` clause for `DELETE` specifies the maximum number of rows to be deleted. If the number of rows to delete is larger than the limit, repeat the `DELETE` statement until the number of affected rows is less than the `LIMIT` value.

Subqueries

You cannot delete from a table and select from the same table in a subquery.

Partitioned Table Support

`DELETE` supports explicit partition selection using the `PARTITION` clause, which takes a list of the comma-separated names of one or more partitions or subpartitions (or both) from which to select rows to be dropped. Partitions not included in the list are ignored. Given a partitioned table `t` with a partition named `p0`, executing the statement `DELETE FROM t PARTITION (p0)` has the same effect on the table as executing `ALTER TABLE t TRUNCATE PARTITION (p0)`; in both cases, all rows in partition `p0` are dropped.

`PARTITION` can be used along with a `WHERE` condition, in which case the condition is tested only on rows in the listed partitions. For example, `DELETE FROM t PARTITION (p0) WHERE c < 5` deletes rows only from partition `p0` for which the condition `c < 5` is true; rows in any other partitions are not checked and thus not affected by the `DELETE`.

The `PARTITION` clause can also be used in multiple-table `DELETE` statements. You can use up to one such option per table named in the `FROM` option.

For more information and examples, see Section 24.5, “Partition Selection”.

Auto-Increment Columns

If you delete the row containing the maximum value for an `AUTO_INCREMENT` column, the value is not reused for a `MyISAM` or `InnoDB` table. If you delete all rows in the table with `DELETE FROM tbl_name` (without a `WHERE` clause) in `autocommit` mode, the sequence starts over for all storage engines except `InnoDB` and `MyISAM`. There are some exceptions to this behavior for `InnoDB` tables, as discussed in Section 15.6.1.6, “AUTO_INCREMENT Handling in InnoDB”.

For MyISAM tables, you can specify an AUTO_INCREMENT secondary column in a multiple-column key. In this case, reuse of values deleted from the top of the sequence occurs even for MyISAM tables. See Section 3.6.9, “Using AUTO_INCREMENT”.

Modifiers

The DELETE statement supports the following modifiers:

- If you specify the LOW_PRIORITY modifier, the server delays execution of the DELETE until no other clients are reading from the table. This affects only storage engines that use only table-level locking (such as MyISAM, MEMORY, and MERGE).
- For MyISAM tables, if you use the QUICK modifier, the storage engine does not merge index leaves during delete, which may speed up some kinds of delete operations.
- The IGNORE modifier causes MySQL to ignore ignorable errors during the process of deleting rows. (Errors encountered during the parsing stage are processed in the usual manner.) Errors that are ignored due to the use of IGNORE are returned as warnings. For more information, see The Effect of IGNORE on Statement Execution.

Order of Deletion

If the DELETE statement includes an ORDER BY clause, rows are deleted in the order specified by the clause. This is useful primarily in conjunction with LIMIT. For example, the following statement finds rows matching the WHERE clause, sorts them by timestamp_column, and deletes the first (oldest) one:

```
DELETE FROM somelog WHERE user = 'jcole'
ORDER BY timestamp_column LIMIT 1;
```

ORDER BY also helps to delete rows in an order required to avoid referential integrity violations.

InnoDB Tables

If you are deleting many rows from a large table, you may exceed the lock table size for an InnoDB table. To avoid this problem, or simply to minimize the time that the table remains locked, the following strategy (which does not use DELETE at all) might be helpful:

1. Select the rows *not* to be deleted into an empty table that has the same structure as the original table:

   ```
   INSERT INTO t_copy SELECT * FROM t WHERE ... ;
   ```

2. Use RENAME TABLE to atomically move the original table out of the way and rename the copy to the original name:

   ```
   RENAME TABLE t TO t_old, t_copy TO t;
   ```

3. Drop the original table:

   ```
   DROP TABLE t_old;
   ```

No other sessions can access the tables involved while RENAME TABLE executes, so the rename operation is not subject to concurrency problems. See Section 13.1.36, “RENAME TABLE Statement”.

MyISAM Tables

In MyISAM tables, deleted rows are maintained in a linked list and subsequent INSERT operations reuse old row positions. To reclaim unused space and reduce file sizes, use the OPTIMIZE TABLE statement or the myisamchk utility to reorganize tables. OPTIMIZE TABLE is easier to use, but myisamchk is faster. See Section 13.7.3.4, “OPTIMIZE TABLE Statement”, and Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”.

The QUICK modifier affects whether index leaves are merged for delete operations. DELETE QUICK is most useful for applications where index values for deleted rows are replaced by similar index values from rows inserted later. In this case, the holes left by deleted values are reused.

DELETE QUICK is not useful when deleted values lead to underfilled index blocks spanning a range of index values for which new inserts occur again. In this case, use of QUICK can lead to wasted space in the index that remains unreclaimed. Here is an example of such a scenario:

1. Create a table that contains an indexed AUTO_INCREMENT column.
2. Insert many rows into the table. Each insert results in an index value that is added to the high end of the index.
3. Delete a block of rows at the low end of the column range using DELETE QUICK.

In this scenario, the index blocks associated with the deleted index values become underfilled but are not merged with other index blocks due to the use of QUICK. They remain underfilled when new inserts occur, because new rows do not have index values in the deleted range. Furthermore, they remain underfilled even if you later use DELETE without QUICK, unless some of the deleted index values happen to lie in index blocks within or adjacent to the underfilled blocks. To reclaim unused index space under these circumstances, use OPTIMIZE TABLE.

If you are going to delete many rows from a table, it might be faster to use DELETE QUICK followed by OPTIMIZE TABLE. This rebuilds the index rather than performing many index block merge operations.

Multi-Table Deletes

You can specify multiple tables in a DELETE statement to delete rows from one or more tables depending on the condition in the WHERE clause. You cannot use ORDER BY or LIMIT in a multiple-table DELETE. The *table_references* clause lists the tables involved in the join, as described in Section 13.2.13.2, “JOIN Clause”.

For the first multiple-table syntax, only matching rows from the tables listed before the FROM clause are deleted. For the second multiple-table syntax, only matching rows from the tables listed in the FROM clause (before the USING clause) are deleted. The effect is that you can delete rows from many tables at the same time and have additional tables that are used only for searching:

```
DELETE t1, t2 FROM t1 INNER JOIN t2 INNER JOIN t3
WHERE t1.id=t2.id AND t2.id=t3.id;
```

Or:

```
DELETE FROM t1, t2 USING t1 INNER JOIN t2 INNER JOIN t3
WHERE t1.id=t2.id AND t2.id=t3.id;
```

These statements use all three tables when searching for rows to delete, but delete matching rows only from tables t1 and t2.

The preceding examples use INNER JOIN, but multiple-table DELETE statements can use other types of join permitted in SELECT statements, such as LEFT JOIN. For example, to delete rows that exist in t1 that have no match in t2, use a LEFT JOIN:

```
DELETE t1 FROM t1 LEFT JOIN t2 ON t1.id=t2.id WHERE t2.id IS NULL;
```

The syntax permits .* after each *tbl_name* for compatibility with Access.

If you use a multiple-table DELETE statement involving InnoDB tables for which there are foreign key constraints, the MySQL optimizer might process tables in an order that differs from that of their parent/child relationship. In this case, the statement fails and rolls back. Instead, you should delete from a single table and rely on the ON DELETE capabilities that InnoDB provides to cause the other tables to be modified accordingly.

Note

If you declare an alias for a table, you must use the alias when referring to the table:

```
DELETE t1 FROM test AS t1, test2 WHERE ...
```

Table aliases in a multiple-table `DELETE` should be declared only in the *`table_references`* part of the statement. Elsewhere, alias references are permitted but not alias declarations.

Correct:

```
DELETE a1, a2 FROM t1 AS a1 INNER JOIN t2 AS a2
WHERE a1.id=a2.id;

DELETE FROM a1, a2 USING t1 AS a1 INNER JOIN t2 AS a2
WHERE a1.id=a2.id;
```

Incorrect:

```
DELETE t1 AS a1, t2 AS a2 FROM t1 INNER JOIN t2
WHERE a1.id=a2.id;

DELETE FROM t1 AS a1, t2 AS a2 USING t1 INNER JOIN t2
WHERE a1.id=a2.id;
```

Table aliases are also supported for single-table `DELETE` statements beginning with MySQL 8.0.16. (Bug #89410,Bug #27455809)

13.2.3 DO Statement

```
DO expr [, expr] ...
```

`DO` executes the expressions but does not return any results. In most respects, `DO` is shorthand for `SELECT expr, ...`, but has the advantage that it is slightly faster when you do not care about the result.

`DO` is useful primarily with functions that have side effects, such as `RELEASE_LOCK()`.

Example: This `SELECT` statement pauses, but also produces a result set:

```
mysql> SELECT SLEEP(5);
+----------+
| SLEEP(5) |
+----------+
|        0 |
+----------+
1 row in set (5.02 sec)
```

`DO`, on the other hand, pauses without producing a result set.:

```
mysql> DO SLEEP(5);
Query OK, 0 rows affected (4.99 sec)
```

This could be useful, for example in a stored function or trigger, which prohibit statements that produce result sets.

`DO` only executes expressions. It cannot be used in all cases where `SELECT` can be used. For example, `DO id FROM t1` is invalid because it references a table.

13.2.4 EXCEPT Clause

```
query_expression_body EXCEPT [ALL | DISTINCT] query_expression_body
    [EXCEPT [ALL | DISTINCT] query_expression_body]
    [...]
```

```
query_expression_body:
    See Section 13.2.14, “Set Operations with UNION, INTERSECT, and EXCEPT”
```

`EXCEPT` limits the result from the first query block to those rows which are (also) not found in the second. As with `UNION` and `INTERSECT`, either query block can make use of any of `SELECT`, `TABLE`, or `VALUES`. An example using the tables `a`, `b`, and `c` defined in Section 13.2.8, “INTERSECT Clause”, is shown here:

```
mysql> TABLE a EXCEPT TABLE b;
+------+------+
| m    | n    |
+------+------+
|    2 |    3 |
+------+------+
1 row in set (0.00 sec)

mysql> TABLE a EXCEPT TABLE c;
+------+------+
| m    | n    |
+------+------+
|    1 |    2 |
|    2 |    3 |
+------+------+
2 rows in set (0.00 sec)

mysql> TABLE b EXCEPT TABLE c;
+------+------+
| m    | n    |
+------+------+
|    1 |    2 |
+------+------+
1 row in set (0.00 sec)
```

As with `UNION` and `INTERSECT`, if neither `DISTINCT` nor `ALL` is specified, the default is `DISTINCT`.

`DISTINCT` removes duplicates found on either side of the relation, as shown here:

```
mysql> TABLE c EXCEPT DISTINCT TABLE a;
+------+------+
| m    | n    |
+------+------+
|    1 |    3 |
+------+------+
1 row in set (0.00 sec)

mysql> TABLE c EXCEPT ALL TABLE a;
+------+------+
| m    | n    |
+------+------+
|    1 |    3 |
|    1 |    3 |
+------+------+
2 rows in set (0.00 sec)
```

(The first statement has the same effect as `TABLE c EXCEPT TABLE a`.)

Unlike `UNION` or `INTERSECT`, `EXCEPT` is *not* commutative—that is, the result depends on the order of the operands, as shown here:

```
mysql> TABLE a EXCEPT TABLE c;
+------+------+
| m    | n    |
+------+------+
|    1 |    2 |
|    2 |    3 |
+------+------+
2 rows in set (0.00 sec)

mysql> TABLE c EXCEPT TABLE a;
+------+------+
```

```
| m    | n    |
+------+------+
|    1 |    3 |
+------+------+
1 row in set (0.00 sec)
```

As with `UNION`, the result sets to be compared must have the same number of columns. Result set column types are also determined as for `UNION`.

`EXCEPT` was added in MySQL 8.0.31.

13.2.5 HANDLER Statement

```
HANDLER tbl_name OPEN [ [AS] alias]

HANDLER tbl_name READ index_name { = | <= | >= | < | > } (value1,value2,...)
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name READ index_name { FIRST | NEXT | PREV | LAST }
    [ WHERE where_condition ] [LIMIT ... ]
HANDLER tbl_name READ { FIRST | NEXT }
    [ WHERE where_condition ] [LIMIT ... ]

HANDLER tbl_name CLOSE
```

The `HANDLER` statement provides direct access to table storage engine interfaces. It is available for `InnoDB` and `MyISAM` tables.

The `HANDLER ... OPEN` statement opens a table, making it accessible using subsequent `HANDLER ... READ` statements. This table object is not shared by other sessions and is not closed until the session calls `HANDLER ... CLOSE` or the session terminates.

If you open the table using an alias, further references to the open table with other `HANDLER` statements must use the alias rather than the table name. If you do not use an alias, but open the table using a table name qualified by the database name, further references must use the unqualified table name. For example, for a table opened using `mydb.mytable`, further references must use `mytable`.

The first `HANDLER ... READ` syntax fetches a row where the index specified satisfies the given values and the `WHERE` condition is met. If you have a multiple-column index, specify the index column values as a comma-separated list. Either specify values for all the columns in the index, or specify values for a leftmost prefix of the index columns. Suppose that an index `my_idx` includes three columns named `col_a`, `col_b`, and `col_c`, in that order. The `HANDLER` statement can specify values for all three columns in the index, or for the columns in a leftmost prefix. For example:

```
HANDLER ... READ my_idx = (col_a_val,col_b_val,col_c_val) ...
HANDLER ... READ my_idx = (col_a_val,col_b_val) ...
HANDLER ... READ my_idx = (col_a_val) ...
```

To employ the `HANDLER` interface to refer to a table's `PRIMARY KEY`, use the quoted identifier `` `PRIMARY` ``:

```
HANDLER tbl_name READ `PRIMARY` ...
```

The second `HANDLER ... READ` syntax fetches a row from the table in index order that matches the `WHERE` condition.

The third `HANDLER ... READ` syntax fetches a row from the table in natural row order that matches the `WHERE` condition. It is faster than `HANDLER tbl_name READ index_name` when a full table scan is desired. Natural row order is the order in which rows are stored in a `MyISAM` table data file. This statement works for `InnoDB` tables as well, but there is no such concept because there is no separate data file.

Without a `LIMIT` clause, all forms of `HANDLER ... READ` fetch a single row if one is available. To return a specific number of rows, include a `LIMIT` clause. It has the same syntax as for the `SELECT` statement. See Section 13.2.13, “SELECT Statement”.

`HANDLER ... CLOSE` closes a table that was opened with `HANDLER ... OPEN`.

There are several reasons to use the `HANDLER` interface instead of normal `SELECT` statements:

- `HANDLER` is faster than `SELECT`:
  - A designated storage engine handler object is allocated for the `HANDLER ... OPEN`. The object is reused for subsequent `HANDLER` statements for that table; it need not be reinitialized for each one.
  - There is less parsing involved.
  - There is no optimizer or query-checking overhead.
  - The handler interface does not have to provide a consistent look of the data (for example, dirty reads are permitted), so the storage engine can use optimizations that `SELECT` does not normally permit.
- `HANDLER` makes it easier to port to MySQL applications that use a low-level `ISAM`-like interface. (See Section 15.20, “InnoDB memcached Plugin” for an alternative way to adapt applications that use the key-value store paradigm.)
- `HANDLER` enables you to traverse a database in a manner that is difficult (or even impossible) to accomplish with `SELECT`. The `HANDLER` interface is a more natural way to look at data when working with applications that provide an interactive user interface to the database.

`HANDLER` is a somewhat low-level statement. For example, it does not provide consistency. That is, `HANDLER ... OPEN` does *not* take a snapshot of the table, and does *not* lock the table. This means that after a `HANDLER ... OPEN` statement is issued, table data can be modified (by the current session or other sessions) and these modifications might be only partially visible to `HANDLER ... NEXT` or `HANDLER ... PREV` scans.

An open handler can be closed and marked for reopen, in which case the handler loses its position in the table. This occurs when both of the following circumstances are true:

- Any session executes `FLUSH TABLES` or DDL statements on the handler's table.
- The session in which the handler is open executes non-`HANDLER` statements that use tables.

`TRUNCATE TABLE` for a table closes all handlers for the table that were opened with `HANDLER OPEN`.

If a table is flushed with `FLUSH TABLES tbl_name WITH READ LOCK` was opened with `HANDLER`, the handler is implicitly flushed and loses its position.

13.2.6 IMPORT TABLE Statement

```
IMPORT TABLE FROM sdi_file [, sdi_file] ...
```

The `IMPORT TABLE` statement imports `MyISAM` tables based on information contained in `.sdi` (serialized dictionary information) metadata files. `IMPORT TABLE` requires the `FILE` privilege to read the `.sdi` and table content files, and the `CREATE` privilege for the table to be created.

Tables can be exported from one server using `mysqldump` to write a file of SQL statements and imported into another server using `mysql` to process the dump file. `IMPORT TABLE` provides a faster alternative using the “raw” table files.

Prior to import, the files that provide the table content must be placed in the appropriate schema directory for the import server, and the `.sdi` file must be located in a directory accessible to the server. For example, the `.sdi` file can be placed in the directory named by the `secure_file_priv` system variable, or (if `secure_file_priv` is empty) in a directory under the server data directory.

The following example describes how to export `MyISAM` tables named `employees` and `managers` from the `hr` schema of one server and import them into the `hr` schema of another server. The example uses these assumptions (to perform a similar operation on your own system, modify the path names as appropriate):

- For the export server, *`export_basedir`* represents its base directory, and its data directory is *`export_basedir`*`/data`.
- For the import server, *`import_basedir`* represents its base directory, and its data directory is *`import_basedir`*`/data`.
- Table files are exported from the export server into the `/tmp/export` directory and this directory is secure (not accessible to other users).
- The import server uses `/tmp/mysql-files` as the directory named by its `secure_file_priv` system variable.

To export tables from the export server, use this procedure:

1. Ensure a consistent snapshot by executing this statement to lock the tables so that they cannot be modified during export:

   ```
   mysql> FLUSH TABLES hr.employees, hr.managers WITH READ LOCK;
   ```

   While the lock is in effect, the tables can still be used, but only for read access.

2. At the file system level, copy the `.sdi` and table content files from the `hr` schema directory to the secure export directory:

   - The `.sdi` file is located in the `hr` schema directory, but might not have exactly the same basename as the table name. For example, the `.sdi` files for the `employees` and `managers` tables might be named `employees_125.sdi` and `managers_238.sdi`.
   - For a `MyISAM` table, the content files are its `.MYD` data file and `.MYI` index file.

   Given those file names, the copy commands look like this:

   ```
   $> cd export_basedir/data/hr
   $> cp employees_125.sdi /tmp/export
   $> cp managers_238.sdi /tmp/export
   $> cp employees.{MYD,MYI} /tmp/export
   $> cp managers.{MYD,MYI} /tmp/export
   ```

3. Unlock the tables:

   ```
   mysql> UNLOCK TABLES;
   ```

To import tables into the import server, use this procedure:

1. The import schema must exist. If necessary, execute this statement to create it:

   ```
   mysql> CREATE SCHEMA hr;
   ```

2. At the file system level, copy the `.sdi` files to the import server `secure_file_priv` directory, `/tmp/mysql-files`. Also, copy the table content files to the `hr` schema directory:

   ```
   $> cd /tmp/export
   $> cp employees_125.sdi /tmp/mysql-files
   $> cp managers_238.sdi /tmp/mysql-files
   $> cp employees.{MYD,MYI} import_basedir/data/hr
   $> cp managers.{MYD,MYI} import_basedir/data/hr
   ```

3. Import the tables by executing an `IMPORT TABLE` statement that names the `.sdi` files:

   ```
   mysql> IMPORT TABLE FROM
   ```

```
        '/tmp/mysql-files/employees.sdi',
        '/tmp/mysql-files/managers.sdi';
```

The `.sdi` file need not be placed in the import server directory named by the `secure_file_priv` system variable if that variable is empty; it can be in any directory accessible to the server, including the schema directory for the imported table. If the `.sdi` file is placed in that directory, however, it may be rewritten; the import operation creates a new `.sdi` file for the table, which overwrites the old `.sdi` file if the operation uses the same file name for the new file.

Each *`sdi_file`* value must be a string literal that names the `.sdi` file for a table or is a pattern that matches `.sdi` files. If the string is a pattern, any leading directory path and the `.sdi` file name suffix must be given literally. Pattern characters are permitted only in the base name part of the file name:

- `?` matches any single character
- `*` matches any sequence of characters, including no characters

Using a pattern, the previous `IMPORT TABLE` statement could have been written like this (assuming that the `/tmp/mysql-files` directory contains no other `.sdi` files matching the pattern):

```
IMPORT TABLE FROM '/tmp/mysql-files/*.sdi';
```

To interpret the location of `.sdi` file path names, the server uses the same rules for `IMPORT TABLE` as the server-side rules for `LOAD DATA` (that is, the non-`LOCAL` rules). See Section 13.2.9, “LOAD DATA Statement”, paying particular attention to the rules used to interpret relative path names.

`IMPORT TABLE` fails if the `.sdi` or table files cannot be located. After importing a table, the server attempts to open it and reports as warnings any problems detected. To attempt a repair to correct any reported issues, use `REPAIR TABLE`.

`IMPORT TABLE` is not written to the binary log.

Restrictions and Limitations

`IMPORT TABLE` applies only to non-`TEMPORARY` `MyISAM` tables. It does not apply to tables created with a transactional storage engine, tables created with `CREATE TEMPORARY TABLE`, or views.

An `.sdi` file used in an import operation must be generated on a server with the same data dictionary version and sdi version as the import server. The version information of the generating server is found in the `.sdi` file:

```
{
   "mysqld_version_id":80019,
   "dd_version":80017,
   "sdi_version":80016,
   ...
}
```

To determine the data dictionary and sdi version of the import server, you can check the `.sdi` file of a recently created table on the import server.

The table data and index files must be placed in the schema directory for the import server prior to the import operation, unless the table as defined on the export server uses the `DATA DIRECTORY` or `INDEX DIRECTORY` table options. In that case, modify the import procedure using one of these alternatives before executing the `IMPORT TABLE` statement:

- Put the data and index files into the same directory on the import server host as on the export server host, and create symlinks in the import server schema directory to those files.
- Put the data and index files into an import server host directory different from that on the export server host, and create symlinks in the import server schema directory to those files. In addition, modify the `.sdi` file to reflect the different file locations.

- Put the data and index files into the schema directory on the import server host, and modify the `.sdi` file to remove the data and index directory table options.

Any collation IDs stored in the `.sdi` file must refer to the same collations on the export and import servers.

Trigger information for a table is not serialized into the table `.sdi` file, so triggers are not restored by the import operation.

Some edits to an `.sdi` file are permissible prior to executing the `IMPORT TABLE` statement, whereas others are problematic or may even cause the import operation to fail:

- Changing the data directory and index directory table options is required if the locations of the data and index files differ between the export and import servers.
- Changing the schema name is required to import the table into a different schema on the import server than on the export server.
- Changing schema and table names may be required to accommodate differences between file system case-sensitivity semantics on the export and import servers or differences in `lower_case_table_names` settings. Changing the table names in the `.sdi` file may require renaming the table files as well.
- In some cases, changes to column definitions are permitted. Changing data types is likely to cause problems.

13.2.7 INSERT Statement

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    { {VALUES | VALUE} (value_list) [, (value_list)] ... }
    [AS row_alias[(col_alias [, col_alias] ...)]]
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    SET assignment_list
    [AS row_alias[(col_alias [, col_alias] ...)]]
    [ON DUPLICATE KEY UPDATE assignment_list]

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    { SELECT ...
      | TABLE table_name
      | VALUES row_constructor_list
    }
    [ON DUPLICATE KEY UPDATE assignment_list]

value:
    {expr | DEFAULT}

value_list:
    value [, value] ...

row_constructor_list:
    ROW(value_list)[, ROW(value_list)][, ...]

assignment:
    col_name =
          value
        | [row_alias.]col_name
        | [tbl_name.]col_name
        | [row_alias.]col_alias
```

```
assignment_list:
    assignment [, assignment] ...
```

`INSERT` inserts new rows into an existing table. The `INSERT ... VALUES`, `INSERT ... VALUES ROW()`, and `INSERT ... SET` forms of the statement insert rows based on explicitly specified values. The `INSERT ... SELECT` form inserts rows selected from another table or tables. You can also use `INSERT ... TABLE` in MySQL 8.0.19 and later to insert rows from a single table. `INSERT` with an `ON DUPLICATE KEY UPDATE` clause enables existing rows to be updated if a row to be inserted would cause a duplicate value in a `UNIQUE` index or `PRIMARY KEY`. In MySQL 8.0.19 and later, a row alias with one or more optional column aliases can be used with `ON DUPLICATE KEY UPDATE` to refer to the row to be inserted.

For additional information about `INSERT ... SELECT` and `INSERT ... ON DUPLICATE KEY UPDATE`, see Section 13.2.7.1, “INSERT ... SELECT Statement”, and Section 13.2.7.2, “INSERT ... ON DUPLICATE KEY UPDATE Statement”.

In MySQL 8.0, the `DELAYED` keyword is accepted but ignored by the server. For the reasons for this, see Section 13.2.7.3, “INSERT DELAYED Statement”,

Inserting into a table requires the `INSERT` privilege for the table. If the `ON DUPLICATE KEY UPDATE` clause is used and a duplicate key causes an `UPDATE` to be performed instead, the statement requires the `UPDATE` privilege for the columns to be updated. For columns that are read but not modified you need only the `SELECT` privilege (such as for a column referenced only on the right hand side of an *`col_name`*=*`expr`* assignment in an `ON DUPLICATE KEY UPDATE` clause).

When inserting into a partitioned table, you can control which partitions and subpartitions accept new rows. The `PARTITION` clause takes a list of the comma-separated names of one or more partitions or subpartitions (or both) of the table. If any of the rows to be inserted by a given `INSERT` statement do not match one of the partitions listed, the `INSERT` statement fails with the error `Found a row not matching the given partition set`. For more information and examples, see Section 24.5, “Partition Selection”.

`tbl_name` is the table into which rows should be inserted. Specify the columns for which the statement provides values as follows:

- Provide a parenthesized list of comma-separated column names following the table name. In this case, a value for each named column must be provided by the `VALUES` list, `VALUES ROW()` list, or `SELECT` statement. For the `INSERT TABLE` form, the number of columns in the source table must match the number of columns to be inserted.
- If you do not specify a list of column names for `INSERT ... VALUES` or `INSERT ... SELECT`, values for every column in the table must be provided by the `VALUES` list, `SELECT` statement, or `TABLE` statement. If you do not know the order of the columns in the table, use `DESCRIBE` *`tbl_name`* to find out.
- A `SET` clause indicates columns explicitly by name, together with the value to assign each one.

Column values can be given in several ways:

- If strict SQL mode is not enabled, any column not explicitly given a value is set to its default (explicit or implicit) value. For example, if you specify a column list that does not name all the columns in the table, unnamed columns are set to their default values. Default value assignment is described in Section 11.6, “Data Type Default Values”. See also Section 1.6.3.3, “Enforced Constraints on Invalid Data”.

  If strict SQL mode is enabled, an `INSERT` statement generates an error if it does not specify an explicit value for every column that has no default value. See Section 5.1.11, “Server SQL Modes”.
- If both the column list and the `VALUES` list are empty, `INSERT` creates a row with each column set to its default value:

```
INSERT INTO tbl_name () VALUES();
```

If strict mode is not enabled, MySQL uses the implicit default value for any column that has no explicitly defined default. If strict mode is enabled, an error occurs if any column has no default value.

- Use the keyword `DEFAULT` to set a column explicitly to its default value. This makes it easier to write `INSERT` statements that assign values to all but a few columns, because it enables you to avoid writing an incomplete `VALUES` list that does not include a value for each column in the table. Otherwise, you must provide the list of column names corresponding to each value in the `VALUES` list.

- If a generated column is inserted into explicitly, the only permitted value is `DEFAULT`. For information about generated columns, see Section 13.1.20.8, “CREATE TABLE and Generated Columns”.

- In expressions, you can use `DEFAULT(col_name)` to produce the default value for column *`col_name`*.

- Type conversion of an expression *`expr`* that provides a column value might occur if the expression data type does not match the column data type. Conversion of a given value can result in different inserted values depending on the column type. For example, inserting the string `'1999.0e-2'` into an `INT`, `FLOAT`, `DECIMAL(10,6)`, or `YEAR` column inserts the value `1999`, `19.9921`, `19.992100`, or `1999`, respectively. The value stored in the `INT` and `YEAR` columns is `1999` because the string-to-number conversion looks only at as much of the initial part of the string as may be considered a valid integer or year. For the `FLOAT` and `DECIMAL` columns, the string-to-number conversion considers the entire string a valid numeric value.

- An expression *`expr`* can refer to any column that was set earlier in a value list. For example, you can do this because the value for `col2` refers to `col1`, which has previously been assigned:

  ```
  INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);
  ```

  But the following is not legal, because the value for `col1` refers to `col2`, which is assigned after `col1`:

  ```
  INSERT INTO tbl_name (col1,col2) VALUES(col2*2,15);
  ```

  An exception occurs for columns that contain `AUTO_INCREMENT` values. Because `AUTO_INCREMENT` values are generated after other value assignments, any reference to an `AUTO_INCREMENT` column in the assignment returns a `0`.

`INSERT` statements that use `VALUES` syntax can insert multiple rows. To do this, include multiple lists of comma-separated column values, with lists enclosed within parentheses and separated by commas. Example:

```
INSERT INTO tbl_name (a,b,c)
    VALUES(1,2,3), (4,5,6), (7,8,9);
```

Each values list must contain exactly as many values as are to be inserted per row. The following statement is invalid because it contains one list of nine values, rather than three lists of three values each:

```
INSERT INTO tbl_name (a,b,c) VALUES(1,2,3,4,5,6,7,8,9);
```

`VALUE` is a synonym for `VALUES` in this context. Neither implies anything about the number of values lists, nor about the number of values per list. Either may be used whether there is a single values list or multiple lists, and regardless of the number of values per list.

`INSERT` statements using `VALUES ROW()` syntax can also insert multiple rows. In this case, each value list must be contained within a `ROW()` (row constructor), like this:

```
INSERT INTO tbl_name (a,b,c)
```

```
    VALUES ROW(1,2,3), ROW(4,5,6), ROW(7,8,9);
```

The affected-rows value for an `INSERT` can be obtained using the `ROW_COUNT()` SQL function or the `mysql_affected_rows()` C API function. See Section 12.16, “Information Functions”, and mysql_affected_rows().

If you use `INSERT ... VALUES` or `INSERT ... VALUES ROW()` with multiple value lists, or `INSERT ... SELECT` or `INSERT ... TABLE`, the statement returns an information string in this format:

```
Records: N1 Duplicates: N2 Warnings: N3
```

If you are using the C API, the information string can be obtained by invoking the `mysql_info()` function. See mysql_info().

`Records` indicates the number of rows processed by the statement. (This is not necessarily the number of rows actually inserted because `Duplicates` can be nonzero.) `Duplicates` indicates the number of rows that could not be inserted because they would duplicate some existing unique index value. `Warnings` indicates the number of attempts to insert column values that were problematic in some way. Warnings can occur under any of the following conditions:

- Inserting `NULL` into a column that has been declared `NOT NULL`. For multiple-row `INSERT` statements or `INSERT INTO ... SELECT` statements, the column is set to the implicit default value for the column data type. This is `0` for numeric types, the empty string (`''`) for string types, and the “zero” value for date and time types. `INSERT INTO ... SELECT` statements are handled the same way as multiple-row inserts because the server does not examine the result set from the `SELECT` to see whether it returns a single row. (For a single-row `INSERT`, no warning occurs when `NULL` is inserted into a `NOT NULL` column. Instead, the statement fails with an error.)

- Setting a numeric column to a value that lies outside the column range. The value is clipped to the closest endpoint of the range.

- Assigning a value such as `'10.34 a'` to a numeric column. The trailing nonnumeric text is stripped off and the remaining numeric part is inserted. If the string value has no leading numeric part, the column is set to `0`.

- Inserting a string into a string column (`CHAR`, `VARCHAR`, `TEXT`, or `BLOB`) that exceeds the column maximum length. The value is truncated to the column maximum length.

- Inserting a value into a date or time column that is illegal for the data type. The column is set to the appropriate zero value for the type.

- For `INSERT` examples involving `AUTO_INCREMENT` column values, see Section 3.6.9, “Using AUTO_INCREMENT”.

  If `INSERT` inserts a row into a table that has an `AUTO_INCREMENT` column, you can find the value used for that column by using the `LAST_INSERT_ID()` SQL function or the `mysql_insert_id()` C API function.

Note

These two functions do not always behave identically. The behavior of `INSERT` statements with respect to `AUTO_INCREMENT` columns is discussed further in Section 12.16, “Information Functions”, and mysql_insert_id().

The `INSERT` statement supports the following modifiers:

- If you use the `LOW_PRIORITY` modifier, execution of the `INSERT` is delayed until no other clients are reading from the table. This includes other clients that began reading while existing clients are reading, and while the `INSERT LOW_PRIORITY` statement is waiting. It is possible, therefore, for a client that issues an `INSERT LOW_PRIORITY` statement to wait for a very long time.

`LOW_PRIORITY` affects only storage engines that use only table-level locking (such as `MyISAM`, `MEMORY`, and `MERGE`).

Note

`LOW_PRIORITY` should normally not be used with `MyISAM` tables because doing so disables concurrent inserts. See Section 8.11.3, “Concurrent Inserts”.

- If you specify `HIGH_PRIORITY`, it overrides the effect of the `--low-priority-updates` option if the server was started with that option. It also causes concurrent inserts not to be used. See Section 8.11.3, “Concurrent Inserts”.

  `HIGH_PRIORITY` affects only storage engines that use only table-level locking (such as `MyISAM`, `MEMORY`, and `MERGE`).

- If you use the `IGNORE` modifier, ignorable errors that occur while executing the `INSERT` statement are ignored. For example, without `IGNORE`, a row that duplicates an existing `UNIQUE` index or `PRIMARY KEY` value in the table causes a duplicate-key error and the statement is aborted. With `IGNORE`, the row is discarded and no error occurs. Ignored errors generate warnings instead.

  `IGNORE` has a similar effect on inserts into partitioned tables where no partition matching a given value is found. Without `IGNORE`, such `INSERT` statements are aborted with an error. When `INSERT IGNORE` is used, the insert operation fails silently for rows containing the unmatched value, but inserts rows that are matched. For an example, see Section 24.2.2, “LIST Partitioning”.

  Data conversions that would trigger errors abort the statement if `IGNORE` is not specified. With `IGNORE`, invalid values are adjusted to the closest values and inserted; warnings are produced but the statement does not abort. You can determine with the `mysql_info()` C API function how many rows were actually inserted into the table.

  For more information, see The Effect of IGNORE on Statement Execution.

  You can use `REPLACE` instead of `INSERT` to overwrite old rows. `REPLACE` is the counterpart to `INSERT IGNORE` in the treatment of new rows that contain unique key values that duplicate old rows: The new rows replace the old rows rather than being discarded. See Section 13.2.12, “REPLACE Statement”.

- If you specify `ON DUPLICATE KEY UPDATE`, and a row is inserted that would cause a duplicate value in a `UNIQUE` index or `PRIMARY KEY`, an `UPDATE` of the old row occurs. The affected-rows value per row is 1 if the row is inserted as a new row, 2 if an existing row is updated, and 0 if an existing row is set to its current values. If you specify the `CLIENT_FOUND_ROWS` flag to the `mysql_real_connect()` C API function when connecting to `mysqld`, the affected-rows value is 1 (not 0) if an existing row is set to its current values. See Section 13.2.7.2, “INSERT ... ON DUPLICATE KEY UPDATE Statement”.

- `INSERT DELAYED` was deprecated in MySQL 5.6, and is scheduled for eventual removal. In MySQL 8.0, the `DELAYED` modifier is accepted but ignored. Use `INSERT` (without `DELAYED`) instead. See Section 13.2.7.3, “INSERT DELAYED Statement”.

13.2.7.1 INSERT ... SELECT Statement

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {   SELECT ...
      | TABLE table_name
      | VALUES row_constructor_list
    }
    [ON DUPLICATE KEY UPDATE assignment_list]
```

```
value:
    {expr | DEFAULT}

value_list:
    value [, value] ...

row_constructor_list:
    ROW(value_list)[, ROW(value_list)][, ...]

assignment:
    col_name =
          value
        | [row_alias.]col_name
        | [tbl_name.]col_name
        | [row_alias.]col_alias

assignment_list:
    assignment [, assignment] ...
```

With `INSERT ... SELECT`, you can quickly insert many rows into a table from the result of a `SELECT` statement, which can select from one or many tables. For example:

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

Beginning with MySQL 8.0.19, you can use a `TABLE` statement in place of `SELECT`, as shown here:

```
INSERT INTO ta TABLE tb;
```

`TABLE tb` is equivalent to `SELECT * FROM tb`. It can be useful when inserting all columns from the source table into the target table, and no filtering with WHERE is required. In addition, the rows from `TABLE` can be ordered by one or more columns using `ORDER BY`, and the number of rows inserted can be limited using a `LIMIT` clause. For more information, see Section 13.2.16, “TABLE Statement”.

The following conditions hold for `INSERT ... SELECT` statements, and, except where noted, for `INSERT ... TABLE` as well:

- Specify `IGNORE` to ignore rows that would cause duplicate-key violations.
- The target table of the `INSERT` statement may appear in the `FROM` clause of the `SELECT` part of the query, or as the table named by `TABLE`. However, you cannot insert into a table and select from the same table in a subquery.

  When selecting from and inserting into the same table, MySQL creates an internal temporary table to hold the rows from the `SELECT` and then inserts those rows into the target table. However, you cannot use `INSERT INTO t ... SELECT ... FROM t` when `t` is a `TEMPORARY` table, because `TEMPORARY` tables cannot be referred to twice in the same statement. For the same reason, you cannot use `INSERT INTO t ... TABLE t` when `t` is a temporary table. See Section 8.4.4, “Internal Temporary Table Use in MySQL”, and Section B.3.6.2, “TEMPORARY Table Problems”.
- `AUTO_INCREMENT` columns work as usual.
- To ensure that the binary log can be used to re-create the original tables, MySQL does not permit concurrent inserts for `INSERT ... SELECT` or `INSERT ... TABLE` statements (see Section 8.11.3, “Concurrent Inserts”).
- To avoid ambiguous column reference problems when the `SELECT` and the `INSERT` refer to the same table, provide a unique alias for each table used in the `SELECT` part, and qualify column names in that part with the appropriate alias.

  The `TABLE` statement does not support aliases.

You can explicitly select which partitions or subpartitions (or both) of the source or target table (or both) are to be used with a `PARTITION` clause following the name of the table. When `PARTITION`

is used with the name of the source table in the SELECT portion of the statement, rows are selected only from the partitions or subpartitions named in its partition list. When PARTITION is used with the name of the target table for the INSERT portion of the statement, it must be possible to insert all rows selected into the partitions or subpartitions named in the partition list following the option. Otherwise, the INSERT ... SELECT statement fails. For more information and examples, see Section 24.5, "Partition Selection".

TABLE does not support a PARTITION clause.

For INSERT ... SELECT statements, see Section 13.2.7.2, "INSERT ... ON DUPLICATE KEY UPDATE Statement" for conditions under which the SELECT columns can be referred to in an ON DUPLICATE KEY UPDATE clause. This also works for INSERT ... TABLE.

The order in which a SELECT or TABLE statement with no ORDER BY clause returns rows is nondeterministic. This means that, when using replication, there is no guarantee that such a SELECT returns rows in the same order on the source and the replica, which can lead to inconsistencies between them. To prevent this from occurring, always write INSERT ... SELECT or INSERT ... TABLE statements that are to be replicated using an ORDER BY clause that produces the same row order on the source and the replica. See also Section 17.5.1.18, "Replication and LIMIT".

Due to this issue, INSERT ... SELECT ON DUPLICATE KEY UPDATE and INSERT IGNORE ... SELECT statements are flagged as unsafe for statement-based replication. Such statements produce a warning in the error log when using statement-based mode and are written to the binary log using the row-based format when using MIXED mode. (Bug #11758262, Bug #50439)

See also Section 17.2.1.1, "Advantages and Disadvantages of Statement-Based and Row-Based Replication".

13.2.7.2 INSERT ... ON DUPLICATE KEY UPDATE Statement

If you specify an ON DUPLICATE KEY UPDATE clause and a row to be inserted would cause a duplicate value in a UNIQUE index or PRIMARY KEY, an UPDATE of the old row occurs. For example, if column a is declared as UNIQUE and contains the value 1, the following two statements have similar effect:

```
INSERT INTO t1 (a,b,c) VALUES (1,2,3)
  ON DUPLICATE KEY UPDATE c=c+1;

UPDATE t1 SET c=c+1 WHERE a=1;
```

The effects are not quite identical: For an InnoDB table where a is an auto-increment column, the INSERT statement increases the auto-increment value but the UPDATE does not.

If column b is also unique, the INSERT is equivalent to this UPDATE statement instead:

```
UPDATE t1 SET c=c+1 WHERE a=1 OR b=2 LIMIT 1;
```

If a=1 OR b=2 matches several rows, only *one* row is updated. In general, you should try to avoid using an ON DUPLICATE KEY UPDATE clause on tables with multiple unique indexes.

With ON DUPLICATE KEY UPDATE, the affected-rows value per row is 1 if the row is inserted as a new row, 2 if an existing row is updated, and 0 if an existing row is set to its current values. If you specify the CLIENT_FOUND_ROWS flag to the mysql_real_connect() C API function when connecting to mysqld, the affected-rows value is 1 (not 0) if an existing row is set to its current values.

If a table contains an AUTO_INCREMENT column and INSERT ... ON DUPLICATE KEY UPDATE inserts or updates a row, the LAST_INSERT_ID() function returns the AUTO_INCREMENT value.

The ON DUPLICATE KEY UPDATE clause can contain multiple column assignments, separated by commas.

In assignment value expressions in the ON DUPLICATE KEY UPDATE clause, you can use the VALUES(*col_name*) function to refer to column values from the INSERT portion of the INSERT ...

`ON DUPLICATE KEY UPDATE` statement. In other words, `VALUES(col_name)` in the `ON DUPLICATE KEY UPDATE` clause refers to the value of *`col_name`* that would be inserted, had no duplicate-key conflict occurred. This function is especially useful in multiple-row inserts. The `VALUES()` function is meaningful only in the `ON DUPLICATE KEY UPDATE` clause or `INSERT` statements and returns `NULL` otherwise. Example:

```
INSERT INTO t1 (a,b,c) VALUES (1,2,3),(4,5,6)
  ON DUPLICATE KEY UPDATE c=VALUES(a)+VALUES(b);
```

That statement is identical to the following two statements:

```
INSERT INTO t1 (a,b,c) VALUES (1,2,3)
  ON DUPLICATE KEY UPDATE c=3;
INSERT INTO t1 (a,b,c) VALUES (4,5,6)
  ON DUPLICATE KEY UPDATE c=9;
```

Note

The use of `VALUES()` to refer to the new row and columns is deprecated beginning with MySQL 8.0.20, and is subject to removal in a future version of MySQL. Instead, use row and column aliases, as described in the next few paragraphs of this section.

Beginning with MySQL 8.0.19, it is possible to use an alias for the row, with, optionally, one or more of its columns to be inserted, following the `VALUES` or `SET` clause, and preceded by the `AS` keyword. Using the row alias `new`, the statement shown previously using `VALUES()` to access the new column values can be written in the form shown here:

```
INSERT INTO t1 (a,b,c) VALUES (1,2,3),(4,5,6) AS new
  ON DUPLICATE KEY UPDATE c = new.a+new.b;
```

If, in addition, you use the column aliases `m`, `n`, and `p`, you can omit the row alias in the assignment clause and write the same statement like this:

```
INSERT INTO t1 (a,b,c) VALUES (1,2,3),(4,5,6) AS new(m,n,p)
  ON DUPLICATE KEY UPDATE c = m+n;
```

When using column aliases in this fashion, you must still use a row alias following the `VALUES` clause, even if you do not make direct use of it in the assignment clause.

Beginning with MySQL 8.0.20, an `INSERT ... SELECT ... ON DUPLICATE KEY UPDATE` statement that uses `VALUES()` in the `UPDATE` clause, like this one, throws a warning:

```
INSERT INTO t1
  SELECT c, c+d FROM t2
  ON DUPLICATE KEY UPDATE b = VALUES(b);
```

You can eliminate such warnings by using a subquery instead, like this:

```
INSERT INTO t1
  SELECT * FROM (SELECT c, c+d AS e FROM t2) AS dt
  ON DUPLICATE KEY UPDATE b = e;
```

You can also use row and column aliases with a `SET` clause, as mentioned previously. Employing `SET` instead of `VALUES` in the two `INSERT ... ON DUPLICATE KEY UPDATE` statements just shown can be done as shown here:

```
INSERT INTO t1 SET a=1,b=2,c=3 AS new
  ON DUPLICATE KEY UPDATE c = new.a+new.b;

INSERT INTO t1 SET a=1,b=2,c=3 AS new(m,n,p)
  ON DUPLICATE KEY UPDATE c = m+n;
```

The row alias must not be the same as the name of the table. If column aliases are not used, or if they are the same as the column names, they must be distinguished using the row alias in the `ON`

DUPLICATE KEY UPDATE clause. Column aliases must be unique with regard to the row alias to which they apply (that is, no column aliases referring to columns of the same row may be the same).

For INSERT ... SELECT statements, these rules apply regarding acceptable forms of SELECT query expressions that you can refer to in an ON DUPLICATE KEY UPDATE clause:

- References to columns from queries on a single table, which may be a derived table.
- References to columns from queries on a join over multiple tables.
- References to columns from DISTINCT queries.
- References to columns in other tables, as long as the SELECT does not use GROUP BY. One side effect is that you must qualify references to nonunique column names.

References to columns from a UNION are not supported. To work around this restriction, rewrite the UNION as a derived table so that its rows can be treated as a single-table result set. For example, this statement produces an error:

```
INSERT INTO t1 (a, b)
  SELECT c, d FROM t2
  UNION
  SELECT e, f FROM t3
ON DUPLICATE KEY UPDATE b = b + c;
```

Instead, use an equivalent statement that rewrites the UNION as a derived table:

```
INSERT INTO t1 (a, b)
SELECT * FROM
  (SELECT c, d FROM t2
   UNION
   SELECT e, f FROM t3) AS dt
ON DUPLICATE KEY UPDATE b = b + c;
```

The technique of rewriting a query as a derived table also enables references to columns from GROUP BY queries.

Because the results of INSERT ... SELECT statements depend on the ordering of rows from the SELECT and this order cannot always be guaranteed, it is possible when logging INSERT ... SELECT ON DUPLICATE KEY UPDATE statements for the source and the replica to diverge. Thus, INSERT ... SELECT ON DUPLICATE KEY UPDATE statements are flagged as unsafe for statement-based replication. Such statements produce a warning in the error log when using statement-based mode and are written to the binary log using the row-based format when using MIXED mode. An INSERT ... ON DUPLICATE KEY UPDATE statement against a table having more than one unique or primary key is also marked as unsafe. (Bug #11765650, Bug #58637)

See also Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

13.2.7.3 INSERT DELAYED Statement

```
INSERT DELAYED ...
```

The DELAYED option for the INSERT statement is a MySQL extension to standard SQL. In previous versions of MySQL, it can be used for certain kinds of tables (such as MyISAM), such that when a client uses INSERT DELAYED, it gets an okay from the server at once, and the row is queued to be inserted when the table is not in use by any other thread.

DELAYED inserts and replaces were deprecated in MySQL 5.6. In MySQL 8.0, DELAYED is not supported. The server recognizes but ignores the DELAYED keyword, handles the insert as a nondelayed insert, and generates an ER_WARN_LEGACY_SYNTAX_CONVERTED warning: INSERT DELAYED is no longer supported. The statement was converted to INSERT. The DELAYED keyword is scheduled for removal in a future release.

13.2.8 INTERSECT Clause

```
query_expression_body INTERSECT [ALL | DISTINCT] query_expression_body
    [INTERSECT [ALL | DISTINCT] query_expression_body]
    [...]

query_expression_body:
    See Section 13.2.14, “Set Operations with UNION, INTERSECT, and EXCEPT”
```

`INTERSECT` limits the result from multiple query blocks to those rows which are common to all. Example:

```
mysql> TABLE a;
+------+------+
| m    | n    |
+------+------+
|    1 |    2 |
|    2 |    3 |
|    3 |    4 |
+------+------+
3 rows in set (0.00 sec)

mysql> TABLE b;
+------+------+
| m    | n    |
+------+------+
|    1 |    2 |
|    1 |    3 |
|    3 |    4 |
+------+------+
3 rows in set (0.00 sec)

mysql> TABLE c;
+------+------+
| m    | n    |
+------+------+
|    1 |    3 |
|    1 |    3 |
|    3 |    4 |
+------+------+
3 rows in set (0.00 sec)

mysql> TABLE a INTERSECT TABLE b;
+------+------+
| m    | n    |
+------+------+
|    1 |    2 |
|    3 |    4 |
+------+------+
2 rows in set (0.00 sec)

mysql> TABLE a INTERSECT TABLE c;
+------+------+
| m    | n    |
+------+------+
|    3 |    4 |
+------+------+
1 row in set (0.00 sec)
```

As with `UNION` and `EXCEPT`, if neither `DISTINCT` nor `ALL` is specified, the default is `DISTINCT`.

`DISTINCT` can remove duplicates from either side of the intersection, as shown here:

```
mysql> TABLE c INTERSECT DISTINCT TABLE c;
+------+------+
| m    | n    |
+------+------+
|    1 |    3 |
|    3 |    4 |
+------+------+
2 rows in set (0.00 sec)
```

```
mysql> TABLE c INTERSECT ALL TABLE c;
+------+------+
| m    | n    |
+------+------+
|    1 |    3 |
|    1 |    3 |
|    3 |    4 |
+------+------+
3 rows in set (0.00 sec)
```

(`TABLE c INTERSECT TABLE c` is the equivalent of the first of the two statements just shown.)

As with `UNION`, the operands must have the same number of columns. Result set column types are also determined as for `UNION`.

`INTERSECT` has greater precedence than and is evaluated before `UNION` and `EXCEPT`, so that the two statements shown here are equivalent:

```
TABLE r EXCEPT TABLE s INTERSECT TABLE t;

TABLE r EXCEPT (TABLE s INTERSECT TABLE t);
```

For `INTERSECT ALL`, the maximum supported number of duplicates of any unique row in the left hand table is `4294967295`.

`INTERSECT` was added in MySQL 8.0.31.

13.2.9 LOAD DATA Statement

```
LOAD DATA
    [LOW_PRIORITY | CONCURRENT] [LOCAL]
    INFILE 'file_name'
    [REPLACE | IGNORE]
    INTO TABLE tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [CHARACTER SET charset_name]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [(col_name_or_user_var
        [, col_name_or_user_var] ...)]
    [SET col_name={expr | DEFAULT}
        [, col_name={expr | DEFAULT}] ...]
```

The `LOAD DATA` statement reads rows from a text file into a table at a very high speed. The file can be read from the server host or the client host, depending on whether the `LOCAL` modifier is given. `LOCAL` also affects data interpretation and error handling.

`LOAD DATA` is the complement of `SELECT ... INTO OUTFILE`. (See Section 13.2.13.1, “SELECT ... INTO Statement”.) To write data from a table to a file, use `SELECT ... INTO OUTFILE`. To read the file back into a table, use `LOAD DATA`. The syntax of the `FIELDS` and `LINES` clauses is the same for both statements.

The `mysqlimport` utility provides another way to load data files; it operates by sending a `LOAD DATA` statement to the server. See Section 4.5.5, “mysqlimport — A Data Import Program”.

For information about the efficiency of `INSERT` versus `LOAD DATA` and speeding up `LOAD DATA`, see Section 8.2.5.1, “Optimizing INSERT Statements”.

Non-LOCAL Versus LOCAL Operation

The `LOCAL` modifier affects these aspects of `LOAD DATA`, compared to non-`LOCAL` operation:

- It changes the expected location of the input file; see Input File Location.
- It changes the statement security requirements; see Security Requirements.
- It has the same effect as the `IGNORE` modifier on the interpretation of input file contents and error handling; see Duplicate-Key and Error Handling, and Column Value Assignment.

`LOCAL` works only if the server and your client both have been configured to permit it. For example, if `mysqld` was started with the `local_infile` system variable disabled, `LOCAL` produces an error. See Section 6.1.6, “Security Considerations for LOAD DATA LOCAL”.

Input File Character Set

The file name must be given as a literal string. On Windows, specify backslashes in path names as forward slashes or doubled backslashes. The server interprets the file name using the character set indicated by the `character_set_filesystem` system variable.

By default, the server interprets the file contents using the character set indicated by the `character_set_database` system variable. If the file contents use a character set different from this default, it is a good idea to specify that character set by using the `CHARACTER SET` clause. A character set of `binary` specifies “no conversion.”

`SET NAMES` and the setting of `character_set_client` do not affect interpretation of file contents.

`LOAD DATA` interprets all fields in the file as having the same character set, regardless of the data types of the columns into which field values are loaded. For proper interpretation of the file, you must ensure that it was written with the correct character set. For example, if you write a data file with `mysqldump -T` or by issuing a `SELECT ... INTO OUTFILE` statement in `mysql`, be sure to use a

`--default-character-set` option to write output in the character set to be used when the file is loaded with `LOAD DATA`.

Note

It is not possible to load data files that use the `ucs2`, `utf16`, `utf16le`, or `utf32` character set.

Input File Location

These rules determine the `LOAD DATA` input file location:

- If `LOCAL` is not specified, the file must be located on the server host. The server reads the file directly, locating it as follows:
  - If the file name is an absolute path name, the server uses it as given.
  - If the file name is a relative path name with leading components, the server looks for the file relative to its data directory.
  - If the file name has no leading components, the server looks for the file in the database directory of the default database.
- If `LOCAL` is specified, the file must be located on the client host. The client program reads the file, locating it as follows:
  - If the file name is an absolute path name, the client program uses it as given.
  - If the file name is a relative path name, the client program looks for the file relative to its invocation directory.

  When `LOCAL` is used, the client program reads the file and sends its contents to the server. The server creates a copy of the file in the directory where it stores temporary files. See Section B.3.3.5, “Where MySQL Stores Temporary Files”. Lack of sufficient space for the copy in this directory can cause the `LOAD DATA LOCAL` statement to fail.

The non-`LOCAL` rules mean that the server reads a file named as `./myfile.txt` relative to its data directory, whereas it reads a file named as `myfile.txt` from the database directory of the default database. For example, if the following `LOAD DATA` statement is executed while `db1` is the default database, the server reads the file `data.txt` from the database directory for `db1`, even though the statement explicitly loads the file into a table in the `db2` database:

```
LOAD DATA INFILE 'data.txt' INTO TABLE db2.my_table;
```

Note

The server also uses the non-`LOCAL` rules to locate `.sdi` files for the `IMPORT TABLE` statement.

Security Requirements

For a non-`LOCAL` load operation, the server reads a text file located on the server host, so these security requirements must be satisfied:

- You must have the `FILE` privilege. See Section 6.2.2, “Privileges Provided by MySQL”.
- The operation is subject to the `secure_file_priv` system variable setting:
  - If the variable value is a nonempty directory name, the file must be located in that directory.
  - If the variable value is empty (which is insecure), the file need only be readable by the server.

For a LOCAL load operation, the client program reads a text file located on the client host. Because the file contents are sent over the connection by the client to the server, using LOCAL is a bit slower than when the server accesses the file directly. On the other hand, you do not need the FILE privilege, and the file can be located in any directory the client program can access.

Duplicate-Key and Error Handling

The REPLACE and IGNORE modifiers control handling of new (input) rows that duplicate existing table rows on unique key values (PRIMARY KEY or UNIQUE index values):

- With REPLACE, new rows that have the same value as a unique key value in an existing row replace the existing row. See Section 13.2.12, “REPLACE Statement”.
- With IGNORE, new rows that duplicate an existing row on a unique key value are discarded. For more information, see The Effect of IGNORE on Statement Execution.

The LOCAL modifier has the same effect as IGNORE. This occurs because the server has no way to stop transmission of the file in the middle of the operation.

If none of REPLACE, IGNORE, or LOCAL is specified, an error occurs when a duplicate key value is found, and the rest of the text file is ignored.

In addition to affecting duplicate-key handling as just described, IGNORE and LOCAL also affect error handling:

- With neither IGNORE nor LOCAL, data-interpretation errors terminate the operation.
- With IGNORE or LOCAL, data-interpretation errors become warnings and the load operation continues, even if the SQL mode is restrictive. For examples, see Column Value Assignment.

Index Handling

To ignore foreign key constraints during the load operation, execute a SET foreign_key_checks = 0 statement before executing LOAD DATA.

If you use LOAD DATA on an empty MyISAM table, all nonunique indexes are created in a separate batch (as for REPAIR TABLE). Normally, this makes LOAD DATA much faster when you have many indexes. In some extreme cases, you can create the indexes even faster by turning them off with ALTER TABLE ... DISABLE KEYS before loading the file into the table and re-creating the indexes with ALTER TABLE ... ENABLE KEYS after loading the file. See Section 8.2.5.1, “Optimizing INSERT Statements”.

Field and Line Handling

For both the LOAD DATA and SELECT ... INTO OUTFILE statements, the syntax of the FIELDS and LINES clauses is the same. Both clauses are optional, but FIELDS must precede LINES if both are specified.

If you specify a FIELDS clause, each of its subclauses (TERMINATED BY, [OPTIONALLY] ENCLOSED BY, and ESCAPED BY) is also optional, except that you must specify at least one of them. Arguments to these clauses are permitted to contain only ASCII characters.

If you specify no FIELDS or LINES clause, the defaults are the same as if you had written this:

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES TERMINATED BY '\n' STARTING BY ''
```

Backslash is the MySQL escape character within strings in SQL statements. Thus, to specify a literal backslash, you must specify two backslashes for the value to be interpreted as a single backslash. The escape sequences '\t' and '\n' specify tab and newline characters, respectively.

In other words, the defaults cause `LOAD DATA` to act as follows when reading input:

- Look for line boundaries at newlines.
- Do not skip any line prefix.
- Break lines into fields at tabs.
- Do not expect fields to be enclosed within any quoting characters.
- Interpret characters preceded by the escape character `\` as escape sequences. For example, `\t`, `\n`, and `\\` signify tab, newline, and backslash, respectively. See the discussion of `FIELDS ESCAPED BY` later for the full list of escape sequences.

Conversely, the defaults cause `SELECT ... INTO OUTFILE` to act as follows when writing output:

- Write tabs between fields.
- Do not enclose fields within any quoting characters.
- Use `\` to escape instances of tab, newline, or `\` that occur within field values.
- Write newlines at the ends of lines.

Note

For a text file generated on a Windows system, proper file reading might require `LINES TERMINATED BY '\r\n'` because Windows programs typically use two characters as a line terminator. Some programs, such as `WordPad`, might use `\r` as a line terminator when writing files. To read such files, use `LINES TERMINATED BY '\r'`.

If all the input lines have a common prefix that you want to ignore, you can use `LINES STARTING BY '*prefix_string*'` to skip the prefix *and anything before it*. If a line does not include the prefix, the entire line is skipped. Suppose that you issue the following statement:

```
LOAD DATA INFILE '/tmp/test.txt' INTO TABLE test
  FIELDS TERMINATED BY ','  LINES STARTING BY 'xxx';
```

If the data file looks like this:

```
xxx"abc",1
something xxx"def",2
"ghi",3
```

The resulting rows are `("abc",1)` and `("def",2)`. The third row in the file is skipped because it does not contain the prefix.

The `IGNORE *number* LINES` clause can be used to ignore lines at the start of the file. For example, you can use `IGNORE 1 LINES` to skip an initial header line containing column names:

```
LOAD DATA INFILE '/tmp/test.txt' INTO TABLE test IGNORE 1 LINES;
```

When you use `SELECT ... INTO OUTFILE` in tandem with `LOAD DATA` to write data from a database into a file and then read the file back into the database later, the field- and line-handling options for both statements must match. Otherwise, `LOAD DATA` does not interpret the contents of the file properly. Suppose that you use `SELECT ... INTO OUTFILE` to write a file with fields delimited by commas:

```
SELECT * INTO OUTFILE 'data.txt'
  FIELDS TERMINATED BY ','
  FROM table2;
```

To read the comma-delimited file, the correct statement is:

```
LOAD DATA INFILE 'data.txt' INTO TABLE table2
  FIELDS TERMINATED BY ',';
```

If instead you tried to read the file with the statement shown following, it would not work because it instructs `LOAD DATA` to look for tabs between fields:

```
LOAD DATA INFILE 'data.txt' INTO TABLE table2
  FIELDS TERMINATED BY '\t';
```

The likely result is that each input line would be interpreted as a single field.

`LOAD DATA` can be used to read files obtained from external sources. For example, many programs can export data in comma-separated values (CSV) format, such that lines have fields separated by commas and enclosed within double quotation marks, with an initial line of column names. If the lines in such a file are terminated by carriage return/newline pairs, the statement shown here illustrates the field- and line-handling options you would use to load the file:

```
LOAD DATA INFILE 'data.txt' INTO TABLE tbl_name
  FIELDS TERMINATED BY ',' ENCLOSED BY '"'
  LINES TERMINATED BY '\r\n'
  IGNORE 1 LINES;
```

If the input values are not necessarily enclosed within quotation marks, use `OPTIONALLY` before the `ENCLOSED BY` option.

Any of the field- or line-handling options can specify an empty string (`''`). If not empty, the `FIELDS [OPTIONALLY] ENCLOSED BY` and `FIELDS ESCAPED BY` values must be a single character. The `FIELDS TERMINATED BY`, `LINES STARTING BY`, and `LINES TERMINATED BY` values can be more than one character. For example, to write lines that are terminated by carriage return/linefeed pairs, or to read a file containing such lines, specify a `LINES TERMINATED BY '\r\n'` clause.

To read a file containing jokes that are separated by lines consisting of `%%`, you can do this

```
CREATE TABLE jokes
  (a INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
  joke TEXT NOT NULL);
LOAD DATA INFILE '/tmp/jokes.txt' INTO TABLE jokes
  FIELDS TERMINATED BY ''
  LINES TERMINATED BY '\n%%\n' (joke);
```

`FIELDS [OPTIONALLY] ENCLOSED BY` controls quoting of fields. For output (`SELECT ... INTO OUTFILE`), if you omit the word `OPTIONALLY`, all fields are enclosed by the `ENCLOSED BY` character. An example of such output (using a comma as the field delimiter) is shown here:

```
"1","a string","100.20"
"2","a string containing a , comma","102.20"
"3","a string containing a \" quote","102.20"
"4","a string containing a \", quote and comma","102.20"
```

If you specify `OPTIONALLY`, the `ENCLOSED BY` character is used only to enclose values from columns that have a string data type (such as `CHAR`, `BINARY`, `TEXT`, or `ENUM`):

```
1,"a string",100.20
2,"a string containing a , comma",102.20
3,"a string containing a \" quote",102.20
4,"a string containing a \", quote and comma",102.20
```

Occurrences of the `ENCLOSED BY` character within a field value are escaped by prefixing them with the `ESCAPED BY` character. Also, if you specify an empty `ESCAPED BY` value, it is possible to inadvertently generate output that cannot be read properly by `LOAD DATA`. For example, the preceding output just shown would appear as follows if the escape character is empty. Observe that the second field in the fourth line contains a comma following the quote, which (erroneously) appears to terminate the field:

```
1,"a string",100.20
2,"a string containing a , comma",102.20
```

```
3,"a string containing a " quote",102.20
4,"a string containing a ", quote and comma",102.20
```

For input, the `ENCLOSED BY` character, if present, is stripped from the ends of field values. (This is true regardless of whether `OPTIONALLY` is specified; `OPTIONALLY` has no effect on input interpretation.) Occurrences of the `ENCLOSED BY` character preceded by the `ESCAPED BY` character are interpreted as part of the current field value.

If the field begins with the `ENCLOSED BY` character, instances of that character are recognized as terminating a field value only if followed by the field or line `TERMINATED BY` sequence. To avoid ambiguity, occurrences of the `ENCLOSED BY` character within a field value can be doubled and are interpreted as a single instance of the character. For example, if `ENCLOSED BY '"'` is specified, quotation marks are handled as shown here:

```
"The ""BIG"" boss"  -> The "BIG" boss
The "BIG" boss      -> The "BIG" boss
The ""BIG"" boss    -> The ""BIG"" boss
```

`FIELDS ESCAPED BY` controls how to read or write special characters:

- For input, if the `FIELDS ESCAPED BY` character is not empty, occurrences of that character are stripped and the following character is taken literally as part of a field value. Some two-character sequences that are exceptions, where the first character is the escape character. These sequences are shown in the following table (using `\` for the escape character). The rules for `NULL` handling are described later in this section.

Character	Escape Sequence
`\0`	An ASCII NUL (`X'00'`) character
`\b`	A backspace character
`\n`	A newline (linefeed) character
`\r`	A carriage return character
`\t`	A tab character.
`\Z`	ASCII 26 (Control+Z)
`\N`	NULL

  For more information about `\`-escape syntax, see Section 9.1.1, “String Literals”.

  If the `FIELDS ESCAPED BY` character is empty, escape-sequence interpretation does not occur.

- For output, if the `FIELDS ESCAPED BY` character is not empty, it is used to prefix the following characters on output:

  - The `FIELDS ESCAPED BY` character.
  - The `FIELDS [OPTIONALLY] ENCLOSED BY` character.
  - The first character of the `FIELDS TERMINATED BY` and `LINES TERMINATED BY` values, if the `ENCLOSED BY` character is empty or unspecified.
  - ASCII `0` (what is actually written following the escape character is ASCII `0`, not a zero-valued byte).

  If the `FIELDS ESCAPED BY` character is empty, no characters are escaped and `NULL` is output as `NULL`, not `\N`. It is probably not a good idea to specify an empty escape character, particularly if field values in your data contain any of the characters in the list just given.

In certain cases, field- and line-handling options interact:

- If `LINES TERMINATED BY` is an empty string and `FIELDS TERMINATED BY` is nonempty, lines are also terminated with `FIELDS TERMINATED BY`.

- If the `FIELDS TERMINATED BY` and `FIELDS ENCLOSED BY` values are both empty (`''`), a fixed-row (nondelimited) format is used. With fixed-row format, no delimiters are used between fields (but you can still have a line terminator). Instead, column values are read and written using a field width wide enough to hold all values in the field. For `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, and `BIGINT`, the field widths are 4, 6, 8, 11, and 20, respectively, no matter what the declared display width is.

  `LINES TERMINATED BY` is still used to separate lines. If a line does not contain all fields, the rest of the columns are set to their default values. If you do not have a line terminator, you should set this to `''`. In this case, the text file must contain all fields for each row.

  Fixed-row format also affects handling of `NULL` values, as described later.

  

  Note

  Fixed-size format does not work if you are using a multibyte character set.

Handling of `NULL` values varies according to the `FIELDS` and `LINES` options in use:

- For the default `FIELDS` and `LINES` values, `NULL` is written as a field value of `\N` for output, and a field value of `\N` is read as `NULL` for input (assuming that the `ESCAPED BY` character is `\`).
- If `FIELDS ENCLOSED BY` is not empty, a field containing the literal word `NULL` as its value is read as a `NULL` value. This differs from the word `NULL` enclosed within `FIELDS ENCLOSED BY` characters, which is read as the string `'NULL'`.
- If `FIELDS ESCAPED BY` is empty, `NULL` is written as the word `NULL`.
- With fixed-row format (which is used when `FIELDS TERMINATED BY` and `FIELDS ENCLOSED BY` are both empty), `NULL` is written as an empty string. This causes both `NULL` values and empty strings in the table to be indistinguishable when written to the file because both are written as empty strings. If you need to be able to tell the two apart when reading the file back in, you should not use fixed-row format.

An attempt to load `NULL` into a `NOT NULL` column produces either a warning or an error according to the rules described in Column Value Assignment.

Some cases are not supported by `LOAD DATA`:

- Fixed-size rows (`FIELDS TERMINATED BY` and `FIELDS ENCLOSED BY` both empty) and `BLOB` or `TEXT` columns.
- If you specify one separator that is the same as or a prefix of another, `LOAD DATA` cannot interpret the input properly. For example, the following `FIELDS` clause would cause problems:

  ```
  FIELDS TERMINATED BY '"' ENCLOSED BY '"'
  ```

- If `FIELDS ESCAPED BY` is empty, a field value that contains an occurrence of `FIELDS ENCLOSED BY` or `LINES TERMINATED BY` followed by the `FIELDS TERMINATED BY` value causes `LOAD DATA` to stop reading a field or line too early. This happens because `LOAD DATA` cannot properly determine where the field or line value ends.

Column List Specification

The following example loads all columns of the `persondata` table:

```
LOAD DATA INFILE 'persondata.txt' INTO TABLE persondata;
```

By default, when no column list is provided at the end of the `LOAD DATA` statement, input lines are expected to contain a field for each table column. If you want to load only some of a table's columns, specify a column list:

```
LOAD DATA INFILE 'persondata.txt' INTO TABLE persondata
(col_name_or_user_var [, col_name_or_user_var] ...);
```

You must also specify a column list if the order of the fields in the input file differs from the order of the columns in the table. Otherwise, MySQL cannot tell how to match input fields with table columns.

Input Preprocessing

Each instance of *`col_name_or_user_var`* in `LOAD DATA` syntax is either a column name or a user variable. With user variables, the `SET` clause enables you to perform preprocessing transformations on their values before assigning the result to columns.

User variables in the `SET` clause can be used in several ways. The following example uses the first input column directly for the value of `t1.column1`, and assigns the second input column to a user variable that is subjected to a division operation before being used for the value of `t1.column2`:

```
LOAD DATA INFILE 'file.txt'
  INTO TABLE t1
  (column1, @var1)
  SET column2 = @var1/100;
```

The `SET` clause can be used to supply values not derived from the input file. The following statement sets `column3` to the current date and time:

```
LOAD DATA INFILE 'file.txt'
  INTO TABLE t1
  (column1, column2)
  SET column3 = CURRENT_TIMESTAMP;
```

You can also discard an input value by assigning it to a user variable and not assigning the variable to any table column:

```
LOAD DATA INFILE 'file.txt'
  INTO TABLE t1
  (column1, @dummy, column2, @dummy, column3);
```

Use of the column/variable list and `SET` clause is subject to the following restrictions:

- Assignments in the `SET` clause should have only column names on the left hand side of assignment operators.
- You can use subqueries in the right hand side of `SET` assignments. A subquery that returns a value to be assigned to a column may be a scalar subquery only. Also, you cannot use a subquery to select from the table that is being loaded.
- Lines ignored by an `IGNORE` *`number`* `LINES` clause are not processed for the column/variable list or `SET` clause.
- User variables cannot be used when loading data with fixed-row format because user variables do not have a display width.

Column Value Assignment

To process an input line, `LOAD DATA` splits it into fields and uses the values according to the column/variable list and the `SET` clause, if they are present. Then the resulting row is inserted into the table. If there are `BEFORE INSERT` or `AFTER INSERT` triggers for the table, they are activated before or after inserting the row, respectively.

Interpretation of field values and assignment to table columns depends on these factors:

- The SQL mode (the value of the `sql_mode` system variable). The mode can be nonrestrictive, or restrictive in various ways. For example, strict SQL mode can be enabled, or the mode can include values such as `NO_ZERO_DATE` or `NO_ZERO_IN_DATE`.

- Presence or absence of the `IGNORE` and `LOCAL` modifiers.

Those factors combine to produce restrictive or nonrestrictive data interpretation by `LOAD DATA`:

- Data interpretation is restrictive if the SQL mode is restrictive and neither the `IGNORE` nor the `LOCAL` modifier is specified. Errors terminate the load operation.
- Data interpretation is nonrestrictive if the SQL mode is nonrestrictive or the `IGNORE` or `LOCAL` modifier is specified. (In particular, either modifier if specified *overrides* a restrictive SQL mode when the `REPLACE` modifier is omitted.) Errors become warnings and the load operation continues.

Restrictive data interpretation uses these rules:

- Too many or too few fields results an error.
- Assigning `NULL` (that is, `\N`) to a non-`NULL` column results in an error.
- A value that is out of range for the column data type results in an error.
- Invalid values produce errors. For example, a value such as `'x'` for a numeric column results in an error, not conversion to 0.

By contrast, nonrestrictive data interpretation uses these rules:

- If an input line has too many fields, the extra fields are ignored and the number of warnings is incremented.
- If an input line has too few fields, the columns for which input fields are missing are assigned their default values. Default value assignment is described in Section 11.6, “Data Type Default Values”.
- Assigning `NULL` (that is, `\N`) to a non-`NULL` column results in assignment of the implicit default value for the column data type. Implicit default values are described in Section 11.6, “Data Type Default Values”.
- Invalid values produce warnings rather than errors, and are converted to the “closest” valid value for the column data type. Examples:
  - A value such as `'x'` for a numeric column results in conversion to 0.
  - An out-of-range numeric or temporal value is clipped to the closest endpoint of the range for the column data type.
  - An invalid value for a `DATETIME`, `DATE`, or `TIME` column is inserted as the implicit default value, regardless of the SQL mode `NO_ZERO_DATE` setting. The implicit default is the appropriate “zero” value for the type (`'0000-00-00 00:00:00'`, `'0000-00-00'`, or `'00:00:00'`). See Section 11.2, “Date and Time Data Types”.
- `LOAD DATA` interprets an empty field value differently from a missing field:
  - For string types, the column is set to the empty string.
  - For numeric types, the column is set to `0`.
  - For date and time types, the column is set to the appropriate “zero” value for the type. See Section 11.2, “Date and Time Data Types”.

  These are the same values that result if you assign an empty string explicitly to a string, numeric, or date or time type explicitly in an `INSERT` or `UPDATE` statement.

`TIMESTAMP` columns are set to the current date and time only if there is a `NULL` value for the column (that is, `\N`) and the column is not declared to permit `NULL` values, or if the `TIMESTAMP` column default value is the current timestamp and it is omitted from the field list when a field list is specified.

`LOAD DATA` regards all input as strings, so you cannot use numeric values for `ENUM` or `SET` columns the way you can with `INSERT` statements. All `ENUM` and `SET` values must be specified as strings.

`BIT` values cannot be loaded directly using binary notation (for example, `b'011010'`). To work around this, use the `SET` clause to strip off the leading `b'` and trailing `'` and perform a base-2 to base-10 conversion so that MySQL loads the values into the `BIT` column properly:

```
$> cat /tmp/bit_test.txt
b'10'
b'1111111'
$> mysql test
mysql> LOAD DATA INFILE '/tmp/bit_test.txt'
       INTO TABLE bit_test (@var1)
       SET b = CAST(CONV(MID(@var1, 3, LENGTH(@var1)-3), 2, 10) AS UNSIGNED);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Deleted: 0  Skipped: 0  Warnings: 0

mysql> SELECT BIN(b+0) FROM bit_test;
+----------+
| BIN(b+0) |
+----------+
| 10       |
| 1111111  |
+----------+
2 rows in set (0.00 sec)
```

For `BIT` values in `0b` binary notation (for example, `0b011010`), use this `SET` clause instead to strip off the leading `0b`:

```
SET b = CAST(CONV(MID(@var1, 3, LENGTH(@var1)-2), 2, 10) AS UNSIGNED)
```

Partitioned Table Support

`LOAD DATA` supports explicit partition selection using the `PARTITION` clause with a list of one or more comma-separated names of partitions, subpartitions, or both. When this clause is used, if any rows from the file cannot be inserted into any of the partitions or subpartitions named in the list, the statement fails with the error `Found a row not matching the given partition set`. For more information and examples, see Section 24.5, “Partition Selection”.

Concurrency Considerations

With the `LOW_PRIORITY` modifier, execution of the `LOAD DATA` statement is delayed until no other clients are reading from the table. This affects only storage engines that use only table-level locking (such as `MyISAM`, `MEMORY`, and `MERGE`).

With the `CONCURRENT` modifier and a `MyISAM` table that satisfies the condition for concurrent inserts (that is, it contains no free blocks in the middle), other threads can retrieve data from the table while `LOAD DATA` is executing. This modifier affects the performance of `LOAD DATA` a bit, even if no other thread is using the table at the same time.

Statement Result Information

When the `LOAD DATA` statement finishes, it returns an information string in the following format:

```
Records: 1  Deleted: 0  Skipped: 0  Warnings: 0
```

Warnings occur under the same circumstances as when values are inserted using the `INSERT` statement (see Section 13.2.7, “INSERT Statement”), except that `LOAD DATA` also generates warnings when there are too few or too many fields in the input row.

You can use `SHOW WARNINGS` to get a list of the first `max_error_count` warnings as information about what went wrong. See Section 13.7.7.42, “SHOW WARNINGS Statement”.

If you are using the C API, you can get information about the statement by calling the `mysql_info()` function. See mysql_info().

Replication Considerations

`LOAD DATA` is considered unsafe for statement-based replication. If you use `LOAD DATA` with `binlog_format=STATEMENT`, each replica on which the changes are to be applied creates a temporary file containing the data. This temporary file is not encrypted, even if binary log encryption is active on the source, If encryption is required, use row-based or mixed binary logging format instead, for which replicas do not create the temporary file. For more information on the interaction between `LOAD DATA` and replication, see Section 17.5.1.19, “Replication and LOAD DATA”.

Miscellaneous Topics

On Unix, if you need `LOAD DATA` to read from a pipe, you can use the following technique (the example loads a listing of the `/` directory into the table `db1.t1`):

```
mkfifo /mysql/data/db1/ls.dat
chmod 666 /mysql/data/db1/ls.dat
find / -ls > /mysql/data/db1/ls.dat &
mysql -e "LOAD DATA INFILE 'ls.dat' INTO TABLE t1" db1
```

Here you must run the command that generates the data to be loaded and the `mysql` commands either on separate terminals, or run the data generation process in the background (as shown in the preceding example). If you do not do this, the pipe blocks until data is read by the `mysql` process.

13.2.10 LOAD XML Statement

```
LOAD XML
    [LOW_PRIORITY | CONCURRENT] [LOCAL]
    INFILE 'file_name'
    [REPLACE | IGNORE]
    INTO TABLE [db_name.]tbl_name
    [CHARACTER SET charset_name]
    [ROWS IDENTIFIED BY '<tagname>']
    [IGNORE number {LINES | ROWS}]
    [(field_name_or_user_var
        [, field_name_or_user_var] ...)]
    [SET col_name={expr | DEFAULT}
        [, col_name={expr | DEFAULT}] ...]
```

The `LOAD XML` statement reads data from an XML file into a table. The *`file_name`* must be given as a literal string. The *`tagname`* in the optional `ROWS IDENTIFIED BY` clause must also be given as a literal string, and must be surrounded by angle brackets (`<` and `>`).

`LOAD XML` acts as the complement of running the `mysql` client in XML output mode (that is, starting the client with the `--xml` option). To write data from a table to an XML file, you can invoke the `mysql` client with the `--xml` and `-e` options from the system shell, as shown here:

```
$> mysql --xml -e 'SELECT * FROM mydb.mytable' > file.xml
```

To read the file back into a table, use `LOAD XML`. By default, the `<row>` element is considered to be the equivalent of a database table row; this can be changed using the `ROWS IDENTIFIED BY` clause.

This statement supports three different XML formats:

- Column names as attributes and column values as attribute values:

  ```
  <row column1="value1" column2="value2" .../>
  ```

- Column names as tags and column values as the content of these tags:

  ```
  <row>
    <column1>value1</column1>
    <column2>value2</column2>
  </row>
  ```

- Column names are the `name` attributes of `<field>` tags, and values are the contents of these tags:

```
<row>
  <field name='column1'>value1</field>
  <field name='column2'>value2</field>
</row>
```

This is the format used by other MySQL tools, such as `mysqldump`.

All three formats can be used in the same XML file; the import routine automatically detects the format for each row and interprets it correctly. Tags are matched based on the tag or attribute name and the column name.

Prior to MySQL 8.0.21, `LOAD XML` did not support `CDATA` sections in the source XML. (Bug #30753708, Bug #98199)

The following clauses work essentially the same way for `LOAD XML` as they do for `LOAD DATA`:

- `LOW_PRIORITY` or `CONCURRENT`
- `LOCAL`
- `REPLACE` or `IGNORE`
- `CHARACTER SET`
- `SET`

See Section 13.2.9, “LOAD DATA Statement”, for more information about these clauses.

`(field_name_or_user_var, ...)` is a list of one or more comma-separated XML fields or user variables. The name of a user variable used for this purpose must match the name of a field from the XML file, prefixed with `@`. You can use field names to select only desired fields. User variables can be employed to store the corresponding field values for subsequent re-use.

The `IGNORE` *`number`* `LINES` or `IGNORE` *`number`* `ROWS` clause causes the first *`number`* rows in the XML file to be skipped. It is analogous to the `LOAD DATA` statement's `IGNORE ... LINES` clause.

Suppose that we have a table named `person`, created as shown here:

```
USE test;

CREATE TABLE person (
    person_id INT NOT NULL PRIMARY KEY,
    fname VARCHAR(40) NULL,
    lname VARCHAR(40) NULL,
    created TIMESTAMP
);
```

Suppose further that this table is initially empty.

Now suppose that we have a simple XML file `person.xml`, whose contents are as shown here:

```
<list>
  <person person_id="1" fname="Kapek" lname="Sainnouine"/>
  <person person_id="2" fname="Sajon" lname="Rondela"/>
  <person person_id="3"><fname>Likame</fname><lname>Örrtmons</lname></person>
  <person person_id="4"><fname>Slar</fname><lname>Manlanth</lname></person>
  <person><field name="person_id">5</field><field name="fname">Stoma</field>
    <field name="lname">Milu</field></person>
  <person><field name="person_id">6</field><field name="fname">Nirtam</field>
    <field name="lname">Sklödd</field></person>
  <person person_id="7"><fname>Sungam</fname><lname>Dulbåd</lname></person>
  <person person_id="8" fname="Sraref" lname="Encmelt"/>
</list>
```

Each of the permissible XML formats discussed previously is represented in this example file.

To import the data in `person.xml` into the `person` table, you can use this statement:

```
mysql> LOAD XML LOCAL INFILE 'person.xml'
    ->   INTO TABLE person
    ->   ROWS IDENTIFIED BY '<person>';

Query OK, 8 rows affected (0.00 sec)
Records: 8  Deleted: 0  Skipped: 0  Warnings: 0
```

Here, we assume that `person.xml` is located in the MySQL data directory. If the file cannot be found, the following error results:

```
ERROR 2 (HY000): File '/person.xml' not found (Errcode: 2)
```

The `ROWS IDENTIFIED BY '<person>'` clause means that each `<person>` element in the XML file is considered equivalent to a row in the table into which the data is to be imported. In this case, this is the `person` table in the `test` database.

As can be seen by the response from the server, 8 rows were imported into the `test.person` table. This can be verified by a simple `SELECT` statement:

```
mysql> SELECT * FROM person;
+-----------+--------+------------+---------------------+
| person_id | fname  | lname      | created             |
+-----------+--------+------------+---------------------+
|         1 | Kapek  | Sainnouine | 2007-07-13 16:18:47 |
|         2 | Sajon  | Rondela    | 2007-07-13 16:18:47 |
|         3 | Likame | Örrtmons   | 2007-07-13 16:18:47 |
|         4 | Slar   | Manlanth   | 2007-07-13 16:18:47 |
|         5 | Stoma  | Nilu       | 2007-07-13 16:18:47 |
|         6 | Nirtam | Sklöd      | 2007-07-13 16:18:47 |
|         7 | Sungam | Dulbåd     | 2007-07-13 16:18:47 |
|         8 | Sreraf | Encmelt    | 2007-07-13 16:18:47 |
+-----------+--------+------------+---------------------+
8 rows in set (0.00 sec)
```

This shows, as stated earlier in this section, that any or all of the 3 permitted XML formats may appear in a single file and be read using `LOAD XML`.

The inverse of the import operation just shown—that is, dumping MySQL table data into an XML file—can be accomplished using the `mysql` client from the system shell, as shown here:

```
$> mysql --xml -e "SELECT * FROM test.person" > person-dump.xml
$> cat person-dump.xml
<?xml version="1.0"?>

<resultset statement="SELECT * FROM test.person" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <row>
 <field name="person_id">1</field>
 <field name="fname">Kapek</field>
 <field name="lname">Sainnouine</field>
  </row>

  <row>
 <field name="person_id">2</field>
 <field name="fname">Sajon</field>
 <field name="lname">Rondela</field>
  </row>

  <row>
 <field name="person_id">3</field>
 <field name="fname">Likema</field>
 <field name="lname">Örrtmons</field>
  </row>

  <row>
 <field name="person_id">4</field>
 <field name="fname">Slar</field>
 <field name="lname">Manlanth</field>
```

```
  </row>

  <row>
 <field name="person_id">5</field>
 <field name="fname">Stoma</field>
 <field name="lname">Nilu</field>
  </row>

  <row>
 <field name="person_id">6</field>
 <field name="fname">Nirtam</field>
 <field name="lname">Sklöd</field>
  </row>

  <row>
 <field name="person_id">7</field>
 <field name="fname">Sungam</field>
 <field name="lname">Dulbåd</field>
  </row>

  <row>
 <field name="person_id">8</field>
 <field name="fname">Sreraf</field>
 <field name="lname">Encmelt</field>
  </row>
</resultset>
```

Note

The `--xml` option causes the `mysql` client to use XML formatting for its output; the `-e` option causes the client to execute the SQL statement immediately following the option. See Section 4.5.1, “mysql — The MySQL Command-Line Client”.

You can verify that the dump is valid by creating a copy of the `person` table and importing the dump file into the new table, like this:

```
mysql> USE test;
mysql> CREATE TABLE person2 LIKE person;
Query OK, 0 rows affected (0.00 sec)

mysql> LOAD XML LOCAL INFILE 'person-dump.xml'
    ->   INTO TABLE person2;
Query OK, 8 rows affected (0.01 sec)
Records: 8  Deleted: 0  Skipped: 0  Warnings: 0

mysql> SELECT * FROM person2;
+-----------+--------+------------+---------------------+
| person_id | fname  | lname      | created             |
+-----------+--------+------------+---------------------+
|         1 | Kapek  | Sainnouine | 2007-07-13 16:18:47 |
|         2 | Sajon  | Rondela    | 2007-07-13 16:18:47 |
|         3 | Likema | Örrtmons   | 2007-07-13 16:18:47 |
|         4 | Slar   | Manlanth   | 2007-07-13 16:18:47 |
|         5 | Stoma  | Nilu       | 2007-07-13 16:18:47 |
|         6 | Nirtam | Sklöd      | 2007-07-13 16:18:47 |
|         7 | Sungam | Dulbåd     | 2007-07-13 16:18:47 |
|         8 | Sreraf | Encmelt    | 2007-07-13 16:18:47 |
+-----------+--------+------------+---------------------+
8 rows in set (0.00 sec)
```

There is no requirement that every field in the XML file be matched with a column in the corresponding table. Fields which have no corresponding columns are skipped. You can see this by first emptying the `person2` table and dropping the `created` column, then using the same `LOAD XML` statement we just employed previously, like this:

```
mysql> TRUNCATE person2;
Query OK, 8 rows affected (0.26 sec)

mysql> ALTER TABLE person2 DROP COLUMN created;
```

```
Query OK, 0 rows affected (0.52 sec)
Records: 0  Duplicates: 0  Warnings: 0

mysql> SHOW CREATE TABLE person2\G
*************************** 1. row ***************************
       Table: person2
Create Table: CREATE TABLE `person2` (
  `person_id` int NOT NULL,
  `fname` varchar(40) DEFAULT NULL,
  `lname` varchar(40) DEFAULT NULL,
  PRIMARY KEY (`person_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

mysql> LOAD XML LOCAL INFILE 'person-dump.xml'
    ->   INTO TABLE person2;
Query OK, 8 rows affected (0.01 sec)
Records: 8  Deleted: 0  Skipped: 0  Warnings: 0

mysql> SELECT * FROM person2;
+-----------+--------+------------+
| person_id | fname  | lname      |
+-----------+--------+------------+
|         1 | Kapek  | Sainnouine |
|         2 | Sajon  | Rondela    |
|         3 | Likema | Örrtmons   |
|         4 | Slar   | Manlanth   |
|         5 | Stoma  | Nilu       |
|         6 | Nirtam | Sklöd      |
|         7 | Sungam | Dulbåd     |
|         8 | Sreraf | Encmelt    |
+-----------+--------+------------+
8 rows in set (0.00 sec)
```

The order in which the fields are given within each row of the XML file does not affect the operation of `LOAD XML`; the field order can vary from row to row, and is not required to be in the same order as the corresponding columns in the table.

As mentioned previously, you can use a `(field_name_or_user_var, ...)` list of one or more XML fields (to select desired fields only) or user variables (to store the corresponding field values for later use). User variables can be especially useful when you want to insert data from an XML file into table columns whose names do not match those of the XML fields. To see how this works, we first create a table named `individual` whose structure matches that of the `person` table, but whose columns are named differently:

```
mysql> CREATE TABLE individual (
    ->     individual_id INT NOT NULL PRIMARY KEY,
    ->     name1 VARCHAR(40) NULL,
    ->     name2 VARCHAR(40) NULL,
    ->     made TIMESTAMP
    -> );
Query OK, 0 rows affected (0.42 sec)
```

In this case, you cannot simply load the XML file directly into the table, because the field and column names do not match:

```
mysql> LOAD XML INFILE '../bin/person-dump.xml' INTO TABLE test.individual;
ERROR 1263 (22004): Column set to default value; NULL supplied to NOT NULL column 'individual_id' at ro
```

This happens because the MySQL server looks for field names matching the column names of the target table. You can work around this problem by selecting the field values into user variables, then setting the target table's columns equal to the values of those variables using `SET`. You can perform both of these operations in a single statement, as shown here:

```
mysql> LOAD XML INFILE '../bin/person-dump.xml'
    ->     INTO TABLE test.individual (@person_id, @fname, @lname, @created)
    ->     SET individual_id=@person_id, name1=@fname, name2=@lname, made=@created;
Query OK, 8 rows affected (0.05 sec)
Records: 8  Deleted: 0  Skipped: 0  Warnings: 0
```

```
mysql> SELECT * FROM individual;
+---------------+--------+------------+---------------------+
| individual_id | name1  | name2      | made                |
+---------------+--------+------------+---------------------+
|             1 | Kapek  | Sainnouine | 2007-07-13 16:18:47 |
|             2 | Sajon  | Rondela    | 2007-07-13 16:18:47 |
|             3 | Likema | Örrtmons   | 2007-07-13 16:18:47 |
|             4 | Slar   | Manlanth   | 2007-07-13 16:18:47 |
|             5 | Stoma  | Nilu       | 2007-07-13 16:18:47 |
|             6 | Nirtam | Sklöd      | 2007-07-13 16:18:47 |
|             7 | Sungam | Dulbåd     | 2007-07-13 16:18:47 |
|             8 | Srraf  | Encmelt    | 2007-07-13 16:18:47 |
+---------------+--------+------------+---------------------+
8 rows in set (0.00 sec)
```

The names of the user variables *must* match those of the corresponding fields from the XML file, with the addition of the required `@` prefix to indicate that they are variables. The user variables need not be listed or assigned in the same order as the corresponding fields.

Using a `ROWS IDENTIFIED BY '<tagname>'` clause, it is possible to import data from the same XML file into database tables with different definitions. For this example, suppose that you have a file named `address.xml` which contains the following XML:

```
<?xml version="1.0"?>

<list>
  <person person_id="1">
    <fname>Robert</fname>
    <lname>Jones</lname>
    <address address_id="1" street="Mill Creek Road" zip="45365" city="Sidney"/>
    <address address_id="2" street="Main Street" zip="28681" city="Taylorsville"/>
  </person>

  <person person_id="2">
    <fname>Mary</fname>
    <lname>Smith</lname>
    <address address_id="3" street="River Road" zip="80239" city="Denver"/>
    <!-- <address address_id="4" street="North Street" zip="37920" city="Knoxville"/> -->
  </person>

</list>
```

You can again use the `test.person` table as defined previously in this section, after clearing all the existing records from the table and then showing its structure as shown here:

```
mysql< TRUNCATE person;
Query OK, 0 rows affected (0.04 sec)

mysql< SHOW CREATE TABLE person\G
*************************** 1. row ***************************
       Table: person
Create Table: CREATE TABLE `person` (
  `person_id` int(11) NOT NULL,
  `fname` varchar(40) DEFAULT NULL,
  `lname` varchar(40) DEFAULT NULL,
  `created` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`person_id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4
1 row in set (0.00 sec)
```

Now create an `address` table in the `test` database using the following `CREATE TABLE` statement:

```
CREATE TABLE address (
    address_id INT NOT NULL PRIMARY KEY,
    person_id INT NULL,
    street VARCHAR(40) NULL,
    zip INT NULL,
    city VARCHAR(40) NULL,
    created TIMESTAMP
```

```
);
```

To import the data from the XML file into the `person` table, execute the following `LOAD XML` statement, which specifies that rows are to be specified by the `<person>` element, as shown here;

```
mysql> LOAD XML LOCAL INFILE 'address.xml'
    ->   INTO TABLE person
    ->   ROWS IDENTIFIED BY '<person>';
Query OK, 2 rows affected (0.00 sec)
Records: 2  Deleted: 0  Skipped: 0  Warnings: 0
```

You can verify that the records were imported using a `SELECT` statement:

```
mysql> SELECT * FROM person;
+-----------+--------+-------+---------------------+
| person_id | fname  | lname | created             |
+-----------+--------+-------+---------------------+
|         1 | Robert | Jones | 2007-07-24 17:37:06 |
|         2 | Mary   | Smith | 2007-07-24 17:37:06 |
+-----------+--------+-------+---------------------+
2 rows in set (0.00 sec)
```

Since the `<address>` elements in the XML file have no corresponding columns in the `person` table, they are skipped.

To import the data from the `<address>` elements into the `address` table, use the `LOAD XML` statement shown here:

```
mysql> LOAD XML LOCAL INFILE 'address.xml'
    ->   INTO TABLE address
    ->   ROWS IDENTIFIED BY '<address>';
Query OK, 3 rows affected (0.00 sec)
Records: 3  Deleted: 0  Skipped: 0  Warnings: 0
```

You can see that the data was imported using a `SELECT` statement such as this one:

```
mysql> SELECT * FROM address;
+------------+-----------+-----------------+-------+--------------+---------------------+
| address_id | person_id | street          | zip   | city         | created             |
+------------+-----------+-----------------+-------+--------------+---------------------+
|          1 |         1 | Mill Creek Road | 45365 | Sidney       | 2007-07-24 17:37:37 |
|          2 |         1 | Main Street     | 28681 | Taylorsville | 2007-07-24 17:37:37 |
|          3 |         2 | River Road      | 80239 | Denver       | 2007-07-24 17:37:37 |
+------------+-----------+-----------------+-------+--------------+---------------------+
3 rows in set (0.00 sec)
```

The data from the `<address>` element that is enclosed in XML comments is not imported. However, since there is a `person_id` column in the `address` table, the value of the `person_id` attribute from the parent `<person>` element for each `<address>` *is* imported into the `address` table.

Security Considerations. As with the `LOAD DATA` statement, the transfer of the XML file from the client host to the server host is initiated by the MySQL server. In theory, a patched server could be built that would tell the client program to transfer a file of the server's choosing rather than the file named by the client in the `LOAD XML` statement. Such a server could access any file on the client host to which the client user has read access.

In a Web environment, clients usually connect to MySQL from a Web server. A user that can run any command against the MySQL server can use `LOAD XML LOCAL` to read any files to which the Web server process has read access. In this environment, the client with respect to the MySQL server is actually the Web server, not the remote program being run by the user who connects to the Web server.

You can disable loading of XML files from clients by starting the server with `--local-infile=0` or `--local-infile=OFF`. This option can also be used when starting the `mysql` client to disable `LOAD XML` for the duration of the client session.

To prevent a client from loading XML files from the server, do not grant the `FILE` privilege to the corresponding MySQL user account, or revoke this privilege if the client user account already has it.

Important

Revoking the `FILE` privilege (or not granting it in the first place) keeps the user only from executing the `LOAD XML` statement (as well as the `LOAD_FILE()` function; it does *not* prevent the user from executing `LOAD XML LOCAL`. To disallow this statement, you must start the server or the client with `--local-infile=OFF`.

In other words, the `FILE` privilege affects only whether the client can read files on the server; it has no bearing on whether the client can read files on the local file system.

13.2.11 Parenthesized Query Expressions

```
parenthesized_query_expression:
    ( query_expression [order_by_clause] [limit_clause] )
      [order_by_clause]
      [limit_clause]
      [into_clause]

query_expression:
    query_block [set_op query_block [set_op query_block ...]]
      [order_by_clause]
      [limit_clause]
      [into_clause]

query_block:
    SELECT ... | TABLE | VALUES

order_by_clause:
    ORDER BY as for SELECT

limit_clause:
    LIMIT as for SELECT

into_clause:
    INTO as for SELECT

set_op:
    UNION | INTERSECT | EXCEPT
```

MySQL 8.0.22 and higher supports parenthesized query expressions according to the preceding syntax. At its simplest, a parenthesized query expression contains a single `SELECT` or other statement returning a result set and no following optional clauses:

```
(SELECT 1);
(SELECT * FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'mysql');

TABLE t;

VALUES ROW(2, 3, 4), ROW(1, -2, 3);
```

(Support for the `TABLE` and `VALUES` statements is available beginning with MySQL 8.0.19.)

A parenthesized query expression can also contain queries linked by one or more set operations such as `UNION`, and end with any or all of the optional clauses:

```
mysql> (SELECT 1 AS result UNION SELECT 2);
+--------+
| result |
+--------+
|      1 |
|      2 |
+--------+
```

```
mysql> (SELECT 1 AS result UNION SELECT 2) LIMIT 1;
+--------+
| result |
+--------+
|      1 |
+--------+
mysql> (SELECT 1 AS result UNION SELECT 2) LIMIT 1 OFFSET 1;
+--------+
| result |
+--------+
|      2 |
+--------+
mysql> (SELECT 1 AS result UNION SELECT 2)
       ORDER BY result DESC LIMIT 1;
+--------+
| result |
+--------+
|      2 |
+--------+
mysql> (SELECT 1 AS result UNION SELECT 2)
       ORDER BY result DESC LIMIT 1 OFFSET 1;
+--------+
| result |
+--------+
|      1 |
+--------+
mysql> (SELECT 1 AS result UNION SELECT 3 UNION SELECT 2)
       ORDER BY result LIMIT 1 OFFSET 1 INTO @var;
mysql> SELECT @var;
+------+
| @var |
+------+
|    2 |
+------+
```

In addition to `UNION`, the `INTERSECT` and `EXCEPT` set operators are available beginning with MySQL 8.0.31. `INTERSECT` acts before `UNION` and `EXCEPT`, so that the following two statements are equivalent:

```
SELECT a FROM t1 EXCEPT SELECT b FROM t2 INTERSECT SELECT c FROM t3;

SELECT a FROM t1 EXCEPT (SELECT b FROM t2 INTERSECT SELECT c FROM t3);
```

Parenthesized query expressions are also used as query expressions, so a query expression, usually composed of query blocks, may also consist of parenthesized query expressions:

```
(TABLE t1 ORDER BY a) UNION (TABLE t2 ORDER BY b) ORDER BY z;
```

Query blocks may have trailing `ORDER BY` and `LIMIT` clauses, which are applied before the outer set operation, `ORDER BY`, and `LIMIT`.

You cannot have a query block with a trailing `ORDER BY` or `LIMIT` without wrapping it in parentheses but parentheses may be used for enforcement in various ways:

- To enforce `LIMIT` on each query block:

  ```
  (SELECT 1 LIMIT 1) UNION (VALUES ROW(2) LIMIT 1);

  (VALUES ROW(1), ROW(2) LIMIT 2) EXCEPT (SELECT 2 LIMIT 1);
  ```

- To enforce `LIMIT` on both query blocks and the entire query expression:

  ```
  (SELECT 1 LIMIT 1) UNION (SELECT 2 LIMIT 1) LIMIT 1;
  ```

- To enforce `LIMIT` on the entire query expression (with no parentheses):

  ```
  VALUES ROW(1), ROW(2) INTERSECT VALUES ROW(2), ROW(1) LIMIT 1;
  ```

- Hybrid enforcement: `LIMIT` on the first query block and on the entire query expression:

```
(SELECT 1 LIMIT 1) UNION SELECT 2 LIMIT 1;
```

The syntax described in this section is subject to certain restrictions:

- A trailing `INTO` clause for a query expression is not permitted if there is another `INTO` clause inside parentheses.
- Prior to MySQL 8.0.31, when `ORDER BY` or `LIMIT` occurred within a parenthesized query expression and was also applied in the outer query, the result was undefined. This is not an issue in MySQL 8.0.31 and later, where this is handled in accordance with the SQL standard.

  Prior to MySQL 8.0.31, parenthesized query expressions did not permit multiple levels of `ORDER BY` or `LIMIT` operations, and statements containing these were rejected with `ER_NOT_SUPPORTED_YET`. In MySQL 8.0.31 and later, this restriction is lifted, and nested parenthesized query expressions are permitted. The maximum level of nesting supported is 63; this is after any simplifications or merges have been performed by the parser.

  An example of such a statement is shown here:

```
mysql> (SELECT 'a' UNION SELECT 'b' LIMIT 2) LIMIT 3;
+---+
| a |
+---+
| a |
| b |
+---+
2 rows in set (0.00 sec)
```

  You should be aware that, in MySQL 8.0.31 and later, when collapsing parenthesized expression bodies, MySQL follows SQL standard semantics, so that a higher outer limit cannot override an inner lower one. For example, `(SELECT ... LIMIT 5) LIMIT 10` can return no more than five rows.

13.2.12 REPLACE Statement

```
REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    { {VALUES | VALUE} (value_list) [, (value_list)] ...
      |
      VALUES row_constructor_list
    }

REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    SET assignment_list

REPLACE [LOW_PRIORITY | DELAYED]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    {SELECT ... | TABLE table_name}

value:
    {expr | DEFAULT}

value_list:
    value [, value] ...

row_constructor_list:
    ROW(value_list)[, ROW(value_list)][, ...]

assignment:
    col_name = value

assignment_list:
```

```
    assignment [, assignment] ...
```

`REPLACE` works exactly like `INSERT`, except that if an old row in the table has the same value as a new row for a `PRIMARY KEY` or a `UNIQUE` index, the old row is deleted before the new row is inserted. See Section 13.2.7, “INSERT Statement”.

`REPLACE` is a MySQL extension to the SQL standard. It either inserts, or *deletes* and inserts. For another MySQL extension to standard SQL—that either inserts or *updates*—see Section 13.2.7.2, “INSERT ... ON DUPLICATE KEY UPDATE Statement”.

`DELAYED` inserts and replaces were deprecated in MySQL 5.6. In MySQL 8.0, `DELAYED` is not supported. The server recognizes but ignores the `DELAYED` keyword, handles the replace as a nondelayed replace, and generates an `ER_WARN_LEGACY_SYNTAX_CONVERTED` warning: `REPLACE DELAYED is no longer supported. The statement was converted to REPLACE`. The `DELAYED` keyword is scheduled for removal in a future release. release.

Note

`REPLACE` makes sense only if a table has a `PRIMARY KEY` or `UNIQUE` index. Otherwise, it becomes equivalent to `INSERT`, because there is no index to be used to determine whether a new row duplicates another.

Values for all columns are taken from the values specified in the `REPLACE` statement. Any missing columns are set to their default values, just as happens for `INSERT`. You cannot refer to values from the current row and use them in the new row. If you use an assignment such as `SET col_name = col_name + 1`, the reference to the column name on the right hand side is treated as `DEFAULT(col_name)`, so the assignment is equivalent to `SET col_name = DEFAULT(col_name) + 1`.

In MySQL 8.0.19 and later, you can specify the column values that `REPLACE` attempts to insert using `VALUES ROW()`.

To use `REPLACE`, you must have both the `INSERT` and `DELETE` privileges for the table.

If a generated column is replaced explicitly, the only permitted value is `DEFAULT`. For information about generated columns, see Section 13.1.20.8, “CREATE TABLE and Generated Columns”.

`REPLACE` supports explicit partition selection using the `PARTITION` clause with a list of comma-separated names of partitions, subpartitions, or both. As with `INSERT`, if it is not possible to insert the new row into any of these partitions or subpartitions, the `REPLACE` statement fails with the error `Found a row not matching the given partition set`. For more information and examples, see Section 24.5, “Partition Selection”.

The `REPLACE` statement returns a count to indicate the number of rows affected. This is the sum of the rows deleted and inserted. If the count is 1 for a single-row `REPLACE`, a row was inserted and no rows were deleted. If the count is greater than 1, one or more old rows were deleted before the new row was inserted. It is possible for a single row to replace more than one old row if the table contains multiple unique indexes and the new row duplicates values for different old rows in different unique indexes.

The affected-rows count makes it easy to determine whether `REPLACE` only added a row or whether it also replaced any rows: Check whether the count is 1 (added) or greater (replaced).

If you are using the C API, the affected-rows count can be obtained using the `mysql_affected_rows()` function.

You cannot replace into a table and select from the same table in a subquery.

MySQL uses the following algorithm for `REPLACE` (and `LOAD DATA ... REPLACE`):

1. Try to insert the new row into the table
2. While the insertion fails because a duplicate-key error occurs for a primary key or unique index:

a. Delete from the table the conflicting row that has the duplicate key value

b. Try again to insert the new row into the table

It is possible that in the case of a duplicate-key error, a storage engine may perform the `REPLACE` as an update rather than a delete plus insert, but the semantics are the same. There are no user-visible effects other than a possible difference in how the storage engine increments `Handler_xxx` status variables.

Because the results of `REPLACE ... SELECT` statements depend on the ordering of rows from the `SELECT` and this order cannot always be guaranteed, it is possible when logging these statements for the source and the replica to diverge. For this reason, `REPLACE ... SELECT` statements are flagged as unsafe for statement-based replication. such statements produce a warning in the error log when using statement-based mode and are written to the binary log using the row-based format when using `MIXED` mode. See also Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

MySQL 8.0.19 and later supports `TABLE` as well as `SELECT` with `REPLACE`, just as it does with `INSERT`. See Section 13.2.7.1, “INSERT ... SELECT Statement”, for more information and examples.

When modifying an existing table that is not partitioned to accommodate partitioning, or, when modifying the partitioning of an already partitioned table, you may consider altering the table's primary key (see Section 24.6.1, “Partitioning Keys, Primary Keys, and Unique Keys”). You should be aware that, if you do this, the results of `REPLACE` statements may be affected, just as they would be if you modified the primary key of a nonpartitioned table. Consider the table created by the following `CREATE TABLE` statement:

```
CREATE TABLE test (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  data VARCHAR(64) DEFAULT NULL,
  ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (id)
);
```

When we create this table and run the statements shown in the mysql client, the result is as follows:

```
mysql> REPLACE INTO test VALUES (1, 'Old', '2014-08-20 18:47:00');
Query OK, 1 row affected (0.04 sec)

mysql> REPLACE INTO test VALUES (1, 'New', '2014-08-20 18:47:42');
Query OK, 2 rows affected (0.04 sec)

mysql> SELECT * FROM test;
+----+------+---------------------+
| id | data | ts                  |
+----+------+---------------------+
|  1 | New  | 2014-08-20 18:47:42 |
+----+------+---------------------+
1 row in set (0.00 sec)
```

Now we create a second table almost identical to the first, except that the primary key now covers 2 columns, as shown here (emphasized text):

```
CREATE TABLE test2 (
  id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  data VARCHAR(64) DEFAULT NULL,
  ts TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (id, ts)
);
```

When we run on `test2` the same two `REPLACE` statements as we did on the original `test` table, we obtain a different result:

```
mysql> REPLACE INTO test2 VALUES (1, 'Old', '2014-08-20 18:47:00');
Query OK, 1 row affected (0.05 sec)
```

```
mysql> REPLACE INTO test2 VALUES (1, 'New', '2014-08-20 18:47:42');
Query OK, 1 row affected (0.06 sec)

mysql> SELECT * FROM test2;
+----+------+---------------------+
| id | data | ts                  |
+----+------+---------------------+
|  1 | Old  | 2014-08-20 18:47:00 |
|  1 | New  | 2014-08-20 18:47:42 |
+----+------+---------------------+
2 rows in set (0.00 sec)
```

This is due to the fact that, when run on `test2`, both the `id` and `ts` column values must match those of an existing row for the row to be replaced; otherwise, a row is inserted.

13.2.13 SELECT Statement

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
    [HIGH_PRIORITY]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr] ...
    [into_option]
    [FROM table_references
      [PARTITION partition_list]]
    [WHERE where_condition]
    [GROUP BY {col_name | expr | position}, ... [WITH ROLLUP]]
    [HAVING where_condition]
    [WINDOW window_name AS (window_spec)
        [, window_name AS (window_spec)] ...]
    [ORDER BY {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [into_option]
    [FOR {UPDATE | SHARE}
        [OF tbl_name [, tbl_name] ...]
        [NOWAIT | SKIP LOCKED]
      | LOCK IN SHARE MODE]
    [into_option]

into_option: {
    INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
  | INTO DUMPFILE 'file_name'
  | INTO var_name [, var_name] ...
}
```

`SELECT` is used to retrieve rows selected from one or more tables, and can include `UNION` operations and subqueries. Beginning with MySQL 8.0.31, `INTERSECT` and `EXCEPT` operations are also supported. The `UNION`, `INTERSECT`, and `EXCEPT` operators are described in more detail later in this section. See also Section 13.2.15, “Subqueries”.

A `SELECT` statement can start with a `WITH` clause to define common table expressions accessible within the `SELECT`. See Section 13.2.20, “WITH (Common Table Expressions)”.

The most commonly used clauses of `SELECT` statements are these:

- Each *`select_expr`* indicates a column that you want to retrieve. There must be at least one *`select_expr`*.
- *`table_references`* indicates the table or tables from which to retrieve rows. Its syntax is described in Section 13.2.13.2, “JOIN Clause”.
- `SELECT` supports explicit partition selection using the `PARTITION` clause with a list of partitions or subpartitions (or both) following the name of the table in a *`table_reference`* (see

Section 13.2.13.2, “JOIN Clause”). In this case, rows are selected only from the partitions listed, and any other partitions of the table are ignored. For more information and examples, see Section 24.5, “Partition Selection”.

- The `WHERE` clause, if given, indicates the condition or conditions that rows must satisfy to be selected. *`where_condition`* is an expression that evaluates to true for each row to be selected. The statement selects all rows if there is no `WHERE` clause.

  In the `WHERE` expression, you can use any of the functions and operators that MySQL supports, except for aggregate (group) functions. See Section 9.5, “Expressions”, and Chapter 12, *Functions and Operators*.

`SELECT` can also be used to retrieve rows computed without reference to any table.

For example:

```
mysql> SELECT 1 + 1;
        -> 2
```

You are permitted to specify `DUAL` as a dummy table name in situations where no tables are referenced:

```
mysql> SELECT 1 + 1 FROM DUAL;
        -> 2
```

`DUAL` is purely for the convenience of people who require that all `SELECT` statements should have `FROM` and possibly other clauses. MySQL may ignore the clauses. MySQL does not require `FROM DUAL` if no tables are referenced.

In general, clauses used must be given in exactly the order shown in the syntax description. For example, a `HAVING` clause must come after any `GROUP BY` clause and before any `ORDER BY` clause. The `INTO` clause, if present, can appear in any position indicated by the syntax description, but within a given statement can appear only once, not in multiple positions. For more information about `INTO`, see Section 13.2.13.1, “SELECT ... INTO Statement”.

The list of *`select_expr`* terms comprises the select list that indicates which columns to retrieve. Terms specify a column or expression or can use `*`-shorthand:

- A select list consisting only of a single unqualified `*` can be used as shorthand to select all columns from all tables:

  ```
  SELECT * FROM t1 INNER JOIN t2 ...
  ```

- *`tbl_name`*`.*` can be used as a qualified shorthand to select all columns from the named table:

  ```
  SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ...
  ```

- If a table has invisible columns, `*` and *`tbl_name`*`.*` do not include them. To be included, invisible columns must be referenced explicitly.

- Use of an unqualified `*` with other items in the select list may produce a parse error. For example:

  ```
  SELECT id, * FROM t1
  ```

  To avoid this problem, use a qualified *`tbl_name`*`.*` reference:

  ```
  SELECT id, t1.* FROM t1
  ```

  Use qualified *`tbl_name`*`.*` references for each table in the select list:

  ```
  SELECT AVG(score), t1.* FROM t1 ...
  ```

The following list provides additional information about other `SELECT` clauses:

- A *`select_expr`* can be given an alias using `AS` *`alias_name`*. The alias is used as the expression's column name and can be used in `GROUP BY`, `ORDER BY`, or `HAVING` clauses. For example:

```
SELECT CONCAT(last_name,', ',first_name) AS full_name
  FROM mytable ORDER BY full_name;
```

  The `AS` keyword is optional when aliasing a *`select_expr`* with an identifier. The preceding example could have been written like this:

```
SELECT CONCAT(last_name,', ',first_name) full_name
  FROM mytable ORDER BY full_name;
```

  However, because the `AS` is optional, a subtle problem can occur if you forget the comma between two *`select_expr`* expressions: MySQL interprets the second as an alias name. For example, in the following statement, `columnb` is treated as an alias name:

```
SELECT columna columnb FROM mytable;
```

  For this reason, it is good practice to be in the habit of using `AS` explicitly when specifying column aliases.

  It is not permissible to refer to a column alias in a `WHERE` clause, because the column value might not yet be determined when the `WHERE` clause is executed. See Section B.3.4.4, “Problems with Column Aliases”.

- The `FROM` *`table_references`* clause indicates the table or tables from which to retrieve rows. If you name more than one table, you are performing a join. For information on join syntax, see Section 13.2.13.2, “JOIN Clause”. For each table specified, you can optionally specify an alias.

```
tbl_name [[AS] alias] [index_hint]
```

  The use of index hints provides the optimizer with information about how to choose indexes during query processing. For a description of the syntax for specifying these hints, see Section 8.9.4, “Index Hints”.

  You can use `SET max_seeks_for_key=`*`value`* as an alternative way to force MySQL to prefer key scans instead of table scans. See Section 5.1.8, “Server System Variables”.

- You can refer to a table within the default database as *`tbl_name`*, or as *`db_name`*.*`tbl_name`* to specify a database explicitly. You can refer to a column as *`col_name`*, *`tbl_name`*.*`col_name`*, or *`db_name`*.*`tbl_name`*.*`col_name`*. You need not specify a *`tbl_name`* or *`db_name`*.*`tbl_name`* prefix for a column reference unless the reference would be ambiguous. See Section 9.2.2, “Identifier Qualifiers”, for examples of ambiguity that require the more explicit column reference forms.

- A table reference can be aliased using *`tbl_name`* `AS` *`alias_name`* or *`tbl_name`* *`alias_name`*. These statements are equivalent:

```
SELECT t1.name, t2.salary FROM employee AS t1, info AS t2
  WHERE t1.name = t2.name;

SELECT t1.name, t2.salary FROM employee t1, info t2
  WHERE t1.name = t2.name;
```

- Columns selected for output can be referred to in `ORDER BY` and `GROUP BY` clauses using column names, column aliases, or column positions. Column positions are integers and begin with 1:

```
SELECT college, region, seed FROM tournament
  ORDER BY region, seed;

SELECT college, region AS r, seed AS s FROM tournament
  ORDER BY r, s;

SELECT college, region, seed FROM tournament
```

```
  ORDER BY 2, 3;
```

To sort in reverse order, add the `DESC` (descending) keyword to the name of the column in the `ORDER BY` clause that you are sorting by. The default is ascending order; this can be specified explicitly using the `ASC` keyword.

If `ORDER BY` occurs within a parenthesized query expression and also is applied in the outer query, the results are undefined and may change in a future version of MySQL.

Use of column positions is deprecated because the syntax has been removed from the SQL standard.

- Prior to MySQL 8.0.13, MySQL supported a nonstandard syntax extension that permitted explicit `ASC` or `DESC` designators for `GROUP BY` columns. MySQL 8.0.12 and later supports `ORDER BY` with grouping functions so that use of this extension is no longer necessary. (Bug #86312, Bug #26073525) This also means you can sort on an arbitrary column or columns when using `GROUP BY`, like this:

```
SELECT a, b, COUNT(c) AS t FROM test_table GROUP BY a,b ORDER BY a,t DESC;
```

As of MySQL 8.0.13, the `GROUP BY` extension is no longer supported: `ASC` or `DESC` designators for `GROUP BY` columns are not permitted.

- When you use `ORDER BY` or `GROUP BY` to sort a column in a `SELECT`, the server sorts values using only the initial number of bytes indicated by the `max_sort_length` system variable.

- MySQL extends the use of `GROUP BY` to permit selecting fields that are not mentioned in the `GROUP BY` clause. If you are not getting the results that you expect from your query, please read the description of `GROUP BY` found in Section 12.20, “Aggregate Functions”.

- `GROUP BY` permits a `WITH ROLLUP` modifier. See Section 12.20.2, “GROUP BY Modifiers”.

  Previously, it was not permitted to use `ORDER BY` in a query having a `WITH ROLLUP` modifier. This restriction is lifted as of MySQL 8.0.12. See Section 12.20.2, “GROUP BY Modifiers”.

- The `HAVING` clause, like the `WHERE` clause, specifies selection conditions. The `WHERE` clause specifies conditions on columns in the select list, but cannot refer to aggregate functions. The `HAVING` clause specifies conditions on groups, typically formed by the `GROUP BY` clause. The query result includes only groups satisfying the `HAVING` conditions. (If no `GROUP BY` is present, all rows implicitly form a single aggregate group.)

  The `HAVING` clause is applied nearly last, just before items are sent to the client, with no optimization. (`LIMIT` is applied after `HAVING`.)

  The SQL standard requires that `HAVING` must reference only columns in the `GROUP BY` clause or columns used in aggregate functions. However, MySQL supports an extension to this behavior, and permits `HAVING` to refer to columns in the `SELECT` list and columns in outer subqueries as well.

  If the `HAVING` clause refers to a column that is ambiguous, a warning occurs. In the following statement, `col2` is ambiguous because it is used as both an alias and a column name:

```
SELECT COUNT(col1) AS col2 FROM t GROUP BY col2 HAVING col2 = 2;
```

  Preference is given to standard SQL behavior, so if a `HAVING` column name is used both in `GROUP BY` and as an aliased column in the select column list, preference is given to the column in the `GROUP BY` column.

- Do not use `HAVING` for items that should be in the `WHERE` clause. For example, do not write the following:

```
SELECT col_name FROM tbl_name HAVING col_name > 0;
```

  Write this instead:

```
SELECT col_name FROM tbl_name WHERE col_name > 0;
```

- The HAVING clause can refer to aggregate functions, which the WHERE clause cannot:

```
SELECT user, MAX(salary) FROM users
  GROUP BY user HAVING MAX(salary) > 10;
```

(This did not work in some older versions of MySQL.)

- MySQL permits duplicate column names. That is, there can be more than one *select_expr* with the same name. This is an extension to standard SQL. Because MySQL also permits GROUP BY and HAVING to refer to *select_expr* values, this can result in an ambiguity:

```
SELECT 12 AS a, a FROM t GROUP BY a;
```

In that statement, both columns have the name a. To ensure that the correct column is used for grouping, use different names for each *select_expr*.

- The WINDOW clause, if present, defines named windows that can be referred to by window functions. For details, see Section 12.21.4, “Named Windows”.

- MySQL resolves unqualified column or alias references in ORDER BY clauses by searching in the *select_expr* values, then in the columns of the tables in the FROM clause. For GROUP BY or HAVING clauses, it searches the FROM clause before searching in the *select_expr* values. (For GROUP BY and HAVING, this differs from the pre-MySQL 5.0 behavior that used the same rules as for ORDER BY.)

- The LIMIT clause can be used to constrain the number of rows returned by the SELECT statement. LIMIT takes one or two numeric arguments, which must both be nonnegative integer constants, with these exceptions:

  - Within prepared statements, LIMIT parameters can be specified using ? placeholder markers.

  - Within stored programs, LIMIT parameters can be specified using integer-valued routine parameters or local variables.

With two arguments, the first argument specifies the offset of the first row to return, and the second specifies the maximum number of rows to return. The offset of the initial row is 0 (not 1):

```
SELECT * FROM tbl LIMIT 5,10;  # Retrieve rows 6-15
```

To retrieve all rows from a certain offset up to the end of the result set, you can use some large number for the second parameter. This statement retrieves all rows from the 96th row to the last:

```
SELECT * FROM tbl LIMIT 95,18446744073709551615;
```

With one argument, the value specifies the number of rows to return from the beginning of the result set:

```
SELECT * FROM tbl LIMIT 5;     # Retrieve first 5 rows
```

In other words, LIMIT *row_count* is equivalent to LIMIT 0, *row_count*.

For prepared statements, you can use placeholders. The following statements return one row from the tbl table:

```
SET @a=1;
PREPARE STMT FROM 'SELECT * FROM tbl LIMIT ?';
EXECUTE STMT USING @a;
```

The following statements return the second to sixth rows from the tbl table:

```
SET @skip=1; SET @numrows=5;
PREPARE STMT FROM 'SELECT * FROM tbl LIMIT ?, ?';
```

```
EXECUTE STMT USING @skip, @numrows;
```

For compatibility with PostgreSQL, MySQL also supports the `LIMIT row_count OFFSET offset` syntax.

If `LIMIT` occurs within a parenthesized query expression and also is applied in the outer query, the results are undefined and may change in a future version of MySQL.

- The `SELECT ... INTO` form of `SELECT` enables the query result to be written to a file or stored in variables. For more information, see Section 13.2.13.1, “SELECT ... INTO Statement”.

- If you use `FOR UPDATE` with a storage engine that uses page or row locks, rows examined by the query are write-locked until the end of the current transaction.

  You cannot use `FOR UPDATE` as part of the `SELECT` in a statement such as `CREATE TABLE new_table SELECT ... FROM old_table ...`. (If you attempt to do so, the statement is rejected with the error `Can't update table 'old_table' while 'new_table' is being created`.)

  `FOR SHARE` and `LOCK IN SHARE MODE` set shared locks that permit other transactions to read the examined rows but not to update or delete them. `FOR SHARE` and `LOCK IN SHARE MODE` are equivalent. However, `FOR SHARE`, like `FOR UPDATE`, supports `NOWAIT`, `SKIP LOCKED`, and `OF tbl_name` options. `FOR SHARE` is a replacement for `LOCK IN SHARE MODE`, but `LOCK IN SHARE MODE` remains available for backward compatibility.

  `NOWAIT` causes a `FOR UPDATE` or `FOR SHARE` query to execute immediately, returning an error if a row lock cannot be obtained due to a lock held by another transaction.

  `SKIP LOCKED` causes a `FOR UPDATE` or `FOR SHARE` query to execute immediately, excluding rows from the result set that are locked by another transaction.

  `NOWAIT` and `SKIP LOCKED` options are unsafe for statement-based replication.

  

  Note

  Queries that skip locked rows return an inconsistent view of the data. `SKIP LOCKED` is therefore not suitable for general transactional work. However, it may be used to avoid lock contention when multiple sessions access the same queue-like table.

  `OF tbl_name` applies `FOR UPDATE` and `FOR SHARE` queries to named tables. For example:

```
SELECT * FROM t1, t2 FOR SHARE OF t1 FOR UPDATE OF t2;
```

  All tables referenced by the query block are locked when `OF tbl_name` is omitted. Consequently, using a locking clause without `OF tbl_name` in combination with another locking clause returns an error. Specifying the same table in multiple locking clauses returns an error. If an alias is specified as the table name in the `SELECT` statement, a locking clause may only use the alias. If the `SELECT` statement does not specify an alias explicitly, the locking clause may only specify the actual table name.

  For more information about `FOR UPDATE` and `FOR SHARE`, see Section 15.7.2.4, “Locking Reads”. For additional information about `NOWAIT` and `SKIP LOCKED` options, see Locking Read Concurrency with NOWAIT and SKIP LOCKED.

Following the `SELECT` keyword, you can use a number of modifiers that affect the operation of the statement. `HIGH_PRIORITY`, `STRAIGHT_JOIN`, and modifiers beginning with `SQL_` are MySQL extensions to standard SQL.

- The `ALL` and `DISTINCT` modifiers specify whether duplicate rows should be returned. `ALL` (the default) specifies that all matching rows should be returned, including duplicates. `DISTINCT`

specifies removal of duplicate rows from the result set. It is an error to specify both modifiers. `DISTINCTROW` is a synonym for `DISTINCT`.

In MySQL 8.0.12 and later, `DISTINCT` can be used with a query that also uses `WITH ROLLUP`. (Bug #87450, Bug #26640100)

- `HIGH_PRIORITY` gives the `SELECT` higher priority than a statement that updates a table. You should use this only for queries that are very fast and must be done at once. A `SELECT HIGH_PRIORITY` query that is issued while the table is locked for reading runs even if there is an update statement waiting for the table to be free. This affects only storage engines that use only table-level locking (such as `MyISAM`, `MEMORY`, and `MERGE`).

  `HIGH_PRIORITY` cannot be used with `SELECT` statements that are part of a `UNION`.

- `STRAIGHT_JOIN` forces the optimizer to join the tables in the order in which they are listed in the `FROM` clause. You can use this to speed up a query if the optimizer joins the tables in nonoptimal order. `STRAIGHT_JOIN` also can be used in the *`table_references`* list. See Section 13.2.13.2, “JOIN Clause”.

  `STRAIGHT_JOIN` does not apply to any table that the optimizer treats as a `const` or `system` table. Such a table produces a single row, is read during the optimization phase of query execution, and references to its columns are replaced with the appropriate column values before query execution proceeds. These tables appear first in the query plan displayed by `EXPLAIN`. See Section 8.8.1, “Optimizing Queries with EXPLAIN”. This exception may not apply to `const` or `system` tables that are used on the `NULL`-complemented side of an outer join (that is, the right-side table of a `LEFT JOIN` or the left-side table of a `RIGHT JOIN`.

- `SQL_BIG_RESULT` or `SQL_SMALL_RESULT` can be used with `GROUP BY` or `DISTINCT` to tell the optimizer that the result set has many rows or is small, respectively. For `SQL_BIG_RESULT`, MySQL directly uses disk-based temporary tables if they are created, and prefers sorting to using a temporary table with a key on the `GROUP BY` elements. For `SQL_SMALL_RESULT`, MySQL uses in-memory temporary tables to store the resulting table instead of using sorting. This should not normally be needed.

- `SQL_BUFFER_RESULT` forces the result to be put into a temporary table. This helps MySQL free the table locks early and helps in cases where it takes a long time to send the result set to the client. This modifier can be used only for top-level `SELECT` statements, not for subqueries or following `UNION`.

- `SQL_CALC_FOUND_ROWS` tells MySQL to calculate how many rows there would be in the result set, disregarding any `LIMIT` clause. The number of rows can then be retrieved with `SELECT FOUND_ROWS()`. See Section 12.16, “Information Functions”.

  

  Note

  The `SQL_CALC_FOUND_ROWS` query modifier and accompanying `FOUND_ROWS()` function are deprecated as of MySQL 8.0.17; expect them to be removed in a future version of MySQL. See the `FOUND_ROWS()` description for information about an alternative strategy.

- The `SQL_CACHE` and `SQL_NO_CACHE` modifiers were used with the query cache prior to MySQL 8.0. The query cache was removed in MySQL 8.0. The `SQL_CACHE` modifier was removed as well. `SQL_NO_CACHE` is deprecated, and has no effect; expect it to be removed in a future MySQL release.

13.2.13.1 SELECT ... INTO Statement

The `SELECT ... INTO` form of `SELECT` enables a query result to be stored in variables or written to a file:

- `SELECT ... INTO var_list` selects column values and stores them into variables.
- `SELECT ... INTO OUTFILE` writes the selected rows to a file. Column and line terminators can be specified to produce a specific output format.
- `SELECT ... INTO DUMPFILE` writes a single row to a file without any formatting.

A given `SELECT` statement can contain at most one `INTO` clause, although as shown by the `SELECT` syntax description (see Section 13.2.13, “SELECT Statement”), the `INTO` can appear in different positions:

- Before `FROM`. Example:

```
SELECT * INTO @myvar FROM t1;
```

- Before a trailing locking clause. Example:

```
SELECT * FROM t1 INTO @myvar FOR UPDATE;
```

- At the end of the `SELECT`. Example:

```
SELECT * FROM t1 FOR UPDATE INTO @myvar;
```

The `INTO` position at the end of the statement is supported as of MySQL 8.0.20, and is the preferred position. The position before a locking clause is deprecated as of MySQL 8.0.20; expect support for it to be removed in a future version of MySQL. In other words, `INTO` after `FROM` but not at the end of the `SELECT` produces a warning.

An `INTO` clause should not be used in a nested `SELECT` because such a `SELECT` must return its result to the outer context. There are also constraints on the use of `INTO` within `UNION` statements; see Section 13.2.18, “UNION Clause”.

For the `INTO var_list` variant:

- *`var_list`* names a list of one or more variables, each of which can be a user-defined variable, stored procedure or function parameter, or stored program local variable. (Within a prepared `SELECT ... INTO var_list` statement, only user-defined variables are permitted; see Section 13.6.4.2, “Local Variable Scope and Resolution”.)
- The selected values are assigned to the variables. The number of variables must match the number of columns. The query should return a single row. If the query returns no rows, a warning with error code 1329 occurs (`No data`), and the variable values remain unchanged. If the query returns multiple rows, error 1172 occurs (`Result consisted of more than one row`). If it is possible that the statement may retrieve multiple rows, you can use `LIMIT 1` to limit the result set to a single row.

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

`INTO var_list` can also be used with a `TABLE` statement, subject to these restrictions:

- The number of variables must match the number of columns in the table.
- If the table contains more than one row, you must use `LIMIT 1` to limit the result set to a single row. `LIMIT 1` must precede the `INTO` keyword.

An example of such a statement is shown here:

```
TABLE employees ORDER BY lname DESC LIMIT 1
    INTO @id, @fname, @lname, @hired, @separated, @job_code, @store_id;
```

You can also select values from a `VALUES` statement that generates a single row into a set of user variables. In this case, you must employ a table alias, and you must assign each value from the value list to a variable. Each of the two statements shown here is equivalent to `SET @x=2, @y=4, @z=8`:

```
SELECT * FROM (VALUES ROW(2,4,8)) AS t INTO @x,@y,@z;

SELECT * FROM (VALUES ROW(2,4,8)) AS t(a,b,c) INTO @x,@y,@z;
```

User variable names are not case-sensitive. See Section 9.4, “User-Defined Variables”.

The `SELECT ... INTO OUTFILE 'file_name'` form of `SELECT` writes the selected rows to a file. The file is created on the server host, so you must have the `FILE` privilege to use this syntax. *`file_name`* cannot be an existing file, which among other things prevents files such as `/etc/passwd` and database tables from being modified. The `character_set_filesystem` system variable controls the interpretation of the file name.

The `SELECT ... INTO OUTFILE` statement is intended to enable dumping a table to a text file on the server host. To create the resulting file on some other host, `SELECT ... INTO OUTFILE` normally is unsuitable because there is no way to write a path to the file relative to the server host file system, unless the location of the file on the remote host can be accessed using a network-mapped path on the server host file system.

Alternatively, if the MySQL client software is installed on the remote host, you can use a client command such as `mysql -e "SELECT ..." > file_name` to generate the file on that host.

`SELECT ... INTO OUTFILE` is the complement of `LOAD DATA`. Column values are written converted to the character set specified in the `CHARACTER SET` clause. If no such clause is present, values are dumped using the `binary` character set. In effect, there is no character set conversion. If a result set contains columns in several character sets, so is the output data file, and it may not be possible to reload the file correctly.

The syntax for the *`export_options`* part of the statement consists of the same `FIELDS` and `LINES` clauses that are used with the `LOAD DATA` statement. For information about the `FIELDS` and `LINES` clauses, including their default values and permissible values, see Section 13.2.9, “LOAD DATA Statement”.

`FIELDS ESCAPED BY` controls how to write special characters. If the `FIELDS ESCAPED BY` character is not empty, it is used when necessary to avoid ambiguity as a prefix that precedes following characters on output:

- The `FIELDS ESCAPED BY` character
- The `FIELDS [OPTIONALLY] ENCLOSED BY` character
- The first character of the `FIELDS TERMINATED BY` and `LINES TERMINATED BY` values
- ASCII `NUL` (the zero-valued byte; what is actually written following the escape character is ASCII `0`, not a zero-valued byte)

The `FIELDS TERMINATED BY`, `ENCLOSED BY`, `ESCAPED BY`, or `LINES TERMINATED BY` characters *must* be escaped so that you can read the file back in reliably. ASCII `NUL` is escaped to make it easier to view with some pagers.

The resulting file need not conform to SQL syntax, so nothing else need be escaped.

If the `FIELDS ESCAPED BY` character is empty, no characters are escaped and `NULL` is output as `NULL`, not `\N`. It is probably not a good idea to specify an empty escape character, particularly if field values in your data contain any of the characters in the list just given.

`INTO OUTFILE` can also be used with a `TABLE` statement when you want to dump all columns of a table into a text file. In this case, the ordering and number of rows can be controlled using `ORDER BY` and `LIMIT`; these clauses must precede `INTO OUTFILE`. `TABLE ... INTO OUTFILE` supports the same *`export_options`* as does `SELECT ... INTO OUTFILE`, and it is subject to the same restrictions on writing to the file system. An example of such a statement is shown here:

```
TABLE employees ORDER BY lname LIMIT 1000
    INTO OUTFILE '/tmp/employee_data_1.txt'
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"', ESCAPED BY '\'
```

```
    LINES TERMINATED BY '\n';
```

You can also use `SELECT ... INTO OUTFILE` with a `VALUES` statement to write values directly into a file. An example is shown here:

```
SELECT * FROM (VALUES ROW(1,2,3),ROW(4,5,6),ROW(7,8,9)) AS t
    INTO OUTFILE '/tmp/select-values.txt';
```

You must use a table alias; column aliases are also supported, and can optionally be used to write values only from desired columns. You can also use any or all of the export options supported by `SELECT ... INTO OUTFILE` to format the output to the file.

Here is an example that produces a file in the comma-separated values (CSV) format used by many programs:

```
SELECT a,b,a+b INTO OUTFILE '/tmp/result.txt'
  FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
  LINES TERMINATED BY '\n'
  FROM test_table;
```

If you use `INTO DUMPFILE` instead of `INTO OUTFILE`, MySQL writes only one row into the file, without any column or line termination and without performing any escape processing. This is useful for selecting a `BLOB` value and storing it in a file.

`TABLE` also supports `INTO DUMPFILE`. If the table contains more than one row, you must also use `LIMIT 1` to limit the output to a single row. `INTO DUMPFILE` can also be used with `SELECT * FROM (VALUES ROW()[, ...]) AS table_alias [LIMIT 1]`. See Section 13.2.19, “VALUES Statement”.

Note

Any file created by `INTO OUTFILE` or `INTO DUMPFILE` is owned by the operating system user under whose account `mysqld` runs. (You should *never* run `mysqld` as `root` for this and other reasons.) As of MySQL 8.0.17, the umask for file creation is 0640; you must have sufficient access privileges to manipulate the file contents. Prior to MySQL 8.0.17, the umask is 0666 and the file is writable by all users on the server host.

If the `secure_file_priv` system variable is set to a nonempty directory name, the file to be written must be located in that directory.

In the context of `SELECT ... INTO` statements that occur as part of events executed by the Event Scheduler, diagnostics messages (not only errors, but also warnings) are written to the error log, and, on Windows, to the application event log. For additional information, see Section 25.4.5, “Event Scheduler Status”.

As of MySQL 8.0.22, support is provided for periodic synchronization of output files written to by `SELECT INTO OUTFILE` and `SELECT INTO DUMPFILE`, enabled by setting the `select_into_disk_sync` server system variable introduced in that version. Output buffer size and optional delay can be set using, respectively, `select_into_buffer_size` and `select_into_disk_sync_delay`. For more information, see the descriptions of these system variables.

13.2.13.2 JOIN Clause

MySQL supports the following `JOIN` syntax for the *`table_references`* part of `SELECT` statements and multiple-table `DELETE` and `UPDATE` statements:

```
table_references:
    escaped_table_reference [, escaped_table_reference] ...

escaped_table_reference: {
    table_reference
  | { OJ table_reference }
```

```
}

table_reference: {
    table_factor
  | joined_table
}

table_factor: {
    tbl_name [PARTITION (partition_names)]
        [[AS] alias] [index_hint_list]
  | [LATERAL] table_subquery [AS] alias [(col_list)]
  | ( table_references )
}

joined_table: {
    table_reference {[INNER | CROSS] JOIN | STRAIGHT_JOIN} table_factor [join_specification]
  | table_reference {LEFT|RIGHT} [OUTER] JOIN table_reference join_specification
  | table_reference NATURAL [INNER | {LEFT|RIGHT} [OUTER]] JOIN table_factor
}

join_specification: {
    ON search_condition
  | USING (join_column_list)
}

join_column_list:
    column_name [, column_name] ...

index_hint_list:
    index_hint [, index_hint] ...

index_hint: {
    USE {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] ([index_list])
  | {IGNORE|FORCE} {INDEX|KEY}
      [FOR {JOIN|ORDER BY|GROUP BY}] (index_list)
}

index_list:
    index_name [, index_name] ...
```

A table reference is also known as a join expression.

A table reference (when it refers to a partitioned table) may contain a `PARTITION` clause, including a list of comma-separated partitions, subpartitions, or both. This option follows the name of the table and precedes any alias declaration. The effect of this option is that rows are selected only from the listed partitions or subpartitions. Any partitions or subpartitions not named in the list are ignored. For more information and examples, see Section 24.5, “Partition Selection”.

The syntax of *`table_factor`* is extended in MySQL in comparison with standard SQL. The standard accepts only *`table_reference`*, not a list of them inside a pair of parentheses.

This is a conservative extension if each comma in a list of *`table_reference`* items is considered as equivalent to an inner join. For example:

```
SELECT * FROM t1 LEFT JOIN (t2, t3, t4)
                 ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c)
```

is equivalent to:

```
SELECT * FROM t1 LEFT JOIN (t2 CROSS JOIN t3 CROSS JOIN t4)
                 ON (t2.a = t1.a AND t3.b = t1.b AND t4.c = t1.c)
```

In MySQL, `JOIN`, `CROSS JOIN`, and `INNER JOIN` are syntactic equivalents (they can replace each other). In standard SQL, they are not equivalent. `INNER JOIN` is used with an `ON` clause, `CROSS JOIN` is used otherwise.

In general, parentheses can be ignored in join expressions containing only inner join operations. MySQL also supports nested joins. See Section 8.2.1.8, “Nested Join Optimization”.

Index hints can be specified to affect how the MySQL optimizer makes use of indexes. For more information, see Section 8.9.4, “Index Hints”. Optimizer hints and the `optimizer_switch` system variable are other ways to influence optimizer use of indexes. See Section 8.9.3, “Optimizer Hints”, and Section 8.9.2, “Switchable Optimizations”.

The following list describes general factors to take into account when writing joins:

- A table reference can be aliased using *`tbl_name`* `AS` *`alias_name`* or *`tbl_name alias_name`*:

```
SELECT t1.name, t2.salary
  FROM employee AS t1 INNER JOIN info AS t2 ON t1.name = t2.name;

SELECT t1.name, t2.salary
  FROM employee t1 INNER JOIN info t2 ON t1.name = t2.name;
```

- A *`table_subquery`* is also known as a derived table or subquery in the `FROM` clause. See Section 13.2.15.8, “Derived Tables”. Such subqueries *must* include an alias to give the subquery result a table name, and may optionally include a list of table column names in parentheses. A trivial example follows:

```
SELECT * FROM (SELECT 1, 2, 3) AS t1;
```

- The maximum number of tables that can be referenced in a single join is 61. This includes a join handled by merging derived tables and views in the `FROM` clause into the outer query block (see Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization”).

- `INNER JOIN` and `,` (comma) are semantically equivalent in the absence of a join condition: both produce a Cartesian product between the specified tables (that is, each and every row in the first table is joined to each and every row in the second table).

  However, the precedence of the comma operator is less than that of `INNER JOIN`, `CROSS JOIN`, `LEFT JOIN`, and so on. If you mix comma joins with the other join types when there is a join condition, an error of the form `Unknown column '`*`col_name`*`' in 'on clause'` may occur. Information about dealing with this problem is given later in this section.

- The *`search_condition`* used with `ON` is any conditional expression of the form that can be used in a `WHERE` clause. Generally, the `ON` clause serves for conditions that specify how to join tables, and the `WHERE` clause restricts which rows to include in the result set.

- If there is no matching row for the right table in the `ON` or `USING` part in a `LEFT JOIN`, a row with all columns set to `NULL` is used for the right table. You can use this fact to find rows in a table that have no counterpart in another table:

```
SELECT left_tbl.*
  FROM left_tbl LEFT JOIN right_tbl ON left_tbl.id = right_tbl.id
  WHERE right_tbl.id IS NULL;
```

  This example finds all rows in `left_tbl` with an `id` value that is not present in `right_tbl` (that is, all rows in `left_tbl` with no corresponding row in `right_tbl`). See Section 8.2.1.9, “Outer Join Optimization”.

- The `USING(`*`join_column_list`*`)` clause names a list of columns that must exist in both tables. If tables `a` and `b` both contain columns `c1`, `c2`, and `c3`, the following join compares corresponding columns from the two tables:

```
a LEFT JOIN b USING (c1, c2, c3)
```

- The `NATURAL [LEFT] JOIN` of two tables is defined to be semantically equivalent to an `INNER JOIN` or a `LEFT JOIN` with a `USING` clause that names all columns that exist in both tables.

- `RIGHT JOIN` works analogously to `LEFT JOIN`. To keep code portable across databases, it is recommended that you use `LEFT JOIN` instead of `RIGHT JOIN`.

- The `{ OJ ... }` syntax shown in the join syntax description exists only for compatibility with ODBC. The curly braces in the syntax should be written literally; they are not metasyntax as used elsewhere in syntax descriptions.

```
SELECT left_tbl.*
    FROM { OJ left_tbl LEFT OUTER JOIN right_tbl
           ON left_tbl.id = right_tbl.id }
    WHERE right_tbl.id IS NULL;
```

  You can use other types of joins within `{ OJ ... }`, such as `INNER JOIN` or `RIGHT OUTER JOIN`. This helps with compatibility with some third-party applications, but is not official ODBC syntax.

- `STRAIGHT_JOIN` is similar to `JOIN`, except that the left table is always read before the right table. This can be used for those (few) cases for which the join optimizer processes the tables in a suboptimal order.

Some join examples:

```
SELECT * FROM table1, table2;

SELECT * FROM table1 INNER JOIN table2 ON table1.id = table2.id;

SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id;

SELECT * FROM table1 LEFT JOIN table2 USING (id);

SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id
  LEFT JOIN table3 ON table2.id = table3.id;
```

Natural joins and joins with `USING`, including outer join variants, are processed according to the SQL:2003 standard:

- Redundant columns of a `NATURAL` join do not appear. Consider this set of statements:

```
CREATE TABLE t1 (i INT, j INT);
CREATE TABLE t2 (k INT, j INT);
INSERT INTO t1 VALUES(1, 1);
INSERT INTO t2 VALUES(1, 1);
SELECT * FROM t1 NATURAL JOIN t2;
SELECT * FROM t1 JOIN t2 USING (j);
```

  In the first `SELECT` statement, column `j` appears in both tables and thus becomes a join column, so, according to standard SQL, it should appear only once in the output, not twice. Similarly, in the second SELECT statement, column `j` is named in the `USING` clause and should appear only once in the output, not twice.

  Thus, the statements produce this output:

```
+------+------+------+
| j    | i    | k    |
+------+------+------+
|    1 |    1 |    1 |
+------+------+------+
+------+------+------+
| j    | i    | k    |
+------+------+------+
|    1 |    1 |    1 |
+------+------+------+
```

  Redundant column elimination and column ordering occurs according to standard SQL, producing this display order:

  - First, coalesced common columns of the two joined tables, in the order in which they occur in the first table
  - Second, columns unique to the first table, in order in which they occur in that table

- Third, columns unique to the second table, in order in which they occur in that table

The single result column that replaces two common columns is defined using the coalesce operation. That is, for two `t1.a` and `t2.a` the resulting single join column `a` is defined as `a = COALESCE(t1.a, t2.a)`, where:

```
COALESCE(x, y) = (CASE WHEN x IS NOT NULL THEN x ELSE y END)
```

If the join operation is any other join, the result columns of the join consist of the concatenation of all columns of the joined tables.

A consequence of the definition of coalesced columns is that, for outer joins, the coalesced column contains the value of the non-`NULL` column if one of the two columns is always `NULL`. If neither or both columns are `NULL`, both common columns have the same value, so it doesn't matter which one is chosen as the value of the coalesced column. A simple way to interpret this is to consider that a coalesced column of an outer join is represented by the common column of the inner table of a `JOIN`. Suppose that the tables `t1(a, b)` and `t2(a, c)` have the following contents:

```
t1    t2
----  ----
1 x   2 z
2 y   3 w
```

Then, for this join, column `a` contains the values of `t1.a`:

```
mysql> SELECT * FROM t1 NATURAL LEFT JOIN t2;
+------+------+------+
| a    | b    | c    |
+------+------+------+
|    1 | x    | NULL |
|    2 | y    | z    |
+------+------+------+
```

By contrast, for this join, column `a` contains the values of `t2.a`.

```
mysql> SELECT * FROM t1 NATURAL RIGHT JOIN t2;
+------+------+------+
| a    | c    | b    |
+------+------+------+
|    2 | z    | y    |
|    3 | w    | NULL |
+------+------+------+
```

Compare those results to the otherwise equivalent queries with `JOIN ... ON`:

```
mysql> SELECT * FROM t1 LEFT JOIN t2 ON (t1.a = t2.a);
+------+------+------+------+
| a    | b    | a    | c    |
+------+------+------+------+
|    1 | x    | NULL | NULL |
|    2 | y    |    2 | z    |
+------+------+------+------+
```

```
mysql> SELECT * FROM t1 RIGHT JOIN t2 ON (t1.a = t2.a);
+------+------+------+------+
| a    | b    | a    | c    |
+------+------+------+------+
|    2 | y    |    2 | z    |
| NULL | NULL |    3 | w    |
+------+------+------+------+
```

- A `USING` clause can be rewritten as an `ON` clause that compares corresponding columns. However, although `USING` and `ON` are similar, they are not quite the same. Consider the following two queries:

```
a LEFT JOIN b USING (c1, c2, c3)
```

```
a LEFT JOIN b ON a.c1 = b.c1 AND a.c2 = b.c2 AND a.c3 = b.c3
```

With respect to determining which rows satisfy the join condition, both joins are semantically identical.

With respect to determining which columns to display for `SELECT *` expansion, the two joins are not semantically identical. The `USING` join selects the coalesced value of corresponding columns, whereas the `ON` join selects all columns from all tables. For the `USING` join, `SELECT *` selects these values:

```
COALESCE(a.c1, b.c1), COALESCE(a.c2, b.c2), COALESCE(a.c3, b.c3)
```

For the `ON` join, `SELECT *` selects these values:

```
a.c1, a.c2, a.c3, b.c1, b.c2, b.c3
```

With an inner join, `COALESCE(a.c1, b.c1)` is the same as either `a.c1` or `b.c1` because both columns have the same value. With an outer join (such as `LEFT JOIN`), one of the two columns can be `NULL`. That column is omitted from the result.

- An `ON` clause can refer only to its operands.

  Example:

  ```
  CREATE TABLE t1 (i1 INT);
  CREATE TABLE t2 (i2 INT);
  CREATE TABLE t3 (i3 INT);
  SELECT * FROM t1 JOIN t2 ON (i1 = i3) JOIN t3;
  ```

  The statement fails with an `Unknown column 'i3' in 'on clause'` error because `i3` is a column in `t3`, which is not an operand of the `ON` clause. To enable the join to be processed, rewrite the statement as follows:

  ```
  SELECT * FROM t1 JOIN t2 JOIN t3 ON (i1 = i3);
  ```

- `JOIN` has higher precedence than the comma operator (`,`), so the join expression `t1, t2 JOIN t3` is interpreted as `(t1, (t2 JOIN t3))`, not as `((t1, t2) JOIN t3)`. This affects statements that use an `ON` clause because that clause can refer only to columns in the operands of the join, and the precedence affects interpretation of what those operands are.

  Example:

  ```
  CREATE TABLE t1 (i1 INT, j1 INT);
  CREATE TABLE t2 (i2 INT, j2 INT);
  CREATE TABLE t3 (i3 INT, j3 INT);
  INSERT INTO t1 VALUES(1, 1);
  INSERT INTO t2 VALUES(1, 1);
  INSERT INTO t3 VALUES(1, 1);
  SELECT * FROM t1, t2 JOIN t3 ON (t1.i1 = t3.i3);
  ```

  The `JOIN` takes precedence over the comma operator, so the operands for the `ON` clause are `t2` and `t3`. Because `t1.i1` is not a column in either of the operands, the result is an `Unknown column 't1.i1' in 'on clause'` error.

  To enable the join to be processed, use either of these strategies:

  - Group the first two tables explicitly with parentheses so that the operands for the `ON` clause are `(t1, t2)` and `t3`:

    ```
    SELECT * FROM (t1, t2) JOIN t3 ON (t1.i1 = t3.i3);
    ```

- Avoid the use of the comma operator and use `JOIN` instead:

```
SELECT * FROM t1 JOIN t2 JOIN t3 ON (t1.i1 = t3.i3);
```

The same precedence interpretation also applies to statements that mix the comma operator with `INNER JOIN`, `CROSS JOIN`, `LEFT JOIN`, and `RIGHT JOIN`, all of which have higher precedence than the comma operator.

- A MySQL extension compared to the SQL:2003 standard is that MySQL permits you to qualify the common (coalesced) columns of `NATURAL` or `USING` joins, whereas the standard disallows that.

13.2.14 Set Operations with UNION, INTERSECT, and EXCEPT

SQL set operations combine the results of multiple query blocks into a single result. A *query block*, sometimes also known as a *simple table*, is any SQL statement that returns a result set, such as `SELECT`. MySQL 8.0 (8.0.19 and later) also supports `TABLE` and `VALUES` statements. See the individual descriptions of these statements elsewhere in this chapter for additional information.

The SQL standard defines the following three set operations:

- `UNION`: Combine all results from two query blocks into a single result, omitting any duplicates.
- `INTERSECT`: Combine only those rows which the results of two query blocks have in common, omitting any duplicates.
- `EXCEPT`: For two query blocks *A* and *B*, return all results from *A* which are not also present in *B*, omitting any duplicates.

  (Some database systems, such as Oracle, use `MINUS` for the name of this operator. This is not supported in MySQL.)

MySQL has long supported `UNION`; MySQL 8.0 adds support for `INTERSECT` and `EXCEPT` (MySQL 8.0.31 and later).

Each of these set operators supports an `ALL` modifier. When the `ALL` keyword follows a set operator, this causes duplicates to be included in the result. See the following sections covering the individual operators for more information and examples.

All three set operators also support a `DISTINCT` keyword, which suppresses duplicates in the result. Since this is the default behavior for set operators, it is usually not necessary to specify `DISTINCT` explicitly.

In general, query blocks and set operations can be combined in any number and order. A greatly simplified representation is shown here:

```
query_block [set_op query_block] [set_op query_block] ...

query_block:
    SELECT | TABLE | VALUES

set_op:
    UNION | INTERSECT | EXCEPT
```

This can be represented more accurately, and in greater detail, like this:

```
query_expression:
  [with_clause] /* WITH clause */
  query_expression_body
  [order_by_clause] [limit_clause] [into_clause]

query_expression_body:
    query_term
 |  query_expression_body UNION [ALL | DISTINCT] query_term
 |  query_expression_body EXCEPT [ALL | DISTINCT] query_term

query_term:
    query_primary
 |  query_term INTERSECT [ALL | DISTINCT] query_primary

query_primary:
    query_block
 |  '(' query_expression_body [order_by_clause] [limit_clause] [into_clause] ')'

query_block:   /* also known as a simple table */
    query_specification                    /* SELECT statement */
 |  table_value_constructor                /* VALUES statement */
 |  explicit_table                         /* TABLE statement  */
```

You should be aware that `INTERSECT` is evaluated before `UNION` or `EXCEPT`. This means that, for example, `TABLE x UNION TABLE y INTERSECT TABLE z` is always evaluated as `TABLE x UNION (TABLE y INTERSECT TABLE z)`. See Section 13.2.8, “INTERSECT Clause”, for more information.

In addition, you should keep in mind that, while the `UNION` and `INTERSECT` set operators are commutative (ordering is not significant), `EXCEPT` is not (order of operands affects the outcome). In other words, all of the following statements are true:

- `TABLE x UNION TABLE y` and `TABLE y UNION TABLE x` produce the same result, although the ordering of the rows may differ. You can force them to be the same using `ORDER BY`; see ORDER BY and LIMIT in Unions.
- `TABLE x INTERSECT TABLE y` and `TABLE y INTERSECT TABLE x` return the same result.
- `TABLE x EXCEPT TABLE y` and `TABLE y EXCEPT TABLE x` do *not* yield the same result. See Section 13.2.4, “EXCEPT Clause”, for an example.

More information and examples can be found in the sections that follow.

Result Set Column Names and Data Types

The column names for the result of a set operation are taken from the column names of the first query block. Example:

```
mysql> CREATE TABLE t1 (x INT, y INT);
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO t1 VALUES ROW(4,-2), ROW(5,9);
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> CREATE TABLE t2 (a INT, b INT);
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO t2 VALUES ROW(1,2), ROW(3,4);
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> TABLE t1 UNION TABLE t2;
+------+------+
| x    | y    |
+------+------+
```

```
|    4 |   -2 |
|    5 |    9 |
|    1 |    2 |
|    3 |    4 |
+------+------+
4 rows in set (0.00 sec)

mysql> TABLE t2 UNION TABLE t1;
+------+------+
| a    | b    |
+------+------+
|    1 |    2 |
|    3 |    4 |
|    4 |   -2 |
|    5 |    9 |
+------+------+
4 rows in set (0.00 sec)
```

This is true for `UNION`, `EXCEPT`, and `INTERSECT` queries.

Selected columns listed in corresponding positions of each query block should have the same data type. For example, the first column selected by the first statement should have the same type as the first column selected by the other statements. If the data types of corresponding result columns do not match, the types and lengths of the columns in the result take into account the values retrieved by all of the query blocks. For example, the column length in the result set is not constrained to the length of the value from the first statement, as shown here:

```
mysql> SELECT REPEAT('a',1) UNION SELECT REPEAT('b',20);
+----------------------+
| REPEAT('a',1)        |
+----------------------+
| a                    |
| bbbbbbbbbbbbbbbbbbbb |
+----------------------+
```

Set Operations with TABLE and VALUES Statements

Beginning with MySQL 8.0.19, you can also use a `TABLE` statement or `VALUES` statement wherever you can employ the equivalent `SELECT` statement. Assume that tables `t1` and `t2` are created and populated as shown here:

```
CREATE TABLE t1 (x INT, y INT);
INSERT INTO t1 VALUES ROW(4,-2),ROW(5,9);

CREATE TABLE t2 (a INT, b INT);
INSERT INTO t2 VALUES ROW(1,2),ROW(3,4);
```

The preceding being the case, and disregarding the column names in the output of the queries beginning with `VALUES`, all of the following `UNION` queries yield the same result:

```
SELECT * FROM t1 UNION SELECT * FROM t2;
TABLE t1 UNION SELECT * FROM t2;
VALUES ROW(4,-2), ROW(5,9) UNION SELECT * FROM t2;
SELECT * FROM t1 UNION TABLE t2;
TABLE t1 UNION TABLE t2;
VALUES ROW(4,-2), ROW(5,9) UNION TABLE t2;
SELECT * FROM t1 UNION VALUES ROW(4,-2),ROW(5,9);
TABLE t1 UNION VALUES ROW(4,-2),ROW(5,9);
VALUES ROW(4,-2), ROW(5,9) UNION VALUES ROW(4,-2),ROW(5,9);
```

To force the column names to be the same, wrap the query block on the left-hand side in a `SELECT` statement, and use aliases, like this:

```
mysql> SELECT * FROM (TABLE t2) AS t(x,y) UNION TABLE t1;
+------+------+
| x    | y    |
+------+------+
|    1 |    2 |
|    3 |    4 |
```

```
|    4 |   -2 |
|    5 |    9 |
+------+------+
4 rows in set (0.00 sec)
```

Set Operations using DISTINCT and ALL

By default, duplicate rows are removed from results of set operations. The optional `DISTINCT` keyword has the same effect but makes it explicit. With the optional `ALL` keyword, duplicate-row removal does not occur and the result includes all matching rows from all queries in the union.

You can mix `ALL` and `DISTINCT` in the same query. Mixed types are treated such that a set operation using `DISTINCT` overrides any such operation using `ALL` to its left. A `DISTINCT` set can be produced explicitly by using `DISTINCT` with `UNION`, `INTERSECT`, or `EXCEPT`, or implicitly by using the set operations with no following `DISTINCT` or `ALL` keyword.

In MySQL 8.0.19 and later, set operations work the same way when one or more `TABLE` statements, `VALUES` statements, or both, are used to generate the set.

Set Operations with ORDER BY and LIMIT

To apply an `ORDER BY` or `LIMIT` clause to an individual query block used as part of a union, intersection, or other set operation, parenthesize the query block, placing the clause inside the parentheses, like this:

```
(SELECT a FROM t1 WHERE a=10 AND b=1 ORDER BY a LIMIT 10)
UNION
(SELECT a FROM t2 WHERE a=11 AND b=2 ORDER BY a LIMIT 10);

(TABLE t1 ORDER BY x LIMIT 10)
INTERSECT
(TABLE t2 ORDER BY a LIMIT 10);
```

Use of `ORDER BY` for individual query blocks or statements implies nothing about the order in which the rows appear in the final result because the rows produced by a set operation are by default unordered. Therefore, `ORDER BY` in this context typically is used in conjunction with `LIMIT`, to determine the subset of the selected rows to retrieve, even though it does not necessarily affect the order of those rows in the final result. If `ORDER BY` appears without `LIMIT` within a query block, it is optimized away because it has no effect in any case.

To use an `ORDER BY` or `LIMIT` clause to sort or limit the entire result of a set operation, place the `ORDER BY` or `LIMIT` after the last statement:

```
SELECT a FROM t1
EXCEPT
SELECT a FROM t2 WHERE a=11 AND b=2
ORDER BY a LIMIT 10;

TABLE t1
UNION
TABLE t2
ORDER BY a LIMIT 10;
```

If one or more individual statements make use of `ORDER BY`, `LIMIT`, or both, and, in addition, you wish to apply an ORDER BY, LIMIT, or both to the entire result, then each such individual statement must be enclosed in parentheses.

```
(SELECT a FROM t1 WHERE a=10 AND b=1)
EXCEPT
(SELECT a FROM t2 WHERE a=11 AND b=2)
ORDER BY a LIMIT 10;

(TABLE t1 ORDER BY a LIMIT 10)
UNION
TABLE t2
ORDER BY a LIMIT 10;
```

A statement with no `ORDER BY` or `LIMIT` clause does need to be parenthesized; replacing `TABLE t2` with `(TABLE t2)` in the second statement of the two just shown does not alter the result of the `UNION`.

You can also use `ORDER BY` and `LIMIT` with `VALUES` statements in set operations, as shown in this example using the `mysql` client:

```
mysql> VALUES ROW(4,-2), ROW(5,9), ROW(-1,3)
    -> UNION
    -> VALUES ROW(1,2), ROW(3,4), ROW(-1,3)
    -> ORDER BY column_0 DESC LIMIT 3;
+----------+----------+
| column_0 | column_1 |
+----------+----------+
|        5 |        9 |
|        4 |       -2 |
|        3 |        4 |
+----------+----------+
3 rows in set (0.00 sec)
```

(You should keep in mind that neither `TABLE` statements nor `VALUES` statements accept a `WHERE` clause.)

This kind of `ORDER BY` cannot use column references that include a table name (that is, names in *`tbl_name.col_name`* format). Instead, provide a column alias in the first query block, and refer to the alias in the `ORDER BY` clause. (You can also refer to the column in the `ORDER BY` clause using its column position, but such use of column positions is deprecated, and thus subject to eventual removal in a future MySQL release.)

If a column to be sorted is aliased, the `ORDER BY` clause *must* refer to the alias, not the column name. The first of the following statements is permitted, but the second fails with an `Unknown column 'a' in 'order clause'` error:

```
(SELECT a AS b FROM t) UNION (SELECT ...) ORDER BY b;
(SELECT a AS b FROM t) UNION (SELECT ...) ORDER BY a;
```

To cause rows in a `UNION` result to consist of the sets of rows retrieved by each query block one after the other, select an additional column in each query block to use as a sort column and add an `ORDER BY` clause that sorts on that column following the last query block:

```
(SELECT 1 AS sort_col, col1a, col1b, ... FROM t1)
UNION
(SELECT 2, col2a, col2b, ... FROM t2) ORDER BY sort_col;
```

To maintain sort order within individual results, add a secondary column to the `ORDER BY` clause:

```
(SELECT 1 AS sort_col, col1a, col1b, ... FROM t1)
UNION
(SELECT 2, col2a, col2b, ... FROM t2) ORDER BY sort_col, col1a;
```

Use of an additional column also enables you to determine which query block each row comes from. Extra columns can provide other identifying information as well, such as a string that indicates a table name.

Limitations of Set Operations

Set operations in MySQL are subject to some limitations, which are described in the next few paragraphs.

Set operations including `SELECT` statements have the following limitations:

- `HIGH_PRIORITY` in the first `SELECT` has no effect. `HIGH_PRIORITY` in any subsequent `SELECT` produces a syntax error.
- Only the last `SELECT` statement can use an `INTO` clause. However, the entire `UNION` result is written to the `INTO` output destination.

As of MySQL 8.0.20, these two `UNION` variants containing `INTO` are deprecated; you should expect support for them to be removed in a future version of MySQL:

- In the trailing query block of a query expression, use of `INTO` before `FROM` produces a warning. Example:

  ```
  ... UNION SELECT * INTO OUTFILE 'file_name' FROM table_name;
  ```

- In a parenthesized trailing block of a query expression, use of `INTO` (regardless of its position relative to `FROM`) produces a warning. Example:

  ```
  ... UNION (SELECT * INTO OUTFILE 'file_name' FROM table_name);
  ```

  Those variants are deprecated because they are confusing, as if they collect information from the named table rather than the entire query expression (the `UNION`).

Set operations with an aggregate function in an `ORDER BY` clause are rejected with `ER_AGGREGATE_ORDER_FOR_UNION`. Although the error name might suggest that this is exclusive to `UNION` queries, the preceding is also true for `EXCEPT` and `INTERSECT` queries, as shown here:

```
mysql> TABLE t1 INTERSECT TABLE t2 ORDER BY MAX(x);
ERROR 3028 (HY000): Expression #1 of ORDER BY contains aggregate function and applies to a UNION, EXCEP
```

A locking clause (such as `FOR UPDATE` or `LOCK IN SHARE MODE`) applies to the query block it follows. This means that, in a `SELECT` statement used with set operations, a locking clause can be used only if the query block and locking clause are enclosed in parentheses.

13.2.15 Subqueries

A subquery is a `SELECT` statement within another statement.

All subquery forms and operations that the SQL standard requires are supported, as well as a few features that are MySQL-specific.

Here is an example of a subquery:

```
SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
```

In this example, `SELECT * FROM t1 ...` is the *outer query* (or *outer statement*), and `(SELECT column1 FROM t2)` is the *subquery*. We say that the subquery is *nested* within the outer query, and in fact it is possible to nest subqueries within other subqueries, to a considerable depth. A subquery must always appear within parentheses.

The main advantages of subqueries are:

- They allow queries that are *structured* so that it is possible to isolate each part of a statement.
- They provide alternative ways to perform operations that would otherwise require complex joins and unions.
- Many people find subqueries more readable than complex joins or unions. Indeed, it was the innovation of subqueries that gave people the original idea of calling the early SQL “Structured Query Language.”

Here is an example statement that shows the major points about subquery syntax as specified by the SQL standard and supported in MySQL:

```
DELETE FROM t1
WHERE s11 > ANY
 (SELECT COUNT(*) /* no hint */ FROM t2
  WHERE NOT EXISTS
   (SELECT * FROM t3
    WHERE ROW(5*t2.s1,77)=
     (SELECT 50,11*s1 FROM t4 UNION SELECT 50,77 FROM
```

```
      (SELECT * FROM t5) AS t5)));
```

A subquery can return a scalar (a single value), a single row, a single column, or a table (one or more rows of one or more columns). These are called scalar, column, row, and table subqueries. Subqueries that return a particular kind of result often can be used only in certain contexts, as described in the following sections.

There are few restrictions on the type of statements in which subqueries can be used. A subquery can contain many of the keywords or clauses that an ordinary `SELECT` can contain: `DISTINCT`, `GROUP BY`, `ORDER BY`, `LIMIT`, joins, index hints, `UNION` constructs, comments, functions, and so on.

Beginning with MySQL 8.0.19, `TABLE` and `VALUES` statements can be used in subqueries. Subqueries using `VALUES` are generally more verbose versions of subqueries that can be rewritten more compactly using set notation, or with `SELECT` or `TABLE` syntax; assuming that table `ts` is created using the statement `CREATE TABLE ts VALUES ROW(2), ROW(4), ROW(6)`, the statements shown here are all equivalent:

```
SELECT * FROM tt
    WHERE b > ANY (VALUES ROW(2), ROW(4), ROW(6));

SELECT * FROM tt
    WHERE b > ANY (SELECT * FROM ts);

SELECT * FROM tt
    WHERE b > ANY (TABLE ts);
```

Examples of `TABLE` subqueries are shown in the sections that follow.

A subquery's outer statement can be any one of: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `SET`, or `DO`.

For information about how the optimizer handles subqueries, see Section 8.2.2, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions”. For a discussion of restrictions on subquery use, including performance issues for certain forms of subquery syntax, see Section 13.2.15.12, “Restrictions on Subqueries”.

13.2.15.1 The Subquery as Scalar Operand

In its simplest form, a subquery is a scalar subquery that returns a single value. A scalar subquery is a simple operand, and you can use it almost anywhere a single column value or literal is legal, and you can expect it to have those characteristics that all operands have: a data type, a length, an indication that it can be `NULL`, and so on. For example:

```
CREATE TABLE t1 (s1 INT, s2 CHAR(5) NOT NULL);
INSERT INTO t1 VALUES(100, 'abcde');
SELECT (SELECT s2 FROM t1);
```

The subquery in this `SELECT` returns a single value (`'abcde'`) that has a data type of `CHAR`, a length of 5, a character set and collation equal to the defaults in effect at `CREATE TABLE` time, and an indication that the value in the column can be `NULL`. Nullability of the value selected by a scalar subquery is not copied because if the subquery result is empty, the result is `NULL`. For the subquery just shown, if `t1` were empty, the result would be `NULL` even though `s2` is `NOT NULL`.

There are a few contexts in which a scalar subquery cannot be used. If a statement permits only a literal value, you cannot use a subquery. For example, `LIMIT` requires literal integer arguments, and `LOAD DATA` requires a literal string file name. You cannot use subqueries to supply these values.

When you see examples in the following sections that contain the rather spartan construct `(SELECT column1 FROM t1)`, imagine that your own code contains much more diverse and complex constructions.

Suppose that we make two tables:

```
CREATE TABLE t1 (s1 INT);
INSERT INTO t1 VALUES (1);
```

```
CREATE TABLE t2 (s1 INT);
INSERT INTO t2 VALUES (2);
```

Then perform a `SELECT`:

```
SELECT (SELECT s1 FROM t2) FROM t1;
```

The result is `2` because there is a row in `t2` containing a column `s1` that has a value of `2`.

In MySQL 8.0.19 and later, the preceding query can also be written like this, using `TABLE`:

```
SELECT (TABLE t2) FROM t1;
```

A scalar subquery can be part of an expression, but remember the parentheses, even if the subquery is an operand that provides an argument for a function. For example:

```
SELECT UPPER((SELECT s1 FROM t1)) FROM t2;
```

The same result can be obtained in MySQL 8.0.19 and later using `SELECT UPPER((TABLE t1)) FROM t2`.

13.2.15.2 Comparisons Using Subqueries

The most common use of a subquery is in the form:

```
non_subquery_operand comparison_operator (subquery)
```

Where *`comparison_operator`* is one of these operators:

```
=  >  <  >=  <=  <>  !=  <=>
```

For example:

```
... WHERE 'a' = (SELECT column1 FROM t1)
```

MySQL also permits this construct:

```
non_subquery_operand LIKE (subquery)
```

At one time the only legal place for a subquery was on the right side of a comparison, and you might still find some old DBMSs that insist on this.

Here is an example of a common-form subquery comparison that you cannot do with a join. It finds all the rows in table `t1` for which the `column1` value is equal to a maximum value in table `t2`:

```
SELECT * FROM t1
  WHERE column1 = (SELECT MAX(column2) FROM t2);
```

Here is another example, which again is impossible with a join because it involves aggregating for one of the tables. It finds all rows in table `t1` containing a value that occurs twice in a given column:

```
SELECT * FROM t1 AS t
  WHERE 2 = (SELECT COUNT(*) FROM t1 WHERE t1.id = t.id);
```

For a comparison of the subquery to a scalar, the subquery must return a scalar. For a comparison of the subquery to a row constructor, the subquery must be a row subquery that returns a row with the same number of values as the row constructor. See Section 13.2.15.5, “Row Subqueries”.

13.2.15.3 Subqueries with ANY, IN, or SOME

Syntax:

```
operand comparison_operator ANY (subquery)
operand IN (subquery)
operand comparison_operator SOME (subquery)
```

Where *`comparison_operator`* is one of these operators:

```
= > < >= <= <> !=
```

The ANY keyword, which must follow a comparison operator, means “return TRUE if the comparison is TRUE for ANY of the values in the column that the subquery returns.” For example:

```
SELECT s1 FROM t1 WHERE s1 > ANY (SELECT s1 FROM t2);
```

Suppose that there is a row in table t1 containing (10). The expression is TRUE if table t2 contains (21,14,7) because there is a value 7 in t2 that is less than 10. The expression is FALSE if table t2 contains (20,10), or if table t2 is empty. The expression is *unknown* (that is, NULL) if table t2 contains (NULL,NULL,NULL).

When used with a subquery, the word IN is an alias for = ANY. Thus, these two statements are the same:

```
SELECT s1 FROM t1 WHERE s1 = ANY (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 IN    (SELECT s1 FROM t2);
```

IN and = ANY are not synonyms when used with an expression list. IN can take an expression list, but = ANY cannot. See Section 12.4.2, “Comparison Functions and Operators”.

NOT IN is not an alias for <> ANY, but for <> ALL. See Section 13.2.15.4, “Subqueries with ALL”.

The word SOME is an alias for ANY. Thus, these two statements are the same:

```
SELECT s1 FROM t1 WHERE s1 <> ANY  (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 <> SOME (SELECT s1 FROM t2);
```

Use of the word SOME is rare, but this example shows why it might be useful. To most people, the English phrase “a is not equal to any b” means “there is no b which is equal to a,” but that is not what is meant by the SQL syntax. The syntax means “there is some b to which a is not equal.” Using <> SOME instead helps ensure that everyone understands the true meaning of the query.

Beginning with MySQL 8.0.19, you can use TABLE in a scalar IN, ANY, or SOME subquery provided the table contains only a single column. If t2 has only one column, the statements shown previously in this section can be written as shown here, in each case substituting TABLE t2 for SELECT s1 FROM t2:

```
SELECT s1 FROM t1 WHERE s1 > ANY (TABLE t2);

SELECT s1 FROM t1 WHERE s1 = ANY (TABLE t2);

SELECT s1 FROM t1 WHERE s1 IN (TABLE t2);

SELECT s1 FROM t1 WHERE s1 <> ANY  (TABLE t2);

SELECT s1 FROM t1 WHERE s1 <> SOME (TABLE t2);
```

13.2.15.4 Subqueries with ALL

Syntax:

```
operand comparison_operator ALL (subquery)
```

The word ALL, which must follow a comparison operator, means “return TRUE if the comparison is TRUE for ALL of the values in the column that the subquery returns.” For example:

```
SELECT s1 FROM t1 WHERE s1 > ALL (SELECT s1 FROM t2);
```

Suppose that there is a row in table t1 containing (10). The expression is TRUE if table t2 contains (-5,0,+5) because 10 is greater than all three values in t2. The expression is FALSE if table t2 contains (12,6,NULL,-100) because there is a single value 12 in table t2 that is greater than 10. The expression is *unknown* (that is, NULL) if table t2 contains (0,NULL,1).

Finally, the expression is TRUE if table t2 is empty. So, the following expression is TRUE when table t2 is empty:

```
SELECT * FROM t1 WHERE 1 > ALL (SELECT s1 FROM t2);
```

But this expression is `NULL` when table `t2` is empty:

```
SELECT * FROM t1 WHERE 1 > (SELECT s1 FROM t2);
```

In addition, the following expression is `NULL` when table `t2` is empty:

```
SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2);
```

In general, *tables containing `NULL` values* and *empty tables* are “edge cases.” When writing subqueries, always consider whether you have taken those two possibilities into account.

`NOT IN` is an alias for `<> ALL`. Thus, these two statements are the same:

```
SELECT s1 FROM t1 WHERE s1 <> ALL (SELECT s1 FROM t2);
SELECT s1 FROM t1 WHERE s1 NOT IN (SELECT s1 FROM t2);
```

MySQL 8.0.19 supports the `TABLE` statement. As with `IN`, `ANY`, and `SOME`, you can use `TABLE` with `ALL` and `NOT IN` provided that the following two conditions are met:

- The table in the subquery contains only one column
- The subquery does not depend on a column expression

For example, assuming that table `t2` consists of a single column, the last two statements shown previously can be written using `TABLE t2` like this:

```
SELECT s1 FROM t1 WHERE s1 <> ALL (TABLE t2);
SELECT s1 FROM t1 WHERE s1 NOT IN (TABLE t2);
```

A query such as `SELECT * FROM t1 WHERE 1 > ALL (SELECT MAX(s1) FROM t2);` cannot be written using `TABLE t2` because the subquery depends on a column expression.

13.2.15.5 Row Subqueries

Scalar or column subqueries return a single value or a column of values. A *row subquery* is a subquery variant that returns a single row and can thus return more than one column value. Legal operators for row subquery comparisons are:

```
=  >  <  >=  <=  <>  !=  <=>
```

Here are two examples:

```
SELECT * FROM t1
  WHERE (col1,col2) = (SELECT col3, col4 FROM t2 WHERE id = 10);
SELECT * FROM t1
  WHERE ROW(col1,col2) = (SELECT col3, col4 FROM t2 WHERE id = 10);
```

For both queries, if the table `t2` contains a single row with `id = 10`, the subquery returns a single row. If this row has `col3` and `col4` values equal to the `col1` and `col2` values of any rows in `t1`, the `WHERE` expression is `TRUE` and each query returns those `t1` rows. If the `t2` row `col3` and `col4` values are not equal the `col1` and `col2` values of any `t1` row, the expression is `FALSE` and the query returns an empty result set. The expression is *unknown* (that is, `NULL`) if the subquery produces no rows. An error occurs if the subquery produces multiple rows because a row subquery can return at most one row.

For information about how each operator works for row comparisons, see Section 12.4.2, “Comparison Functions and Operators”.

The expressions `(1,2)` and `ROW(1,2)` are sometimes called *row constructors*. The two are equivalent. The row constructor and the row returned by the subquery must contain the same number of values.

A row constructor is used for comparisons with subqueries that return two or more columns. When a subquery returns a single column, this is regarded as a scalar value and not as a row, so a row constructor cannot be used with a subquery that does not return at least two columns. Thus, the following query fails with a syntax error:

```
SELECT * FROM t1 WHERE ROW(1) = (SELECT column1 FROM t2)
```

Row constructors are legal in other contexts. For example, the following two statements are semantically equivalent (and are handled in the same way by the optimizer):

```
SELECT * FROM t1 WHERE (column1,column2) = (1,1);
SELECT * FROM t1 WHERE column1 = 1 AND column2 = 1;
```

The following query answers the request, “find all rows in table `t1` that also exist in table `t2`”:

```
SELECT column1,column2,column3
  FROM t1
  WHERE (column1,column2,column3) IN
         (SELECT column1,column2,column3 FROM t2);
```

For more information about the optimizer and row constructors, see Section 8.2.1.22, “Row Constructor Expression Optimization”

13.2.15.6 Subqueries with EXISTS or NOT EXISTS

If a subquery returns any rows at all, `EXISTS subquery` is `TRUE`, and `NOT EXISTS subquery` is `FALSE`. For example:

```
SELECT column1 FROM t1 WHERE EXISTS (SELECT * FROM t2);
```

Traditionally, an `EXISTS` subquery starts with `SELECT *`, but it could begin with `SELECT 5` or `SELECT column1` or anything at all. MySQL ignores the `SELECT` list in such a subquery, so it makes no difference.

For the preceding example, if `t2` contains any rows, even rows with nothing but `NULL` values, the `EXISTS` condition is `TRUE`. This is actually an unlikely example because a `[NOT] EXISTS` subquery almost always contains correlations. Here are some more realistic examples:

- What kind of store is present in one or more cities?

  ```
  SELECT DISTINCT store_type FROM stores
    WHERE EXISTS (SELECT * FROM cities_stores
                  WHERE cities_stores.store_type = stores.store_type);
  ```

- What kind of store is present in no cities?

  ```
  SELECT DISTINCT store_type FROM stores
    WHERE NOT EXISTS (SELECT * FROM cities_stores
                      WHERE cities_stores.store_type = stores.store_type);
  ```

- What kind of store is present in all cities?

  ```
  SELECT DISTINCT store_type FROM stores s1
    WHERE NOT EXISTS (
      SELECT * FROM cities WHERE NOT EXISTS (
        SELECT * FROM cities_stores
         WHERE cities_stores.city = cities.city
         AND cities_stores.store_type = stores.store_type));
  ```

The last example is a double-nested `NOT EXISTS` query. That is, it has a `NOT EXISTS` clause within a `NOT EXISTS` clause. Formally, it answers the question “does a city exist with a store that is not in `Stores`”? But it is easier to say that a nested `NOT EXISTS` answers the question “is *`x`* `TRUE` for all *`y`*?”

In MySQL 8.0.19 and later, you can also use `NOT EXISTS` or `NOT EXISTS` with `TABLE` in the subquery, like this:

```
SELECT column1 FROM t1 WHERE EXISTS (TABLE t2);
```

The results are the same as when using `SELECT *` with no `WHERE` clause in the subquery.

13.2.15.7 Correlated Subqueries

A *correlated subquery* is a subquery that contains a reference to a table that also appears in the outer query. For example:

```
SELECT * FROM t1
  WHERE column1 = ANY (SELECT column1 FROM t2
                       WHERE t2.column2 = t1.column2);
```

Notice that the subquery contains a reference to a column of `t1`, even though the subquery's `FROM` clause does not mention a table `t1`. So, MySQL looks outside the subquery, and finds `t1` in the outer query.

Suppose that table `t1` contains a row where `column1 = 5` and `column2 = 6`; meanwhile, table `t2` contains a row where `column1 = 5` and `column2 = 7`. The simple expression `... WHERE column1 = ANY (SELECT column1 FROM t2)` would be `TRUE`, but in this example, the `WHERE` clause within the subquery is `FALSE` (because `(5,6)` is not equal to `(5,7)`), so the expression as a whole is `FALSE`.

Scoping rule: MySQL evaluates from inside to outside. For example:

```
SELECT column1 FROM t1 AS x
  WHERE x.column1 = (SELECT column1 FROM t2 AS x
    WHERE x.column1 = (SELECT column1 FROM t3
      WHERE x.column2 = t3.column1));
```

In this statement, `x.column2` must be a column in table `t2` because `SELECT column1 FROM t2 AS x ...` renames `t2`. It is not a column in table `t1` because `SELECT column1 FROM t1 ...` is an outer query that is *farther out*.

Beginning with MySQL 8.0.24, the optimizer can transform a correlated scalar subquery to a derived table when the `subquery_to_derived` flag of the `optimizer_switch` variable is enabled. Consider the query shown here:

```
SELECT * FROM t1
    WHERE ( SELECT a FROM t2
              WHERE t2.a=t1.a ) > 0;
```

To avoid materializing several times for a given derived table, we can instead materialize—once—a derived table which adds a grouping on the join column from the table referenced in the inner query (`t2.a`) and then an outer join on the lifted predicate (`t1.a = derived.a`) in order to select the correct group to match up with the outer row. (If the subquery already has an explicit grouping, the extra grouping is added to the end of the grouping list.) The query previously shown can thus be rewritten like this:

```
SELECT t1.* FROM t1
    LEFT OUTER JOIN
        (SELECT a, COUNT(*) AS ct FROM t2 GROUP BY a) AS derived
    ON  t1.a = derived.a
        AND
        REJECT_IF(
            (ct > 1),
            "ERROR 1242 (21000): Subquery returns more than 1 row"
            )
    WHERE derived.a > 0;
```

In the rewritten query, `REJECT_IF()` represents an internal function which tests a given condition (here, the comparison `ct > 1`) and raises a given error (in this case, `ER_SUBQUERY_NO_1_ROW`) if the condition is true. This reflects the cardinality check that the optimizer performs as part of evaluating the `JOIN` or `WHERE` clause, prior to evaluating any lifted predicate, which is done only if the subquery does not return more than one row.

This type of transformation can be performed, provided the following conditions are met:

- The subquery can be part of a `SELECT` list, `WHERE` condition, or `HAVING` condition, but cannot be part of a `JOIN` condition, and cannot contain a `LIMIT` or `OFFSET` clause. In addition, the subquery cannot contain any set operations such as `UNION`.
- The `WHERE` clause may contain one or more predicates, combined with `AND`. If the `WHERE` clause contains an `OR` clause, it cannot be transformed. At least one of the `WHERE` clause predicates must be eligible for transformation, and none of them may reject transformation.
- To be eligible for transformation, a `WHERE` clause predicate must be an equality predicate in which each operand should be a simple column reference. No other predicates—including other comparison predicates—are eligible for transformation. The predicate must employ the equality operator `=` for making the comparison; the null-safe `<=>` operator is not supported in this context.
- A `WHERE` clause predicate that contains only inner references is not eligible for transformation, since it can be evaluated before the grouping. A `WHERE` clause predicate that contains only outer references is eligible for transformation, even though it can be lifted up to the outer query block. This is made possible by adding a cardinality check without grouping in the derived table.
- To be eligible, a `WHERE` clause predicate must have one operand that contains only inner references and one operand that contains only outer references. If the predicate is not eligible due to this rule, transformation of the query is rejected.
- A correlated column can be present only in the subquery's `WHERE` clause (and not in the `SELECT` list, a `JOIN` or `ORDER BY` clause, a `GROUP BY` list, or a `HAVING` clause). Nor can there be any correlated column inside a derived table in the subquery's `FROM` list.
- A correlated column can not be contained in an aggregate function's list of arguments.
- A correlated column must be resolved in the query block directly containing the subquery being considered for transformation.
- A correlated column cannot be present in a nested scalar subquery in the `WHERE` clause.
- The subquery cannot contain any window functions, and must not contain any aggregate function which aggregates in a query block outer to the subquery. A `COUNT()` aggregate function, if contained in the `SELECT` list element of the subquery, must be at the topmost level, and cannot be part of an expression.

See also Section 13.2.15.8, “Derived Tables”.

13.2.15.8 Derived Tables

This section discusses general characteristics of derived tables. For information about lateral derived tables preceded by the `LATERAL` keyword, see Section 13.2.15.9, “Lateral Derived Tables”.

A derived table is an expression that generates a table within the scope of a query `FROM` clause. For example, a subquery in a `SELECT` statement `FROM` clause is a derived table:

```
SELECT ... FROM (subquery) [AS] tbl_name ...
```

The `JSON_TABLE()` function generates a table and provides another way to create a derived table:

```
SELECT * FROM JSON_TABLE(arg_list) [AS] tbl_name ...
```

The `[AS] tbl_name` clause is mandatory because every table in a `FROM` clause must have a name. Any columns in the derived table must have unique names. Alternatively, *`tbl_name`* may be followed by a parenthesized list of names for the derived table columns:

```
SELECT ... FROM (subquery) [AS] tbl_name (col_list) ...
```

The number of column names must be the same as the number of table columns.

For the sake of illustration, assume that you have this table:

```
CREATE TABLE t1 (s1 INT, s2 CHAR(5), s3 FLOAT);
```

Here is how to use a subquery in the `FROM` clause, using the example table:

```
INSERT INTO t1 VALUES (1,'1',1.0);
INSERT INTO t1 VALUES (2,'2',2.0);
SELECT sb1,sb2,sb3
  FROM (SELECT s1 AS sb1, s2 AS sb2, s3*2 AS sb3 FROM t1) AS sb
  WHERE sb1 > 1;
```

Result:

```
+------+------+------+
| sb1  | sb2  | sb3  |
+------+------+------+
|    2 | 2    |    4 |
+------+------+------+
```

Here is another example: Suppose that you want to know the average of a set of sums for a grouped table. This does not work:

```
SELECT AVG(SUM(column1)) FROM t1 GROUP BY column1;
```

However, this query provides the desired information:

```
SELECT AVG(sum_column1)
  FROM (SELECT SUM(column1) AS sum_column1
        FROM t1 GROUP BY column1) AS t1;
```

Notice that the column name used within the subquery (`sum_column1`) is recognized in the outer query.

The column names for a derived table come from its select list:

```
mysql> SELECT * FROM (SELECT 1, 2, 3, 4) AS dt;
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
```

To provide column names explicitly, follow the derived table name with a parenthesized list of column names:

```
mysql> SELECT * FROM (SELECT 1, 2, 3, 4) AS dt (a, b, c, d);
+---+---+---+---+
| a | b | c | d |
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
```

A derived table can return a scalar, column, row, or table.

Derived tables are subject to these restrictions:

- A derived table cannot contain references to other tables of the same `SELECT` (use a `LATERAL` derived table for that; see Section 13.2.15.9, “Lateral Derived Tables”).
- Prior to MySQL 8.0.14, a derived table cannot contain outer references. This is a MySQL restriction that is lifted in MySQL 8.0.14, not a restriction of the SQL standard. For example, the derived table `dt` in the following query contains a reference `t1.b` to the table `t1` in the outer query:

  ```
  SELECT * FROM t1
  WHERE t1.d > (SELECT AVG(dt.a)
                  FROM (SELECT SUM(t2.a) AS a
                        FROM t2
  ```

```
                      WHERE t2.b = t1.b GROUP BY t2.c) dt
              WHERE dt.a > 10);
```

The query is valid in MySQL 8.0.14 and higher. Before 8.0.14, it produces an error: `Unknown column 't1.b' in 'where clause'`

The optimizer determines information about derived tables in such a way that `EXPLAIN` does not need to materialize them. See Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization”.

It is possible under certain circumstances that using `EXPLAIN SELECT` modifies table data. This can occur if the outer query accesses any tables and an inner query invokes a stored function that changes one or more rows of a table. Suppose that there are two tables `t1` and `t2` in database `d1`, and a stored function `f1` that modifies `t2`, created as shown here:

```
CREATE DATABASE d1;
USE d1;
CREATE TABLE t1 (c1 INT);
CREATE TABLE t2 (c1 INT);
CREATE FUNCTION f1(p1 INT) RETURNS INT
  BEGIN
    INSERT INTO t2 VALUES (p1);
    RETURN p1;
  END;
```

Referencing the function directly in an `EXPLAIN SELECT` has no effect on `t2`, as shown here:

```
mysql> SELECT * FROM t2;
Empty set (0.02 sec)

mysql> EXPLAIN SELECT f1(5)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
   partitions: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra: No tables used
1 row in set (0.01 sec)

mysql> SELECT * FROM t2;
Empty set (0.01 sec)
```

This is because the `SELECT` statement did not reference any tables, as can be seen in the `table` and `Extra` columns of the output. This is also true of the following nested `SELECT`:

```
mysql> EXPLAIN SELECT NOW() AS a1, (SELECT f1(5)) AS a2\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra: No tables used
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+-------+------+-----------------------------------------+
```

```
| Level | Code | Message                                      |
+-------+------+----------------------------------------------+
| Note  | 1249 | Select 2 was reduced during optimization |
+-------+------+----------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

However, if the outer `SELECT` references any tables, the optimizer executes the statement in the subquery as well, with the result that `t2` is modified:

```
mysql> EXPLAIN SELECT * FROM t1 AS a1, (SELECT f1(5)) AS a2\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: <derived2>
   partitions: NULL
         type: system
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 2. row ***************************
           id: 1
  select_type: PRIMARY
        table: a1
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1
     filtered: 100.00
        Extra: NULL
*************************** 3. row ***************************
           id: 2
  select_type: DERIVED
        table: NULL
   partitions: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
     filtered: NULL
        Extra: No tables used
3 rows in set (0.00 sec)

mysql> SELECT * FROM t2;
+------+
| c1   |
+------+
|    5 |
+------+
1 row in set (0.00 sec)
```

The derived table optimization can also be employed with many correlated (scalar) subqueries (MySQL 8.0.24 and later). For more information and examples, see Section 13.2.15.7, “Correlated Subqueries”.

13.2.15.9 Lateral Derived Tables

A derived table cannot normally refer to (depend on) columns of preceding tables in the same `FROM` clause. As of MySQL 8.0.14, a derived table may be defined as a lateral derived table to specify that such references are permitted.

Nonlateral derived tables are specified using the syntax discussed in Section 13.2.15.8, “Derived Tables”. The syntax for a lateral derived table is the same as for a nonlateral derived table except that the keyword `LATERAL` is specified before the derived table specification. The `LATERAL` keyword must precede each table to be used as a lateral derived table.

Lateral derived tables are subject to these restrictions:

- A lateral derived table can occur only in a `FROM` clause, either in a list of tables separated with commas or in a join specification (`JOIN`, `INNER JOIN`, `CROSS JOIN`, `LEFT [OUTER] JOIN`, or `RIGHT [OUTER] JOIN`).
- If a lateral derived table is in the right operand of a join clause and contains a reference to the left operand, the join operation must be an `INNER JOIN`, `CROSS JOIN`, or `LEFT [OUTER] JOIN`.

  If the table is in the left operand and contains a reference to the right operand, the join operation must be an `INNER JOIN`, `CROSS JOIN`, or `RIGHT [OUTER] JOIN`.
- If a lateral derived table references an aggregate function, the function's aggregation query cannot be the one that owns the `FROM` clause in which the lateral derived table occurs.
- In accordance with the SQL standard, MySQL always treats a join with a table function such as `JSON_TABLE()` as though `LATERAL` had been used. This is true regardless of MySQL release version, which is why it is possible to join against this function even in MySQL versions prior to 8.0.14. In MySQL 8.0.14 and later, the `LATERAL` keyword is implicit, and is not allowed before `JSON_TABLE()`. This is also according to the SQL standard.

The following discussion shows how lateral derived tables make possible certain SQL operations that cannot be done with nonlateral derived tables or that require less-efficient workarounds.

Suppose that we want to solve this problem: Given a table of people in a sales force (where each row describes a member of the sales force), and a table of all sales (where each row describes a sale: salesperson, customer, amount, date), determine the size and customer of the largest sale for each salesperson. This problem can be approached two ways.

First approach to solving the problem: For each salesperson, calculate the maximum sale size, and also find the customer who provided this maximum. In MySQL, that can be done like this:

```
SELECT
  salesperson.name,
  -- find maximum sale size for this salesperson
  (SELECT MAX(amount) AS amount
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id)
  AS amount,
  -- find customer for this maximum size
  (SELECT customer_name
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id
    AND all_sales.amount =
         -- find maximum size, again
         (SELECT MAX(amount) AS amount
           FROM all_sales
           WHERE all_sales.salesperson_id = salesperson.id))
  AS customer_name
FROM
  salesperson;
```

That query is inefficient because it calculates the maximum size twice per salesperson (once in the first subquery and once in the second).

We can try to achieve an efficiency gain by calculating the maximum once per salesperson and “caching” it in a derived table, as shown by this modified query:

```
SELECT
  salesperson.name,
```

```
  max_sale.amount,
  max_sale_customer.customer_name
FROM
  salesperson,
  -- calculate maximum size, cache it in transient derived table max_sale
  (SELECT MAX(amount) AS amount
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id)
  AS max_sale,
  -- find customer, reusing cached maximum size
  (SELECT customer_name
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id
    AND all_sales.amount =
        -- the cached maximum size
        max_sale.amount)
  AS max_sale_customer;
```

However, the query is illegal in SQL-92 because derived tables cannot depend on other tables in the same `FROM` clause. Derived tables must be constant over the query's duration, not contain references to columns of other `FROM` clause tables. As written, the query produces this error:

```
ERROR 1054 (42S22): Unknown column 'salesperson.id' in 'where clause'
```

In SQL:1999, the query becomes legal if the derived tables are preceded by the `LATERAL` keyword (which means “this derived table depends on previous tables on its left side”):

```
SELECT
  salesperson.name,
  max_sale.amount,
  max_sale_customer.customer_name
FROM
  salesperson,
  -- calculate maximum size, cache it in transient derived table max_sale
  LATERAL
  (SELECT MAX(amount) AS amount
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id)
  AS max_sale,
  -- find customer, reusing cached maximum size
  LATERAL
  (SELECT customer_name
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id
    AND all_sales.amount =
        -- the cached maximum size
        max_sale.amount)
  AS max_sale_customer;
```

A lateral derived table need not be constant and is brought up to date each time a new row from a preceding table on which it depends is processed by the top query.

Second approach to solving the problem: A different solution could be used if a subquery in the `SELECT` list could return multiple columns:

```
SELECT
  salesperson.name,
  -- find maximum size and customer at same time
  (SELECT amount, customer_name
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id
    ORDER BY amount DESC LIMIT 1)
FROM
  salesperson;
```

That is efficient but illegal. It does not work because such subqueries can return only a single column:

```
ERROR 1241 (21000): Operand should contain 1 column(s)
```

One attempt at rewriting the query is to select multiple columns from a derived table:

```
SELECT
  salesperson.name,
  max_sale.amount,
  max_sale.customer_name
FROM
  salesperson,
  -- find maximum size and customer at same time
  (SELECT amount, customer_name
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id
    ORDER BY amount DESC LIMIT 1)
  AS max_sale;
```

However, that also does not work. The derived table is dependent on the `salesperson` table and thus fails without `LATERAL`:

```
ERROR 1054 (42S22): Unknown column 'salesperson.id' in 'where clause'
```

Adding the `LATERAL` keyword makes the query legal:

```
SELECT
  salesperson.name,
  max_sale.amount,
  max_sale.customer_name
FROM
  salesperson,
  -- find maximum size and customer at same time
  LATERAL
  (SELECT amount, customer_name
    FROM all_sales
    WHERE all_sales.salesperson_id = salesperson.id
    ORDER BY amount DESC LIMIT 1)
  AS max_sale;
```

In short, `LATERAL` is the efficient solution to all drawbacks in the two approaches just discussed.

13.2.15.10 Subquery Errors

There are some errors that apply only to subqueries. This section describes them.

- Unsupported subquery syntax:

  ```
  ERROR 1235 (ER_NOT_SUPPORTED_YET)
  SQLSTATE = 42000
  Message = "This version of MySQL doesn't yet support
  'LIMIT & IN/ALL/ANY/SOME subquery'"
  ```

  This means that MySQL does not support statements like the following:

  ```
  SELECT * FROM t1 WHERE s1 IN (SELECT s2 FROM t2 ORDER BY s1 LIMIT 1)
  ```

- Incorrect number of columns from subquery:

  ```
  ERROR 1241 (ER_OPERAND_COL)
  SQLSTATE = 21000
  Message = "Operand should contain 1 column(s)"
  ```

  This error occurs in cases like this:

  ```
  SELECT (SELECT column1, column2 FROM t2) FROM t1;
  ```

  You may use a subquery that returns multiple columns, if the purpose is row comparison. In other contexts, the subquery must be a scalar operand. See Section 13.2.15.5, “Row Subqueries”.

- Incorrect number of rows from subquery:

  ```
  ERROR 1242 (ER_SUBSELECT_NO_1_ROW)
  SQLSTATE = 21000
  Message = "Subquery returns more than 1 row"
  ```

This error occurs for statements where the subquery must return at most one row but returns multiple rows. Consider the following example:

```
SELECT * FROM t1 WHERE column1 = (SELECT column1 FROM t2);
```

If `SELECT column1 FROM t2` returns just one row, the previous query works. If the subquery returns more than one row, error 1242 occurs. In that case, the query should be rewritten as:

```
SELECT * FROM t1 WHERE column1 = ANY (SELECT column1 FROM t2);
```

- Incorrectly used table in subquery:

```
Error 1093 (ER_UPDATE_TABLE_USED)
SQLSTATE = HY000
Message = "You can't specify target table 'x'
for update in FROM clause"
```

This error occurs in cases such as the following, which attempts to modify a table and select from the same table in the subquery:

```
UPDATE t1 SET column2 = (SELECT MAX(column1) FROM t1);
```

You can use a common table expression or derived table to work around this. See Section 13.2.15.12, “Restrictions on Subqueries”.

In MySQL 8.0.19 and later, all of the errors described in this section also apply when using `TABLE` in subqueries.

For transactional storage engines, the failure of a subquery causes the entire statement to fail. For nontransactional storage engines, data modifications made before the error was encountered are preserved.

13.2.15.11 Optimizing Subqueries

Development is ongoing, so no optimization tip is reliable for the long term. The following list provides some interesting tricks that you might want to play with. See also Section 8.2.2, “Optimizing Subqueries, Derived Tables, View References, and Common Table Expressions”.

- Move clauses from outside to inside the subquery. For example, use this query:

```
SELECT * FROM t1
  WHERE s1 IN (SELECT s1 FROM t1 UNION ALL SELECT s1 FROM t2);
```

Instead of this query:

```
SELECT * FROM t1
  WHERE s1 IN (SELECT s1 FROM t1) OR s1 IN (SELECT s1 FROM t2);
```

For another example, use this query:

```
SELECT (SELECT column1 + 5 FROM t1) FROM t2;
```

Instead of this query:

```
SELECT (SELECT column1 FROM t1) + 5 FROM t2;
```

13.2.15.12 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 Section 8.2.2.4, “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 Section 8.9.3, “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 Section 8.2.2.4, “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 Section 13.2.15.10, “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.

13.2.16 TABLE Statement

`TABLE` is a DML statement introduced in MySQL 8.0.19 which returns rows and columns of the named table.

```
TABLE table_name [ORDER BY column_name] [LIMIT number [OFFSET number]]
```

The `TABLE` statement in some ways acts like `SELECT`. Given the existence of a table named `t`, the following two statements produce identical output:

```
TABLE t;

SELECT * FROM t;
```

You can order and limit the number of rows produced by `TABLE` using `ORDER BY` and `LIMIT` clauses, respectively. These function identically to the same clauses when used with `SELECT` (including an optional `OFFSET` clause with `LIMIT`), as you can see here:

```
mysql> TABLE t;
+----+----+
| a  | b  |
+----+----+
|  1 |  2 |
|  6 |  7 |
|  9 |  5 |
| 10 | -4 |
| 11 | -1 |
| 13 |  3 |
| 14 |  6 |
+----+----+
7 rows in set (0.00 sec)

mysql> TABLE t ORDER BY b;
+----+----+
| a  | b  |
+----+----+
| 10 | -4 |
| 11 | -1 |
|  1 |  2 |
| 13 |  3 |
|  9 |  5 |
| 14 |  6 |
|  6 |  7 |
+----+----+
7 rows in set (0.00 sec)

mysql> TABLE t LIMIT 3;
+---+---+
| a | b |
+---+---+
| 1 | 2 |
| 6 | 7 |
| 9 | 5 |
+---+---+
3 rows in set (0.00 sec)

mysql> TABLE t ORDER BY b LIMIT 3;
+----+----+
| a  | b  |
+----+----+
| 10 | -4 |
| 11 | -1 |
|  1 |  2 |
+----+----+
3 rows in set (0.00 sec)

mysql> TABLE t ORDER BY b LIMIT 3 OFFSET 2;
+----+----+
| a  | b  |
+----+----+
|  1 |  2 |
| 13 |  3 |
|  9 |  5 |
```

```
+----+----+
3 rows in set (0.00 sec)
```

`TABLE` differs from `SELECT` in two key respects:

- `TABLE` always displays all columns of the table.

  Exception: The output of `TABLE` does *not* include invisible columns. See Section 13.1.20.10, “Invisible Columns”.

- `TABLE` does not allow for any arbitrary filtering of rows; that is, `TABLE` does not support any `WHERE` clause.

For limiting which table columns are returned, filtering rows beyond what can be accomplished using `ORDER BY` and `LIMIT`, or both, use `SELECT`.

`TABLE` can be used with temporary tables.

`TABLE` can also be used in place of `SELECT` in a number of other constructs, including those listed here:

- With set operators such as `UNION`, as shown here:

```
mysql> TABLE t1;
+---+----+
| a | b  |
+---+----+
| 2 | 10 |
| 5 |  3 |
| 7 |  8 |
+---+----+
3 rows in set (0.00 sec)

mysql> TABLE t2;
+---+---+
| a | b |
+---+---+
| 1 | 2 |
| 3 | 4 |
| 6 | 7 |
+---+---+
3 rows in set (0.00 sec)

mysql> TABLE t1 UNION TABLE t2;
+---+----+
| a | b  |
+---+----+
| 2 | 10 |
| 5 |  3 |
| 7 |  8 |
| 1 |  2 |
| 3 |  4 |
| 6 |  7 |
+---+----+
6 rows in set (0.00 sec)
```

The `UNION` just shown is equivalent to the following statement:

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2;
+---+----+
| a | b  |
+---+----+
| 2 | 10 |
| 5 |  3 |
| 7 |  8 |
| 1 |  2 |
| 3 |  4 |
| 6 |  7 |
+---+----+
```

```
6 rows in set (0.00 sec)
```

`TABLE` can also be used together in set operations with `SELECT` statements, `VALUES` statements, or both. See Section 13.2.18, “UNION Clause”, Section 13.2.4, “EXCEPT Clause”, and Section 13.2.8, “INTERSECT Clause”, for more information and examples. See also Section 13.2.14, “Set Operations with UNION, INTERSECT, and EXCEPT”.

- With `INTO` to populate user variables, and with `INTO OUTFILE` or `INTO DUMPFILE` to write table data to a file. See Section 13.2.13.1, “SELECT ... INTO Statement”, for more specific information and examples.
- In many cases where you can employ subqueries. Given any table `t1` with a column named `a`, and a second table `t2` having a single column, statements such as the following are possible:

```
SELECT * FROM t1 WHERE a IN (TABLE t2);
```

Assuming that the single column of table `t1` is named `x`, the preceding is equivalent to each of the statements shown here (and produces exactly the same result in either case):

```
SELECT * FROM t1 WHERE a IN (SELECT x FROM t2);

SELECT * FROM t1 WHERE a IN (SELECT * FROM t2);
```

See Section 13.2.15, “Subqueries”, for more information.

- With `INSERT` and `REPLACE` statements, where you would otherwise use `SELECT *`. See Section 13.2.7.1, “INSERT ... SELECT Statement”, for more information and examples.
- `TABLE` can also be used in many cases in place of the `SELECT` in `CREATE TABLE ... SELECT` or `CREATE VIEW ... SELECT`. See the descriptions of these statements for more information and examples.

13.2.17 UPDATE Statement

`UPDATE` is a DML statement that modifies rows in a table.

An `UPDATE` statement can start with a `WITH` clause to define common table expressions accessible within the `UPDATE`. See Section 13.2.20, “WITH (Common Table Expressions)”.

Single-table syntax:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_reference
    SET assignment_list
    [WHERE where_condition]
    [ORDER BY ...]
    [LIMIT row_count]

value:
    {expr | DEFAULT}

assignment:
    col_name = value

assignment_list:
    assignment [, assignment] ...
```

Multiple-table syntax:

```
UPDATE [LOW_PRIORITY] [IGNORE] table_references
    SET assignment_list
    [WHERE where_condition]
```

For the single-table syntax, the `UPDATE` statement updates columns of existing rows in the named table with new values. The `SET` clause indicates which columns to modify and the values they should be given. Each value can be given as an expression, or the keyword `DEFAULT` to set a column

explicitly to its default value. The `WHERE` clause, if given, specifies the conditions that identify which rows to update. With no `WHERE` clause, all rows are updated. If the `ORDER BY` clause is specified, the rows are updated in the order that is specified. The `LIMIT` clause places a limit on the number of rows that can be updated.

For the multiple-table syntax, `UPDATE` updates rows in each table named in *`table_references`* that satisfy the conditions. Each matching row is updated once, even if it matches the conditions multiple times. For multiple-table syntax, `ORDER BY` and `LIMIT` cannot be used.

For partitioned tables, both the single-single and multiple-table forms of this statement support the use of a `PARTITION` clause as part of a table reference. This option takes a list of one or more partitions or subpartitions (or both). Only the partitions (or subpartitions) listed are checked for matches, and a row that is not in any of these partitions or subpartitions is not updated, whether it satisfies the *`where_condition`* or not.

Note

Unlike the case when using `PARTITION` with an `INSERT` or `REPLACE` statement, an otherwise valid `UPDATE ... PARTITION` statement is considered successful even if no rows in the listed partitions (or subpartitions) match the *`where_condition`*.

For more information and examples, see Section 24.5, “Partition Selection”.

`where_condition` is an expression that evaluates to true for each row to be updated. For expression syntax, see Section 9.5, “Expressions”.

`table_references` and *`where_condition`* are specified as described in Section 13.2.13, “SELECT Statement”.

You need the `UPDATE` privilege only for columns referenced in an `UPDATE` that are actually updated. You need only the `SELECT` privilege for any columns that are read but not modified.

The `UPDATE` statement supports the following modifiers:

- With the `LOW_PRIORITY` modifier, execution of the `UPDATE` is delayed until no other clients are reading from the table. This affects only storage engines that use only table-level locking (such as `MyISAM`, `MEMORY`, and `MERGE`).
- With the `IGNORE` modifier, the update statement does not abort even if errors occur during the update. Rows for which duplicate-key conflicts occur on a unique key value are not updated. Rows updated to values that would cause data conversion errors are updated to the closest valid values instead. For more information, see The Effect of IGNORE on Statement Execution.

`UPDATE IGNORE` statements, including those having an `ORDER BY` clause, are flagged as unsafe for statement-based replication. (This is because the order in which the rows are updated determines which rows are ignored.) Such statements produce a warning in the error log when using statement-based mode and are written to the binary log using the row-based format when using `MIXED` mode. (Bug #11758262, Bug #50439) See Section 17.2.1.3, “Determination of Safe and Unsafe Statements in Binary Logging”, for more information.

If you access a column from the table to be updated in an expression, `UPDATE` uses the current value of the column. For example, the following statement sets `col1` to one more than its current value:

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

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;
```

Single-table `UPDATE` assignments are generally evaluated from left to right. For multiple-table updates, there is no guarantee that assignments are carried out in any particular order.

If you set a column to the value it currently has, MySQL notices this and does not update it.

If you update a column that has been declared `NOT NULL` by setting to `NULL`, an error occurs if strict SQL mode is enabled; otherwise, the column is set to the implicit default value for the column data type and the warning count is incremented. The implicit default value is `0` for numeric types, the empty string (`''`) for string types, and the "zero" value for date and time types. See Section 11.6, "Data Type Default Values".

If a generated column is updated explicitly, the only permitted value is `DEFAULT`. For information about generated columns, see Section 13.1.20.8, "CREATE TABLE and Generated Columns".

`UPDATE` returns the number of rows that were actually changed. The `mysql_info()` C API function returns the number of rows that were matched and updated and the number of warnings that occurred during the `UPDATE`.

You can use `LIMIT` *`row_count`* to restrict the scope of the `UPDATE`. A `LIMIT` clause is a rows-matched restriction. The statement stops as soon as it has found *`row_count`* rows that satisfy the `WHERE` clause, whether or not they actually were changed.

If an `UPDATE` statement includes an `ORDER BY` clause, the rows are updated in the order specified by the clause. This can be useful in certain situations that might otherwise result in an error. Suppose that a table `t` contains a column `id` that has a unique index. The following statement could fail with a duplicate-key error, depending on the order in which rows are updated:

```
UPDATE t SET id = id + 1;
```

For example, if the table contains 1 and 2 in the `id` column and 1 is updated to 2 before 2 is updated to 3, an error occurs. To avoid this problem, add an `ORDER BY` clause to cause the rows with larger `id` values to be updated before those with smaller values:

```
UPDATE t SET id = id + 1 ORDER BY id DESC;
```

You can also perform `UPDATE` operations covering multiple tables. However, you cannot use `ORDER BY` or `LIMIT` with a multiple-table `UPDATE`. The *`table_references`* clause lists the tables involved in the join. Its syntax is described in Section 13.2.13.2, "JOIN Clause". Here is an example:

```
UPDATE items,month SET items.price=month.price
WHERE items.id=month.id;
```

The preceding example shows an inner join that uses the comma operator, but multiple-table `UPDATE` statements can use any type of join permitted in `SELECT` statements, such as `LEFT JOIN`.

If you use a multiple-table `UPDATE` statement involving `InnoDB` tables for which there are foreign key constraints, the MySQL optimizer might process tables in an order that differs from that of their parent/child relationship. In this case, the statement fails and rolls back. Instead, update a single table and rely on the `ON UPDATE` capabilities that `InnoDB` provides to cause the other tables to be modified accordingly. See Section 13.1.20.5, "FOREIGN KEY Constraints".

You cannot update a table and select directly from the same table in a subquery. You can work around this by using a multi-table update in which one of the tables is derived from the table that you actually wish to update, and referring to the derived table using an alias. Suppose you wish to update a table named `items` which is defined using the statement shown here:

```
CREATE TABLE items (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    wholesale DECIMAL(6,2) NOT NULL DEFAULT 0.00,
    retail DECIMAL(6,2) NOT NULL DEFAULT 0.00,
    quantity BIGINT NOT NULL DEFAULT 0
);
```

To reduce the retail price of any items for which the markup is 30% or greater and of which you have fewer than one hundred in stock, you might try to use an `UPDATE` statement such as the one following, which uses a subquery in the `WHERE` clause. As shown here, this statement does not work:

```
mysql> UPDATE items
     > SET retail = retail * 0.9
     > WHERE id IN
     >     (SELECT id FROM items
     >         WHERE retail / wholesale >= 1.3 AND quantity > 100);
ERROR 1093 (HY000): You can't specify target table 'items' for update in FROM clause
```

Instead, you can employ a multi-table update in which the subquery is moved into the list of tables to be updated, using an alias to reference it in the outermost `WHERE` clause, like this:

```
UPDATE items,
       (SELECT id FROM items
        WHERE id IN
            (SELECT id FROM items
             WHERE retail / wholesale >= 1.3 AND quantity < 100))
        AS discounted
SET items.retail = items.retail * 0.9
WHERE items.id = discounted.id;
```

Because the optimizer tries by default to merge the derived table `discounted` into the outermost query block, this works only if you force materialization of the derived table. You can do this by setting the `derived_merge` flag of the `optimizer_switch` system variable to `off` before running the update, or by using the `NO_MERGE` optimizer hint, as shown here:

```
UPDATE /*+ NO_MERGE(discounted) */ items,
       (SELECT id FROM items
        WHERE retail / wholesale >= 1.3 AND quantity < 100)
        AS discounted
    SET items.retail = items.retail * 0.9
    WHERE items.id = discounted.id;
```

The advantage of using the optimizer hint in such a case is that it applies only within the query block where it is used, so that it is not necessary to change the value of `optimizer_switch` again after executing the `UPDATE`.

Another possibility is to rewrite the subquery so that it does not use `IN` or `EXISTS`, like this:

```
UPDATE items,
       (SELECT id, retail / wholesale AS markup, quantity FROM items)
       AS discounted
    SET items.retail = items.retail * 0.9
    WHERE discounted.markup >= 1.3
    AND discounted.quantity < 100
    AND items.id = discounted.id;
```

In this case, the subquery is materialized by default rather than merged, so it is not necessary to disable merging of the derived table.

13.2.18 UNION Clause

```
query_expression_body UNION [ALL | DISTINCT] query_block
    [UNION [ALL | DISTINCT] query_expression_body]
    [...]

query_expression_body:
    See Section 13.2.14, “Set Operations with UNION, INTERSECT, and EXCEPT”
```

`UNION` combines the result from multiple query blocks into a single result set. This example uses `SELECT` statements:

```
mysql> SELECT 1, 2;
+---+---+
| 1 | 2 |
+---+---+
```

```
| 1 | 2 |
+---+---+
mysql> SELECT 'a', 'b';
+---+---+
| a | b |
+---+---+
| a | b |
+---+---+
mysql> SELECT 1, 2 UNION SELECT 'a', 'b';
+---+---+
| 1 | 2 |
+---+---+
| 1 | 2 |
| a | b |
+---+---+
```

UNION Handing in MySQL 8.0 Compared to MySQL 5.7

In MySQL 8.0, the parser rules for `SELECT` and `UNION` were refactored to be more consistent (the same `SELECT` syntax applies uniformly in each such context) and reduce duplication. Compared to MySQL 5.7, several user-visible effects resulted from this work, which may require rewriting of certain statements:

- `NATURAL JOIN` permits an optional `INNER` keyword (`NATURAL INNER JOIN`), in compliance with standard SQL.
- Right-deep joins without parentheses are permitted (for example, `... JOIN ... JOIN ... ON ... ON`), in compliance with standard SQL.
- `STRAIGHT_JOIN` now permits a `USING` clause, similar to other inner joins.
- The parser accepts parentheses around query expressions. For example, `(SELECT ... UNION SELECT ...)` is permitted. See also Section 13.2.11, “Parenthesized Query Expressions”.
- The parser better conforms to the documented permitted placement of the `SQL_CACHE` and `SQL_NO_CACHE` query modifiers.
- Left-hand nesting of unions, previously permitted only in subqueries, is now permitted in top-level statements. For example, this statement is now accepted as valid:

  ```
  (SELECT 1 UNION SELECT 1) UNION SELECT 1;
  ```

- Locking clauses (`FOR UPDATE`, `LOCK IN SHARE MODE`) are allowed only in non-`UNION` queries. This means that parentheses must be used for `SELECT` statements containing locking clauses. This statement is no longer accepted as valid:

  ```
  SELECT 1 FOR UPDATE UNION SELECT 1 FOR UPDATE;
  ```

  Instead, write the statement like this:

  ```
  (SELECT 1 FOR UPDATE) UNION (SELECT 1 FOR UPDATE);
  ```

13.2.19 VALUES Statement

`VALUES` is a DML statement introduced in MySQL 8.0.19 which returns a set of one or more rows as a table. In other words, it is a table value constructor which also functions as a standalone SQL statement.

```
VALUES row_constructor_list [ORDER BY column_designator] [LIMIT number]

row_constructor_list:
    ROW(value_list)[, ROW(value_list)][, ...]

value_list:
    value[, value][, ...]
```

```
column_designator:
    column_index
```

The VALUES statement consists of the VALUES keyword followed by a list of one or more row constructors, separated by commas. A row constructor consists of the ROW() row constructor clause with a value list of one or more scalar values enclosed in the parentheses. A value can be a literal of any MySQL data type or an expression that resolves to a scalar value.

ROW() cannot be empty (but each of the supplied scalar values can be NULL). Each ROW() in the same VALUES statement must have the same number of values in its value list.

The DEFAULT keyword is not supported by VALUES and causes a syntax error, except when it is used to supply values in an INSERT statement.

The output of VALUES is a table:

```
mysql> VALUES ROW(1,-2,3), ROW(5,7,9), ROW(4,6,8);
+----------+----------+----------+
| column_0 | column_1 | column_2 |
+----------+----------+----------+
|        1 |       -2 |        3 |
|        5 |        7 |        9 |
|        4 |        6 |        8 |
+----------+----------+----------+
3 rows in set (0.00 sec)
```

The columns of the table output from VALUES have the implicitly named columns column_0, column_1, column_2, and so on, always beginning with 0. This fact can be used to order the rows by column using an optional ORDER BY clause in the same way that this clause works with a SELECT statement, as shown here:

```
mysql> VALUES ROW(1,-2,3), ROW(5,7,9), ROW(4,6,8) ORDER BY column_1;
+----------+----------+----------+
| column_0 | column_1 | column_2 |
+----------+----------+----------+
|        1 |       -2 |        3 |
|        4 |        6 |        8 |
|        5 |        7 |        9 |
+----------+----------+----------+
3 rows in set (0.00 sec)
```

In MySQL 8.0.21 and later, the VALUES statement also supports a LIMIT clause for limiting the number of rows in the output. (Previously, LIMIT was allowed but did nothing.)

The VALUES statement is permissive regarding data types of column values; you can mix types within the same column, as shown here:

```
mysql> VALUES ROW("q", 42, '2019-12-18'),
    ->     ROW(23, "abc", 98.6),
    ->     ROW(27.0002, "Mary Smith", '{"a": 10, "b": 25}');
+----------+------------+--------------------+
| column_0 | column_1   | column_2           |
+----------+------------+--------------------+
| q        | 42         | 2019-12-18         |
| 23       | abc        | 98.6               |
| 27.0002  | Mary Smith | {"a": 10, "b": 25} |
+----------+------------+--------------------+
3 rows in set (0.00 sec)
```

Important

VALUES with one or more instances of ROW() acts as a table value constructor; although it can be used to supply values in an INSERT or REPLACE statement, do not confuse it with the VALUES keyword that is also used for this purpose. You should also not confuse it with the VALUES() function that refers to column values in INSERT ... ON DUPLICATE KEY UPDATE.

> You should also bear in mind that `ROW()` is a row value constructor (see Section 13.2.15.5, “Row Subqueries”), whereas `VALUES ROW()` is a table value constructor; the two cannot be used interchangeably.

`VALUES` can be used in many cases where you could employ `SELECT`, including those listed here:

- With `UNION`, as shown here:

```
mysql> SELECT 1,2 UNION SELECT 10,15;
+----+----+
| 1  | 2  |
+----+----+
|  1 |  2 |
| 10 | 15 |
+----+----+
2 rows in set (0.00 sec)

mysql> VALUES ROW(1,2) UNION VALUES ROW(10,15);
+----------+----------+
| column_0 | column_1 |
+----------+----------+
|        1 |        2 |
|       10 |       15 |
+----------+----------+
2 rows in set (0.00 sec)
```

  You can union together constructed tables having more than one row, like this:

```
mysql> VALUES ROW(1,2), ROW(3,4), ROW(5,6)
    >     UNION VALUES ROW(10,15),ROW(20,25);
+----------+----------+
| column_0 | column_1 |
+----------+----------+
|        1 |        2 |
|        3 |        4 |
|        5 |        6 |
|       10 |       15 |
|       20 |       25 |
+----------+----------+
5 rows in set (0.00 sec)
```

  You can also (and it is usually preferable to) omit `UNION` altogether in such cases and use a single **`VALUES`** statement, like this:

```
mysql> VALUES ROW(1,2), ROW(3,4), ROW(5,6), ROW(10,15), ROW(20,25);
+----------+----------+
| column_0 | column_1 |
+----------+----------+
|        1 |        2 |
|        3 |        4 |
|        5 |        6 |
|       10 |       15 |
|       20 |       25 |
+----------+----------+
```

  `VALUES` can also be used in unions with `SELECT` statements, `TABLE` statements, or both.

  The constructed tables in the `UNION` must contain the same number of columns, just as if you were using `SELECT`. See Section 13.2.18, “UNION Clause”, for further examples.

  In MySQL 8.0.31 and later, you can use `EXCEPT` and `INTERSECT` with `VALUES` in much the same way as `UNION`, as shown here:

```
mysql> VALUES ROW(1,2), ROW(3,4), ROW(5,6)
    ->   INTERSECT
    -> VALUES ROW(10,15), ROW(20,25), ROW(3,4);
+----------+----------+
| column_0 | column_1 |
```

```
+----------+----------+
|        3 |        4 |
+----------+----------+
1 row in set (0.00 sec)

mysql> VALUES ROW(1,2), ROW(3,4), ROW(5,6)
    ->   EXCEPT
    -> VALUES ROW(10,15), ROW(20,25), ROW(3,4);
+----------+----------+
| column_0 | column_1 |
+----------+----------+
|        1 |        2 |
|        5 |        6 |
+----------+----------+
2 rows in set (0.00 sec)
```

See Section 13.2.4, “EXCEPT Clause”, and Section 13.2.8, “INTERSECT Clause”, for more information.

- In joins. See Section 13.2.13.2, “JOIN Clause”, for more information and examples.
- In place of `VALUES()` in an `INSERT` or `REPLACE` statement, in which case its semantics differ slightly from what is described here. See Section 13.2.7, “INSERT Statement”, for details.
- In place of the source table in `CREATE TABLE ... SELECT` and `CREATE VIEW ... SELECT`. See the descriptions of these statements for more information and examples.

13.2.20 WITH (Common Table Expressions)

A common table expression (CTE) is a named temporary result set that exists within the scope of a single statement and that can be referred to later within that statement, possibly multiple times. The following discussion describes how to write statements that use CTEs.

- Common Table Expressions
- Recursive Common Table Expressions
- Limiting Common Table Expression Recursion
- Recursive Common Table Expression Examples
- Common Table Expressions Compared to Similar Constructs

For information about CTE optimization, see Section 8.2.2.4, “Optimizing Derived Tables, View References, and Common Table Expressions with Merging or Materialization”.

Additional Resources

These articles contain additional information about using CTEs in MySQL, including many examples:

- MySQL 8.0 Labs: [Recursive] Common Table Expressions in MySQL (CTEs)
- MySQL 8.0 Labs: [Recursive] Common Table Expressions in MySQL (CTEs), Part Two – how to generate series
- MySQL 8.0 Labs: [Recursive] Common Table Expressions in MySQL (CTEs), Part Three – hierarchies
- MySQL 8.0.1: [Recursive] Common Table Expressions in MySQL (CTEs), Part Four – depth-first or breadth-first traversal, transitive closure, cycle avoidance

Common Table Expressions

To specify common table expressions, use a `WITH` clause that has one or more comma-separated subclauses. Each subclause provides a subquery that produces a result set, and associates a name

with the subquery. The following example defines CTEs named `cte1` and `cte2` in the `WITH` clause, and refers to them in the top-level `SELECT` that follows the `WITH` clause:

```
WITH
  cte1 AS (SELECT a, b FROM table1),
  cte2 AS (SELECT c, d FROM table2)
SELECT b, d FROM cte1 JOIN cte2
WHERE cte1.a = cte2.c;
```

In the statement containing the `WITH` clause, each CTE name can be referenced to access the corresponding CTE result set.

A CTE name can be referenced in other CTEs, enabling CTEs to be defined based on other CTEs.

A CTE can refer to itself to define a recursive CTE. Common applications of recursive CTEs include series generation and traversal of hierarchical or tree-structured data.

Common table expressions are an optional part of the syntax for DML statements. They are defined using a `WITH` clause:

```
with_clause:
    WITH [RECURSIVE]
        cte_name [(col_name [, col_name] ...)] AS (subquery)
        [, cte_name [(col_name [, col_name] ...)] AS (subquery)] ...
```

`cte_name` names a single common table expression and can be used as a table reference in the statement containing the `WITH` clause.

The *`subquery`* part of `AS (subquery)` is called the “subquery of the CTE” and is what produces the CTE result set. The parentheses following `AS` are required.

A common table expression is recursive if its subquery refers to its own name. The `RECURSIVE` keyword must be included if any CTE in the `WITH` clause is recursive. For more information, see Recursive Common Table Expressions.

Determination of column names for a given CTE occurs as follows:

- If a parenthesized list of names follows the CTE name, those names are the column names:

  ```
  WITH cte (col1, col2) AS
  (
    SELECT 1, 2
    UNION ALL
    SELECT 3, 4
  )
  SELECT col1, col2 FROM cte;
  ```

  The number of names in the list must be the same as the number of columns in the result set.

- Otherwise, the column names come from the select list of the first `SELECT` within the `AS (subquery)` part:

  ```
  WITH cte AS
  (
    SELECT 1 AS col1, 2 AS col2
    UNION ALL
    SELECT 3, 4
  )
  SELECT col1, col2 FROM cte;
  ```

A `WITH` clause is permitted in these contexts:

- At the beginning of `SELECT`, `UPDATE`, and `DELETE` statements.

  ```
  WITH ... SELECT ...
  WITH ... UPDATE ...
  WITH ... DELETE ...
  ```

- At the beginning of subqueries (including derived table subqueries):

```
SELECT ... WHERE id IN (WITH ... SELECT ...) ...
SELECT * FROM (WITH ... SELECT ...) AS dt ...
```

- Immediately preceding `SELECT` for statements that include a `SELECT` statement:

```
INSERT ... WITH ... SELECT ...
REPLACE ... WITH ... SELECT ...
CREATE TABLE ... WITH ... SELECT ...
CREATE VIEW ... WITH ... SELECT ...
DECLARE CURSOR ... WITH ... SELECT ...
EXPLAIN ... WITH ... SELECT ...
```

Only one `WITH` clause is permitted at the same level. `WITH` followed by `WITH` at the same level is not permitted, so this is illegal:

```
WITH cte1 AS (...) WITH cte2 AS (...) SELECT ...
```

To make the statement legal, use a single `WITH` clause that separates the subclauses by a comma:

```
WITH cte1 AS (...), cte2 AS (...) SELECT ...
```

However, a statement can contain multiple `WITH` clauses if they occur at different levels:

```
WITH cte1 AS (SELECT 1)
SELECT * FROM (WITH cte2 AS (SELECT 2) SELECT * FROM cte2 JOIN cte1) AS dt;
```

A `WITH` clause can define one or more common table expressions, but each CTE name must be unique to the clause. This is illegal:

```
WITH cte1 AS (...), cte1 AS (...) SELECT ...
```

To make the statement legal, define the CTEs with unique names:

```
WITH cte1 AS (...), cte2 AS (...) SELECT ...
```

A CTE can refer to itself or to other CTEs:

- A self-referencing CTE is recursive.
- A CTE can refer to CTEs defined earlier in the same `WITH` clause, but not those defined later.

  This constraint rules out mutually-recursive CTEs, where `cte1` references `cte2` and `cte2` references `cte1`. One of those references must be to a CTE defined later, which is not permitted.
- A CTE in a given query block can refer to CTEs defined in query blocks at a more outer level, but not CTEs defined in query blocks at a more inner level.

For resolving references to objects with the same names, derived tables hide CTEs; and CTEs hide base tables, `TEMPORARY` tables, and views. Name resolution occurs by searching for objects in the same query block, then proceeding to outer blocks in turn while no object with the name is found.

Like derived tables, a CTE cannot contain outer references prior to MySQL 8.0.14. This is a MySQL restriction that is lifted in MySQL 8.0.14, not a restriction of the SQL standard. For additional syntax considerations specific to recursive CTEs, see Recursive Common Table Expressions.

Recursive Common Table Expressions

A recursive common table expression is one having a subquery that refers to its own name. For example:

```
WITH RECURSIVE cte (n) AS
(
  SELECT 1
```

```
  UNION ALL
  SELECT n + 1 FROM cte WHERE n < 5
)
SELECT * FROM cte;
```

When executed, the statement produces this result, a single column containing a simple linear sequence:

```
+------+
| n    |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
|    5 |
+------+
```

A recursive CTE has this structure:

- The `WITH` clause must begin with `WITH RECURSIVE` if any CTE in the `WITH` clause refers to itself. (If no CTE refers to itself, `RECURSIVE` is permitted but not required.)

  If you forget `RECURSIVE` for a recursive CTE, this error is a likely result:

  ```
  ERROR 1146 (42S02): Table 'cte_name' doesn't exist
  ```

- The recursive CTE subquery has two parts, separated by `UNION ALL` or `UNION [DISTINCT]`:

  ```
  SELECT ...      -- return initial row set
  UNION ALL
  SELECT ...      -- return additional row sets
  ```

  The first `SELECT` produces the initial row or rows for the CTE and does not refer to the CTE name. The second `SELECT` produces additional rows and recurses by referring to the CTE name in its `FROM` clause. Recursion ends when this part produces no new rows. Thus, a recursive CTE consists of a nonrecursive `SELECT` part followed by a recursive `SELECT` part.

  Each `SELECT` part can itself be a union of multiple `SELECT` statements.

- The types of the CTE result columns are inferred from the column types of the nonrecursive `SELECT` part only, and the columns are all nullable. For type determination, the recursive `SELECT` part is ignored.

- If the nonrecursive and recursive parts are separated by `UNION DISTINCT`, duplicate rows are eliminated. This is useful for queries that perform transitive closures, to avoid infinite loops.

- Each iteration of the recursive part operates only on the rows produced by the previous iteration. If the recursive part has multiple query blocks, iterations of each query block are scheduled in unspecified order, and each query block operates on rows that have been produced either by its previous iteration or by other query blocks since that previous iteration's end.

The recursive CTE subquery shown earlier has this nonrecursive part that retrieves a single row to produce the initial row set:

```
SELECT 1
```

The CTE subquery also has this recursive part:

```
SELECT n + 1 FROM cte WHERE n < 5
```

At each iteration, that `SELECT` produces a row with a new value one greater than the value of `n` from the previous row set. The first iteration operates on the initial row set (`1`) and produces `1+1=2`; the second iteration operates on the first iteration's row set (`2`) and produces `2+1=3`; and so forth. This continues until recursion ends, which occurs when `n` is no longer less than 5.

If the recursive part of a CTE produces wider values for a column than the nonrecursive part, it may be necessary to widen the column in the nonrecursive part to avoid data truncation. Consider this statement:

```
WITH RECURSIVE cte AS
(
  SELECT 1 AS n, 'abc' AS str
  UNION ALL
  SELECT n + 1, CONCAT(str, str) FROM cte WHERE n < 3
)
SELECT * FROM cte;
```

In nonstrict SQL mode, the statement produces this output:

```
+------+------+
| n    | str  |
+------+------+
|    1 | abc  |
|    2 | abc  |
|    3 | abc  |
+------+------+
```

The `str` column values are all `'abc'` because the nonrecursive `SELECT` determines the column widths. Consequently, the wider `str` values produced by the recursive `SELECT` are truncated.

In strict SQL mode, the statement produces an error:

```
ERROR 1406 (22001): Data too long for column 'str' at row 1
```

To address this issue, so that the statement does not produce truncation or errors, use `CAST()` in the nonrecursive `SELECT` to make the `str` column wider:

```
WITH RECURSIVE cte AS
(
  SELECT 1 AS n, CAST('abc' AS CHAR(20)) AS str
  UNION ALL
  SELECT n + 1, CONCAT(str, str) FROM cte WHERE n < 3
)
SELECT * FROM cte;
```

Now the statement produces this result, without truncation:

```
+------+--------------+
| n    | str          |
+------+--------------+
|    1 | abc          |
|    2 | abcabc       |
|    3 | abcabcabcabc |
+------+--------------+
```

Columns are accessed by name, not position, which means that columns in the recursive part can access columns in the nonrecursive part that have a different position, as this CTE illustrates:

```
WITH RECURSIVE cte AS
(
  SELECT 1 AS n, 1 AS p, -1 AS q
  UNION ALL
  SELECT n + 1, q * 2, p * 2 FROM cte WHERE n < 5
)
SELECT * FROM cte;
```

Because `p` in one row is derived from `q` in the previous row, and vice versa, the positive and negative values swap positions in each successive row of the output:

```
+------+------+------+
| n    | p    | q    |
+------+------+------+
|    1 |    1 |   -1 |
|    2 |   -2 |    2 |
```

```
|    3 |    4 |   -4 |
|    4 |   -8 |    8 |
|    5 |   16 |  -16 |
+------+------+------+
```

Some syntax constraints apply within recursive CTE subqueries:

- The recursive `SELECT` part must not contain these constructs:
  - Aggregate functions such as `SUM()`
  - Window functions
  - `GROUP BY`
  - `ORDER BY`
  - `DISTINCT`

  Prior to MySQL 8.0.19, the recursive `SELECT` part of a recursive CTE also could not use a `LIMIT` clause. This restriction is lifted in MySQL 8.0.19, and `LIMIT` is now supported in such cases, along with an optional `OFFSET` clause. The effect on the result set is the same as when using `LIMIT` in the outermost `SELECT`, but is also more efficient, since using it with the recursive `SELECT` stops the generation of rows as soon as the requested number of them has been produced.

  These constraints do not apply to the nonrecursive `SELECT` part of a recursive CTE. The prohibition on `DISTINCT` applies only to `UNION` members; `UNION DISTINCT` is permitted.

- The recursive `SELECT` part must reference the CTE only once and only in its `FROM` clause, not in any subquery. It can reference tables other than the CTE and join them with the CTE. If used in a join like this, the CTE must not be on the right side of a `LEFT JOIN`.

These constraints come from the SQL standard, other than the MySQL-specific exclusions of `ORDER BY`, `LIMIT` (MySQL 8.0.18 and earlier), and `DISTINCT`.

For recursive CTEs, `EXPLAIN` output rows for recursive `SELECT` parts display `Recursive` in the `Extra` column.

Cost estimates displayed by `EXPLAIN` represent cost per iteration, which might differ considerably from total cost. The optimizer cannot predict the number of iterations because it cannot predict at what point the `WHERE` clause becomes false.

CTE actual cost may also be affected by result set size. A CTE that produces many rows may require an internal temporary table large enough to be converted from in-memory to on-disk format and may suffer a performance penalty. If so, increasing the permitted in-memory temporary table size may improve performance; see Section 8.4.4, “Internal Temporary Table Use in MySQL”.

Limiting Common Table Expression Recursion

It is important for recursive CTEs that the recursive `SELECT` part include a condition to terminate recursion. As a development technique to guard against a runaway recursive CTE, you can force termination by placing a limit on execution time:

- The `cte_max_recursion_depth` system variable enforces a limit on the number of recursion levels for CTEs. The server terminates execution of any CTE that recurses more levels than the value of this variable.
- The `max_execution_time` system variable enforces an execution timeout for `SELECT` statements executed within the current session.
- The `MAX_EXECUTION_TIME` optimizer hint enforces a per-query execution timeout for the `SELECT` statement in which it appears.

Suppose that a recursive CTE is mistakenly written with no recursion execution termination condition:

```
WITH RECURSIVE cte (n) AS
(
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM cte
)
SELECT * FROM cte;
```

By default, `cte_max_recursion_depth` has a value of 1000, causing the CTE to terminate when it recurses past 1000 levels. Applications can change the session value to adjust for their requirements:

```
SET SESSION cte_max_recursion_depth = 10;      -- permit only shallow recursion
SET SESSION cte_max_recursion_depth = 1000000; -- permit deeper recursion
```

You can also set the global `cte_max_recursion_depth` value to affect all sessions that begin subsequently.

For queries that execute and thus recurse slowly or in contexts for which there is reason to set the `cte_max_recursion_depth` value very high, another way to guard against deep recursion is to set a per-session timeout. To do so, execute a statement like this prior to executing the CTE statement:

```
SET max_execution_time = 1000; -- impose one second timeout
```

Alternatively, include an optimizer hint within the CTE statement itself:

```
WITH RECURSIVE cte (n) AS
(
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM cte
)
SELECT /*+ SET_VAR(cte_max_recursion_depth = 1M) */ * FROM cte;

WITH RECURSIVE cte (n) AS
(
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM cte
)
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM cte;
```

Beginning with MySQL 8.0.19, you can also use `LIMIT` within the recursive query to impose a maximum number of rows to be returned to the outermost `SELECT`, for example:

```
WITH RECURSIVE cte (n) AS
(
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM cte LIMIT 10000
)
SELECT * FROM cte;
```

You can do this in addition to or instead of setting a time limit. Thus, the following CTE terminates after returning ten thousand rows or running for one second (1000 milliseconds), whichever occurs first:

```
WITH RECURSIVE cte (n) AS
(
  SELECT 1
  UNION ALL
  SELECT n + 1 FROM cte LIMIT 10000
)
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM cte;
```

If a recursive query without an execution time limit enters an infinite loop, you can terminate it from another session using `KILL QUERY`. Within the session itself, the client program used to run the query

might provide a way to kill the query. For example, in `mysql`, typing **Control+C** interrupts the current statement.

Recursive Common Table Expression Examples

As mentioned previously, recursive common table expressions (CTEs) are frequently used for series generation and traversing hierarchical or tree-structured data. This section shows some simple examples of these techniques.

- Fibonacci Series Generation
- Date Series Generation
- Hierarchical Data Traversal

Fibonacci Series Generation

A Fibonacci series begins with the two numbers 0 and 1 (or 1 and 1) and each number after that is the sum of the previous two numbers. A recursive common table expression can generate a Fibonacci series if each row produced by the recursive `SELECT` has access to the two previous numbers from the series. The following CTE generates a 10-number series using 0 and 1 as the first two numbers:

```
WITH RECURSIVE fibonacci (n, fib_n, next_fib_n) AS
(
  SELECT 1, 0, 1
  UNION ALL
  SELECT n + 1, next_fib_n, fib_n + next_fib_n
    FROM fibonacci WHERE n < 10
)
SELECT * FROM fibonacci;
```

The CTE produces this result:

```
+------+-------+------------+
| n    | fib_n | next_fib_n |
+------+-------+------------+
|    1 |     0 |          1 |
|    2 |     1 |          1 |
|    3 |     1 |          2 |
|    4 |     2 |          3 |
|    5 |     3 |          5 |
|    6 |     5 |          8 |
|    7 |     8 |         13 |
|    8 |    13 |         21 |
|    9 |    21 |         34 |
|   10 |    34 |         55 |
+------+-------+------------+
```

How the CTE works:

- `n` is a display column to indicate that the row contains the `n`-th Fibonacci number. For example, the 8th Fibonacci number is 13.
- The `fib_n` column displays Fibonacci number `n`.
- The `next_fib_n` column displays the next Fibonacci number after number `n`. This column provides the next series value to the next row, so that row can produce the sum of the two previous series values in its `fib_n` column.
- Recursion ends when `n` reaches 10. This is an arbitrary choice, to limit the output to a small set of rows.

The preceding output shows the entire CTE result. To select just part of it, add an appropriate `WHERE` clause to the top-level `SELECT`. For example, to select the 8th Fibonacci number, do this:

```
mysql> WITH RECURSIVE fibonacci ...
       ...
```

```
       SELECT fib_n FROM fibonacci WHERE n = 8;
+-------+
| fib_n |
+-------+
|    13 |
+-------+
```

Date Series Generation

A common table expression can generate a series of successive dates, which is useful for generating summaries that include a row for all dates in the series, including dates not represented in the summarized data.

Suppose that a table of sales numbers contains these rows:

```
mysql> SELECT * FROM sales ORDER BY date, price;
+------------+--------+
| date       | price  |
+------------+--------+
| 2017-01-03 | 100.00 |
| 2017-01-03 | 200.00 |
| 2017-01-06 |  50.00 |
| 2017-01-08 |  10.00 |
| 2017-01-08 |  20.00 |
| 2017-01-08 | 150.00 |
| 2017-01-10 |   5.00 |
+------------+--------+
```

This query summarizes the sales per day:

```
mysql> SELECT date, SUM(price) AS sum_price
       FROM sales
       GROUP BY date
       ORDER BY date;
+------------+-----------+
| date       | sum_price |
+------------+-----------+
| 2017-01-03 |    300.00 |
| 2017-01-06 |     50.00 |
| 2017-01-08 |    180.00 |
| 2017-01-10 |      5.00 |
+------------+-----------+
```

However, that result contains “holes” for dates not represented in the range of dates spanned by the table. A result that represents all dates in the range can be produced using a recursive CTE to generate that set of dates, joined with a `LEFT JOIN` to the sales data.

Here is the CTE to generate the date range series:

```
WITH RECURSIVE dates (date) AS
(
  SELECT MIN(date) FROM sales
  UNION ALL
  SELECT date + INTERVAL 1 DAY FROM dates
  WHERE date + INTERVAL 1 DAY <= (SELECT MAX(date) FROM sales)
)
SELECT * FROM dates;
```

The CTE produces this result:

```
+------------+
| date       |
+------------+
| 2017-01-03 |
| 2017-01-04 |
| 2017-01-05 |
| 2017-01-06 |
| 2017-01-07 |
| 2017-01-08 |
| 2017-01-09 |
```

```
| 2017-01-10 |
+------------+
```

How the CTE works:

- The nonrecursive `SELECT` produces the lowest date in the date range spanned by the `sales` table.
- Each row produced by the recursive `SELECT` adds one day to the date produced by the previous row.
- Recursion ends after the dates reach the highest date in the date range spanned by the `sales` table.

Joining the CTE with a `LEFT JOIN` against the `sales` table produces the sales summary with a row for each date in the range:

```
WITH RECURSIVE dates (date) AS
(
  SELECT MIN(date) FROM sales
  UNION ALL
  SELECT date + INTERVAL 1 DAY FROM dates
  WHERE date + INTERVAL 1 DAY <= (SELECT MAX(date) FROM sales)
)
SELECT dates.date, COALESCE(SUM(price), 0) AS sum_price
FROM dates LEFT JOIN sales ON dates.date = sales.date
GROUP BY dates.date
ORDER BY dates.date;
```

The output looks like this:

```
+------------+-----------+
| date       | sum_price |
+------------+-----------+
| 2017-01-03 |    300.00 |
| 2017-01-04 |      0.00 |
| 2017-01-05 |      0.00 |
| 2017-01-06 |     50.00 |
| 2017-01-07 |      0.00 |
| 2017-01-08 |    180.00 |
| 2017-01-09 |      0.00 |
| 2017-01-10 |      5.00 |
+------------+-----------+
```

Some points to note:

- Are the queries inefficient, particularly the one with the `MAX()` subquery executed for each row in the recursive `SELECT`? `EXPLAIN` shows that the subquery containing `MAX()` is evaluated only once and the result is cached.
- The use of `COALESCE()` avoids displaying `NULL` in the `sum_price` column on days for which no sales data occur in the `sales` table.

Hierarchical Data Traversal

Recursive common table expressions are useful for traversing data that forms a hierarchy. Consider these statements that create a small data set that shows, for each employee in a company, the employee name and ID number, and the ID of the employee's manager. The top-level employee (the CEO), has a manager ID of `NULL` (no manager).

```
CREATE TABLE employees (
  id         INT PRIMARY KEY NOT NULL,
  name       VARCHAR(100) NOT NULL,
  manager_id INT NULL,
  INDEX (manager_id),
FOREIGN KEY (manager_id) REFERENCES employees (id)
);
INSERT INTO employees VALUES
(333, "Yasmina", NULL),  # Yasmina is the CEO (manager_id is NULL)
(198, "John", 333),      # John has ID 198 and reports to 333 (Yasmina)
```

```
(692, "Tarek", 333),
(29, "Pedro", 198),
(4610, "Sarah", 29),
(72, "Pierre", 29),
(123, "Adil", 692);
```

The resulting data set looks like this:

```
mysql> SELECT * FROM employees ORDER BY id;
+------+---------+------------+
| id   | name    | manager_id |
+------+---------+------------+
|   29 | Pedro   |        198 |
|   72 | Pierre  |         29 |
|  123 | Adil    |        692 |
|  198 | John    |        333 |
|  333 | Yasmina |       NULL |
|  692 | Tarek   |        333 |
| 4610 | Sarah   |         29 |
+------+---------+------------+
```

To produce the organizational chart with the management chain for each employee (that is, the path from CEO to employee), use a recursive CTE:

```
WITH RECURSIVE employee_paths (id, name, path) AS
(
  SELECT id, name, CAST(id AS CHAR(200))
    FROM employees
    WHERE manager_id IS NULL
  UNION ALL
  SELECT e.id, e.name, CONCAT(ep.path, ',', e.id)
    FROM employee_paths AS ep JOIN employees AS e
      ON ep.id = e.manager_id
)
SELECT * FROM employee_paths ORDER BY path;
```

The CTE produces this output:

```
+------+---------+-----------------+
| id   | name    | path            |
+------+---------+-----------------+
|  333 | Yasmina | 333             |
|  198 | John    | 333,198         |
|   29 | Pedro   | 333,198,29      |
| 4610 | Sarah   | 333,198,29,4610 |
|   72 | Pierre  | 333,198,29,72   |
|  692 | Tarek   | 333,692         |
|  123 | Adil    | 333,692,123     |
+------+---------+-----------------+
```

How the CTE works:

- The nonrecursive `SELECT` produces the row for the CEO (the row with a `NULL` manager ID).

  The `path` column is widened to `CHAR(200)` to ensure that there is room for the longer `path` values produced by the recursive `SELECT`.

- Each row produced by the recursive `SELECT` finds all employees who report directly to an employee produced by a previous row. For each such employee, the row includes the employee ID and name, and the employee management chain. The chain is the manager's chain, with the employee ID added to the end.

- Recursion ends when employees have no others who report to them.

To find the path for a specific employee or employees, add a `WHERE` clause to the top-level `SELECT`. For example, to display the results for Tarek and Sarah, modify that `SELECT` like this:

```
mysql> WITH RECURSIVE ...
       ...
```

```
      SELECT * FROM employees_extended
      WHERE id IN (692, 4610)
      ORDER BY path;
+------+-------+-----------------+
| id   | name  | path            |
+------+-------+-----------------+
| 4610 | Sarah | 333,198,29,4610 |
|  692 | Tarek | 333,692         |
+------+-------+-----------------+
```

Common Table Expressions Compared to Similar Constructs

Common table expressions (CTEs) are similar to derived tables in some ways:

- Both constructs are named.
- Both constructs exist for the scope of a single statement.

Because of these similarities, CTEs and derived tables often can be used interchangeably. As a trivial example, these statements are equivalent:

```
WITH cte AS (SELECT 1) SELECT * FROM cte;
SELECT * FROM (SELECT 1) AS dt;
```

However, CTEs have some advantages over derived tables:

- A derived table can be referenced only a single time within a query. A CTE can be referenced multiple times. To use multiple instances of a derived table result, you must derive the result multiple times.
- A CTE can be self-referencing (recursive).
- One CTE can refer to another.
- A CTE may be easier to read when its definition appears at the beginning of the statement rather than embedded within it.

CTEs are similar to tables created with `CREATE [TEMPORARY] TABLE` but need not be defined or dropped explicitly. For a CTE, you need no privileges to create tables.

13.3 Transactional and Locking Statements

MySQL supports local transactions (within a given client session) through statements such as `SET autocommit`, `START TRANSACTION`, `COMMIT`, and `ROLLBACK`. See Section 13.3.1, “START TRANSACTION, COMMIT, and ROLLBACK Statements”. XA transaction support enables MySQL to participate in distributed transactions as well. See Section 13.3.8, “XA Transactions”.

13.3.1 START TRANSACTION, COMMIT, and ROLLBACK Statements

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

These statements provide control over use of transactions:

- `START TRANSACTION` or `BEGIN` start a new transaction.

- `COMMIT` commits the current transaction, making its changes permanent.
- `ROLLBACK` rolls back the current transaction, canceling its changes.
- `SET autocommit` disables or enables the default autocommit mode for the current session.

By default, MySQL runs with autocommit mode enabled. This means that, when not otherwise inside a transaction, each statement is atomic, as if it were surrounded by `START TRANSACTION` and `COMMIT`. You cannot use `ROLLBACK` to undo the effect; however, if an error occurs during statement execution, the statement is rolled back.

To disable autocommit mode implicitly for a single series of statements, use the `START TRANSACTION` statement:

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

With `START TRANSACTION`, autocommit remains disabled until you end the transaction with `COMMIT` or `ROLLBACK`. The autocommit mode then reverts to its previous state.

`START TRANSACTION` permits several modifiers that control transaction characteristics. To specify multiple modifiers, separate them by commas.

- The `WITH CONSISTENT SNAPSHOT` modifier starts a consistent read for storage engines that are capable of it. This applies only to `InnoDB`. The effect is the same as issuing a `START TRANSACTION` followed by a `SELECT` from any `InnoDB` table. See Section 15.7.2.3, “Consistent Nonlocking Reads”. The `WITH CONSISTENT SNAPSHOT` modifier does not change the current transaction isolation level, so it provides a consistent snapshot only if the current isolation level is one that permits a consistent read. The only isolation level that permits a consistent read is `REPEATABLE READ`. For all other isolation levels, the `WITH CONSISTENT SNAPSHOT` clause is ignored. A warning is generated when the `WITH CONSISTENT SNAPSHOT` clause is ignored.
- The `READ WRITE` and `READ ONLY` modifiers set the transaction access mode. They permit or prohibit changes to tables used in the transaction. The `READ ONLY` restriction prevents the transaction from modifying or locking both transactional and nontransactional tables that are visible to other transactions; the transaction can still modify or lock temporary tables.

  MySQL enables extra optimizations for queries on `InnoDB` tables when the transaction is known to be read-only. Specifying `READ ONLY` ensures these optimizations are applied in cases where the read-only status cannot be determined automatically. See Section 8.5.3, “Optimizing InnoDB Read-Only Transactions” for more information.

  If no access mode is specified, the default mode applies. Unless the default has been changed, it is read/write. It is not permitted to specify both `READ WRITE` and `READ ONLY` in the same statement.

  In read-only mode, it remains possible to change tables created with the `TEMPORARY` keyword using DML statements. Changes made with DDL statements are not permitted, just as with permanent tables.

  For additional information about transaction access mode, including ways to change the default mode, see Section 13.3.7, “SET TRANSACTION Statement”.

  If the `read_only` system variable is enabled, explicitly starting a transaction with `START TRANSACTION READ WRITE` requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege).

Important

Many APIs used for writing MySQL client applications (such as JDBC) provide their own methods for starting transactions that can (and sometimes should) be

used instead of sending a `START TRANSACTION` statement from the client. See Chapter 29, *Connectors and APIs*, or the documentation for your API, for more information.

To disable autocommit mode explicitly, use the following statement:

```
SET autocommit=0;
```

After disabling autocommit mode by setting the `autocommit` variable to zero, changes to transaction-safe tables (such as those for `InnoDB` or `NDB`) are not made permanent immediately. You must use `COMMIT` to store your changes to disk or `ROLLBACK` to ignore the changes.

`autocommit` is a session variable and must be set for each session. To disable autocommit mode for each new connection, see the description of the `autocommit` system variable at Section 5.1.8, “Server System Variables”.

`BEGIN` and `BEGIN WORK` are supported as aliases of `START TRANSACTION` for initiating a transaction. `START TRANSACTION` is standard SQL syntax, is the recommended way to start an ad-hoc transaction, and permits modifiers that `BEGIN` does not.

The `BEGIN` statement differs from the use of the `BEGIN` keyword that starts a `BEGIN ... END` compound statement. The latter does not begin a transaction. See Section 13.6.1, “BEGIN ... END Compound Statement”.

Note

Within all stored programs (stored procedures and functions, triggers, and events), the parser treats `BEGIN [WORK]` as the beginning of a `BEGIN ... END` block. Begin a transaction in this context with `START TRANSACTION` instead.

The optional `WORK` keyword is supported for `COMMIT` and `ROLLBACK`, as are the `CHAIN` and `RELEASE` clauses. `CHAIN` and `RELEASE` can be used for additional control over transaction completion. The value of the `completion_type` system variable determines the default completion behavior. See Section 5.1.8, “Server System Variables”.

The `AND CHAIN` clause causes a new transaction to begin as soon as the current one ends, and the new transaction has the same isolation level as the just-terminated transaction. The new transaction also uses the same access mode (`READ WRITE` or `READ ONLY`) as the just-terminated transaction. The `RELEASE` clause causes the server to disconnect the current client session after terminating the current transaction. Including the `NO` keyword suppresses `CHAIN` or `RELEASE` completion, which can be useful if the `completion_type` system variable is set to cause chaining or release completion by default.

Beginning a transaction causes any pending transaction to be committed. See Section 13.3.3, “Statements That Cause an Implicit Commit”, for more information.

Beginning a transaction also causes table locks acquired with `LOCK TABLES` to be released, as though you had executed `UNLOCK TABLES`. Beginning a transaction does not release a global read lock acquired with `FLUSH TABLES WITH READ LOCK`.

For best results, transactions should be performed using only tables managed by a single transaction-safe storage engine. Otherwise, the following problems can occur:

- If you use tables from more than one transaction-safe storage engine (such as `InnoDB`), and the transaction isolation level is not `SERIALIZABLE`, it is possible that when one transaction commits, another ongoing transaction that uses the same tables sees only some of the changes made by the first transaction. That is, the atomicity of transactions is not guaranteed with mixed engines and inconsistencies can result. (If mixed-engine transactions are infrequent, you can use `SET TRANSACTION ISOLATION LEVEL` to set the isolation level to `SERIALIZABLE` on a per-transaction basis as necessary.)

- If you use tables that are not transaction-safe within a transaction, changes to those tables are stored at once, regardless of the status of autocommit mode.
- If you issue a `ROLLBACK` statement after updating a nontransactional table within a transaction, an `ER_WARNING_NOT_COMPLETE_ROLLBACK` warning occurs. Changes to transaction-safe tables are rolled back, but not changes to nontransaction-safe tables.

Each transaction is stored in the binary log in one chunk, upon `COMMIT`. Transactions that are rolled back are not logged. (**Exception**: Modifications to nontransactional tables cannot be rolled back. If a transaction that is rolled back includes modifications to nontransactional tables, the entire transaction is logged with a `ROLLBACK` statement at the end to ensure that modifications to the nontransactional tables are replicated.) See Section 5.4.4, “The Binary Log”.

You can change the isolation level or access mode for transactions with the `SET TRANSACTION` statement. See Section 13.3.7, “SET TRANSACTION Statement”.

Rolling back can be a slow operation that may occur implicitly without the user having explicitly asked for it (for example, when an error occurs). Because of this, `SHOW PROCESSLIST` displays `Rolling back` in the `State` column for the session, not only for explicit rollbacks performed with the `ROLLBACK` statement but also for implicit rollbacks.

Note

In MySQL 8.0, `BEGIN`, `COMMIT`, and `ROLLBACK` are not affected by `--replicate-do-db` or `--replicate-ignore-db` rules.

When `InnoDB` performs a complete rollback of a transaction, all locks set by the transaction are released. If a single SQL statement within a transaction rolls back as a result of an error, such as a duplicate key error, locks set by the statement are preserved while the transaction remains active. This happens because `InnoDB` stores row locks in a format such that it cannot know afterward which lock was set by which statement.

If a `SELECT` statement within a transaction calls a stored function, and a statement within the stored function fails, that statement rolls back. If `ROLLBACK` is executed for the transaction subsequently, the entire transaction rolls back.

13.3.2 Statements That Cannot Be Rolled Back

Some statements cannot be rolled back. In general, these include data definition language (DDL) statements, such as those that create or drop databases, those that create, drop, or alter tables or stored routines.

You should design your transactions not to include such statements. If you issue a statement early in a transaction that cannot be rolled back, and then another statement later fails, the full effect of the transaction cannot be rolled back in such cases by issuing a `ROLLBACK` statement.

13.3.3 Statements That Cause an Implicit Commit

The statements listed in this section (and any synonyms for them) implicitly end any transaction active in the current session, as if you had done a `COMMIT` before executing the statement.

Most of these statements also cause an implicit commit after executing. The intent is to handle each such statement in its own special transaction. Transaction-control and locking statements are exceptions: If an implicit commit occurs before execution, another does not occur after.

- **Data definition language (DDL) statements that define or modify database objects.** `ALTER EVENT`, `ALTER FUNCTION`, `ALTER PROCEDURE`, `ALTER SERVER`, `ALTER TABLE`, `ALTER TABLESPACE`, `ALTER VIEW`, `CREATE DATABASE`, `CREATE EVENT`, `CREATE FUNCTION`, `CREATE INDEX`, `CREATE PROCEDURE`, `CREATE ROLE`, `CREATE SERVER`, `CREATE SPATIAL REFERENCE`

SYSTEM, CREATE TABLE, CREATE TABLESPACE, CREATE TRIGGER, CREATE VIEW, DROP DATABASE, DROP EVENT, DROP FUNCTION, DROP INDEX, DROP PROCEDURE, DROP ROLE, DROP SERVER, DROP SPATIAL REFERENCE SYSTEM, DROP TABLE, DROP TABLESPACE, DROP TRIGGER, DROP VIEW, INSTALL PLUGIN, RENAME TABLE, TRUNCATE TABLE, UNINSTALL PLUGIN.

CREATE TABLE and DROP TABLE statements do not commit a transaction if the TEMPORARY keyword is used. (This does not apply to other operations on temporary tables such as ALTER TABLE and CREATE INDEX, which do cause a commit.) However, although no implicit commit occurs, neither can the statement be rolled back, which means that the use of such statements causes transactional atomicity to be violated. For example, if you use CREATE TEMPORARY TABLE and then roll back the transaction, the table remains in existence.

The CREATE TABLE statement in InnoDB is processed as a single transaction. This means that a ROLLBACK from the user does not undo CREATE TABLE statements the user made during that transaction.

CREATE TABLE ... SELECT causes an implicit commit before and after the statement is executed when you are creating nontemporary tables. (No commit occurs for CREATE TEMPORARY TABLE ... SELECT.)

- **Statements that implicitly use or modify tables in the `mysql` database.** ALTER USER, CREATE USER, DROP USER, GRANT, RENAME USER, REVOKE, SET PASSWORD.
- **Transaction-control and locking statements.** BEGIN, LOCK TABLES, SET autocommit = 1 (if the value is not already 1), START TRANSACTION, UNLOCK TABLES.

  UNLOCK TABLES commits a transaction only if any tables currently have been locked with LOCK TABLES to acquire nontransactional table locks. A commit does not occur for UNLOCK TABLES following FLUSH TABLES WITH READ LOCK because the latter statement does not acquire table-level locks.

  Transactions cannot be nested. This is a consequence of the implicit commit performed for any current transaction when you issue a START TRANSACTION statement or one of its synonyms.

  Statements that cause an implicit commit cannot be used in an XA transaction while the transaction is in an ACTIVE state.

  The BEGIN statement differs from the use of the BEGIN keyword that starts a BEGIN ... END compound statement. The latter does not cause an implicit commit. See Section 13.6.1, “BEGIN ... END Compound Statement”.

- **Data loading statements.** LOAD DATA. LOAD DATA causes an implicit commit only for tables using the NDB storage engine.
- **Administrative statements.** ANALYZE TABLE, CACHE INDEX, CHECK TABLE, FLUSH, LOAD INDEX INTO CACHE, OPTIMIZE TABLE, REPAIR TABLE, RESET (but not RESET PERSIST).
- **Replication control statements**. START REPLICA, STOP REPLICA, RESET REPLICA, CHANGE REPLICATION SOURCE TO, CHANGE MASTER TO. The SLAVE keyword was replaced with REPLICA in MySQL 8.0.22.

13.3.4 SAVEPOINT, ROLLBACK TO SAVEPOINT, and RELEASE SAVEPOINT Statements

```
SAVEPOINT identifier
ROLLBACK [WORK] TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

InnoDB supports the SQL statements SAVEPOINT, ROLLBACK TO SAVEPOINT, RELEASE SAVEPOINT and the optional WORK keyword for ROLLBACK.

The SAVEPOINT statement sets a named transaction savepoint with a name of *identifier*. If the current transaction has a savepoint with the same name, the old savepoint is deleted and a new one is set.

The ROLLBACK TO SAVEPOINT statement rolls back a transaction to the named savepoint without terminating the transaction. Modifications that the current transaction made to rows after the savepoint was set are undone in the rollback, but InnoDB does *not* release the row locks that were stored in memory after the savepoint. (For a new inserted row, the lock information is carried by the transaction ID stored in the row; the lock is not separately stored in memory. In this case, the row lock is released in the undo.) Savepoints that were set at a later time than the named savepoint are deleted.

If the ROLLBACK TO SAVEPOINT statement returns the following error, it means that no savepoint with the specified name exists:

```
ERROR 1305 (42000): SAVEPOINT identifier does not exist
```

The RELEASE SAVEPOINT statement removes the named savepoint from the set of savepoints of the current transaction. No commit or rollback occurs. It is an error if the savepoint does not exist.

All savepoints of the current transaction are deleted if you execute a COMMIT, or a ROLLBACK that does not name a savepoint.

A new savepoint level is created when a stored function is invoked or a trigger is activated. The savepoints on previous levels become unavailable and thus do not conflict with savepoints on the new level. When the function or trigger terminates, any savepoints it created are released and the previous savepoint level is restored.

13.3.5 LOCK INSTANCE FOR BACKUP and UNLOCK INSTANCE Statements

```
LOCK INSTANCE FOR BACKUP

UNLOCK INSTANCE
```

LOCK INSTANCE FOR BACKUP acquires an instance-level *backup lock* that permits DML during an online backup while preventing operations that could result in an inconsistent snapshot.

Executing the LOCK INSTANCE FOR BACKUP statement requires the BACKUP_ADMIN privilege. The BACKUP_ADMIN privilege is automatically granted to users with the RELOAD privilege when performing an in-place upgrade to MySQL 8.0 from an earlier version.

Multiple sessions can hold a backup lock simultaneously.

UNLOCK INSTANCE releases a backup lock held by the current session. A backup lock held by a session is also released if the session is terminated.

LOCK INSTANCE FOR BACKUP prevents files from being created, renamed, or removed. REPAIR TABLE TRUNCATE TABLE, OPTIMIZE TABLE, and account management statements are blocked. See Section 13.7.1, “Account Management Statements”. Operations that modify InnoDB files that are not recorded in the InnoDB redo log are also blocked.

LOCK INSTANCE FOR BACKUP permits DDL operations that only affect user-created temporary tables. In effect, files that belong to user-created temporary tables can be created, renamed, or removed while a backup lock is held. Creation of binary log files is also permitted.

PURGE BINARY LOGS should not be issued while a LOCK INSTANCE FOR BACKUP statement is in effect for the instance, because it contravenes the rules of the backup lock by removing files from the server. From MySQL 8.0.28, this is disallowed.

A backup lock acquired by LOCK INSTANCE FOR BACKUP is independent of transactional locks and locks taken by FLUSH TABLES *tbl_name* [, *tbl_name*] ... WITH READ LOCK, and the following sequences of statements are permitted:

```
LOCK INSTANCE FOR BACKUP;
```

```
FLUSH TABLES tbl_name [, tbl_name] ... WITH READ LOCK;
UNLOCK TABLES;
UNLOCK INSTANCE;
```

```
FLUSH TABLES tbl_name [, tbl_name] ... WITH READ LOCK;
LOCK INSTANCE FOR BACKUP;
UNLOCK INSTANCE;
UNLOCK TABLES;
```

The `lock_wait_timeout` setting defines the amount of time that a `LOCK INSTANCE FOR BACKUP` statement waits to acquire a lock before giving up.

13.3.6 LOCK TABLES and UNLOCK TABLES Statements

```
LOCK TABLES
    tbl_name [[AS] alias] lock_type
    [, tbl_name [[AS] alias] lock_type] ...

lock_type: {
    READ [LOCAL]
  | [LOW_PRIORITY] WRITE
}

UNLOCK TABLES
```

MySQL enables client sessions to acquire table locks explicitly for the purpose of cooperating with other sessions for access to tables, or to prevent other sessions from modifying tables during periods when a session requires exclusive access to them. A session can acquire or release locks only for itself. One session cannot acquire locks for another session or release locks held by another session.

Locks may be used to emulate transactions or to get more speed when updating tables. This is explained in more detail in Table-Locking Restrictions and Conditions.

`LOCK TABLES` explicitly acquires table locks for the current client session. Table locks can be acquired for base tables or views. You must have the `LOCK TABLES` privilege, and the `SELECT` privilege for each object to be locked.

For view locking, `LOCK TABLES` adds all base tables used in the view to the set of tables to be locked and locks them automatically. For tables underlying any view being locked, `LOCK TABLES` checks that the view definer (for `SQL SECURITY DEFINER` views) or invoker (for all views) has the proper privileges on the tables.

If you lock a table explicitly with `LOCK TABLES`, any tables used in triggers are also locked implicitly, as described in LOCK TABLES and Triggers.

If you lock a table explicitly with `LOCK TABLES`, any tables related by a foreign key constraint are opened and locked implicitly. For foreign key checks, a shared read-only lock (`LOCK TABLES READ`) is taken on related tables. For cascading updates, a shared-nothing write lock (`LOCK TABLES WRITE`) is taken on related tables that are involved in the operation.

`UNLOCK TABLES` explicitly releases any table locks held by the current session. `LOCK TABLES` implicitly releases any table locks held by the current session before acquiring new locks.

Another use for `UNLOCK TABLES` is to release the global read lock acquired with the `FLUSH TABLES WITH READ LOCK` statement, which enables you to lock all tables in all databases. See Section 13.7.8.3, “FLUSH Statement”. (This is a very convenient way to get backups if you have a file system such as Veritas that can take snapshots in time.)

A table lock protects only against inappropriate reads or writes by other sessions. A session holding a `WRITE` lock can perform table-level operations such as `DROP TABLE` or `TRUNCATE TABLE`. For sessions holding a `READ` lock, `DROP TABLE` and `TRUNCATE TABLE` operations are not permitted.

The following discussion applies only to non-`TEMPORARY` tables. `LOCK TABLES` is permitted (but ignored) for a `TEMPORARY` table. The table can be accessed freely by the session within which it was

created, regardless of what other locking may be in effect. No lock is necessary because no other session can see the table.

- Table Lock Acquisition
- Table Lock Release
- Interaction of Table Locking and Transactions
- LOCK TABLES and Triggers
- Table-Locking Restrictions and Conditions

Table Lock Acquisition

To acquire table locks within the current session, use the `LOCK TABLES` statement, which acquires metadata locks (see Section 8.11.4, “Metadata Locking”).

The following lock types are available:

`READ [LOCAL]` lock:

- The session that holds the lock can read the table (but not write it).
- Multiple sessions can acquire a `READ` lock for the table at the same time.
- Other sessions can read the table without explicitly acquiring a `READ` lock.
- The `LOCAL` modifier enables nonconflicting `INSERT` statements (concurrent inserts) by other sessions to execute while the lock is held. (See Section 8.11.3, “Concurrent Inserts”.) However, `READ LOCAL` cannot be used if you are going to manipulate the database using processes external to the server while you hold the lock. For `InnoDB` tables, `READ LOCAL` is the same as `READ`.

`[LOW_PRIORITY] WRITE` lock:

- The session that holds the lock can read and write the table.
- Only the session that holds the lock can access the table. No other session can access it until the lock is released.
- Lock requests for the table by other sessions block while the `WRITE` lock is held.
- The `LOW_PRIORITY` modifier has no effect. In previous versions of MySQL, it affected locking behavior, but this is no longer true. It is now deprecated and its use produces a warning. Use `WRITE` without `LOW_PRIORITY` instead.

`WRITE` locks normally have higher priority than `READ` locks to ensure that updates are processed as soon as possible. This means that if one session obtains a `READ` lock and then another session requests a `WRITE` lock, subsequent `READ` lock requests wait until the session that requested the `WRITE` lock has obtained the lock and released it. (An exception to this policy can occur for small values of the `max_write_lock_count` system variable; see Section 8.11.4, “Metadata Locking”.)

If the `LOCK TABLES` statement must wait due to locks held by other sessions on any of the tables, it blocks until all locks can be acquired.

A session that requires locks must acquire all the locks that it needs in a single `LOCK TABLES` statement. While the locks thus obtained are held, the session can access only the locked tables. For example, in the following sequence of statements, an error occurs for the attempt to access `t2` because it was not locked in the `LOCK TABLES` statement:

```
mysql> LOCK TABLES t1 READ;
mysql> SELECT COUNT(*) FROM t1;
+----------+
| COUNT(*) |
```

```
+----------+
|        3 |
+----------+
mysql> SELECT COUNT(*) FROM t2;
ERROR 1100 (HY000): Table 't2' was not locked with LOCK TABLES
```

Tables in the `INFORMATION_SCHEMA` database are an exception. They can be accessed without being locked explicitly even while a session holds table locks obtained with `LOCK TABLES`.

You cannot refer to a locked table multiple times in a single query using the same name. Use aliases instead, and obtain a separate lock for the table and each alias:

```
mysql> LOCK TABLE t WRITE, t AS t1 READ;
mysql> INSERT INTO t SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> INSERT INTO t SELECT * FROM t AS t1;
```

The error occurs for the first `INSERT` because there are two references to the same name for a locked table. The second `INSERT` succeeds because the references to the table use different names.

If your statements refer to a table by means of an alias, you must lock the table using that same alias. It does not work to lock the table without specifying the alias:

```
mysql> LOCK TABLE t READ;
mysql> SELECT * FROM t AS myalias;
ERROR 1100: Table 'myalias' was not locked with LOCK TABLES
```

Conversely, if you lock a table using an alias, you must refer to it in your statements using that alias:

```
mysql> LOCK TABLE t AS myalias READ;
mysql> SELECT * FROM t;
ERROR 1100: Table 't' was not locked with LOCK TABLES
mysql> SELECT * FROM t AS myalias;
```

Table Lock Release

When the table locks held by a session are released, they are all released at the same time. A session can release its locks explicitly, or locks may be released implicitly under certain conditions.

- A session can release its locks explicitly with `UNLOCK TABLES`.
- If a session issues a `LOCK TABLES` statement to acquire a lock while already holding locks, its existing locks are released implicitly before the new locks are granted.
- If a session begins a transaction (for example, with `START TRANSACTION`), an implicit `UNLOCK TABLES` is performed, which causes existing locks to be released. (For additional information about the interaction between table locking and transactions, see Interaction of Table Locking and Transactions.)

If the connection for a client session terminates, whether normally or abnormally, the server implicitly releases all table locks held by the session (transactional and nontransactional). If the client reconnects, the locks are no longer in effect. In addition, if the client had an active transaction, the server rolls back the transaction upon disconnect, and if reconnect occurs, the new session begins with autocommit enabled. For this reason, clients may wish to disable auto-reconnect. With auto-reconnect in effect, the client is not notified if reconnect occurs but any table locks or current transaction are lost. With auto-reconnect disabled, if the connection drops, an error occurs for the next statement issued. The client can detect the error and take appropriate action such as reacquiring the locks or redoing the transaction. See Automatic Reconnection Control.

Note

If you use `ALTER TABLE` on a locked table, it may become unlocked. For example, if you attempt a second `ALTER TABLE` operation, the result may be an error `Table 'tbl_name' was not locked with LOCK TABLES`.

To handle this, lock the table again prior to the second alteration. See also Section B.3.6.1, “Problems with ALTER TABLE”.

Interaction of Table Locking and Transactions

`LOCK TABLES` and `UNLOCK TABLES` interact with the use of transactions as follows:

- `LOCK TABLES` is not transaction-safe and implicitly commits any active transaction before attempting to lock the tables.
- `UNLOCK TABLES` implicitly commits any active transaction, but only if `LOCK TABLES` has been used to acquire table locks. For example, in the following set of statements, `UNLOCK TABLES` releases the global read lock but does not commit the transaction because no table locks are in effect:

```
FLUSH TABLES WITH READ LOCK;
START TRANSACTION;
SELECT ... ;
UNLOCK TABLES;
```

- Beginning a transaction (for example, with `START TRANSACTION`) implicitly commits any current transaction and releases existing table locks.
- `FLUSH TABLES WITH READ LOCK` acquires a global read lock and not table locks, so it is not subject to the same behavior as `LOCK TABLES` and `UNLOCK TABLES` with respect to table locking and implicit commits. For example, `START TRANSACTION` does not release the global read lock. See Section 13.7.8.3, “FLUSH Statement”.
- Other statements that implicitly cause transactions to be committed do not release existing table locks. For a list of such statements, see Section 13.3.3, “Statements That Cause an Implicit Commit”.
- The correct way to use `LOCK TABLES` and `UNLOCK TABLES` with transactional tables, such as `InnoDB` tables, is to begin a transaction with `SET autocommit = 0` (not `START TRANSACTION`) followed by `LOCK TABLES`, and to not call `UNLOCK TABLES` until you commit the transaction explicitly. For example, if you need to write to table `t1` and read from table `t2`, you can do this:

```
SET autocommit=0;
LOCK TABLES t1 WRITE, t2 READ, ...;
... do something with tables t1 and t2 here ...
COMMIT;
UNLOCK TABLES;
```

  When you call `LOCK TABLES`, `InnoDB` internally takes its own table lock, and MySQL takes its own table lock. `InnoDB` releases its internal table lock at the next commit, but for MySQL to release its table lock, you have to call `UNLOCK TABLES`. You should not have `autocommit = 1`, because then `InnoDB` releases its internal table lock immediately after the call of `LOCK TABLES`, and deadlocks can very easily happen. `InnoDB` does not acquire the internal table lock at all if `autocommit = 1`, to help old applications avoid unnecessary deadlocks.
- `ROLLBACK` does not release table locks.

LOCK TABLES and Triggers

If you lock a table explicitly with `LOCK TABLES`, any tables used in triggers are also locked implicitly:

- The locks are taken as the same time as those acquired explicitly with the `LOCK TABLES` statement.
- The lock on a table used in a trigger depends on whether the table is used only for reading. If so, a read lock suffices. Otherwise, a write lock is used.
- If a table is locked explicitly for reading with `LOCK TABLES`, but needs to be locked for writing because it might be modified within a trigger, a write lock is taken rather than a read lock. (That is, an implicit write lock needed due to the table's appearance within a trigger causes an explicit read lock request for the table to be converted to a write lock request.)

Suppose that you lock two tables, `t1` and `t2`, using this statement:

```
LOCK TABLES t1 WRITE, t2 READ;
```

If `t1` or `t2` have any triggers, tables used within the triggers are also locked. Suppose that `t1` has a trigger defined like this:

```
CREATE TRIGGER t1_a_ins AFTER INSERT ON t1 FOR EACH ROW
BEGIN
  UPDATE t4 SET count = count+1
      WHERE id = NEW.id AND EXISTS (SELECT a FROM t3);
  INSERT INTO t2 VALUES(1, 2);
END;
```

The result of the `LOCK TABLES` statement is that `t1` and `t2` are locked because they appear in the statement, and `t3` and `t4` are locked because they are used within the trigger:

- `t1` is locked for writing per the `WRITE` lock request.
- `t2` is locked for writing, even though the request is for a `READ` lock. This occurs because `t2` is inserted into within the trigger, so the `READ` request is converted to a `WRITE` request.
- `t3` is locked for reading because it is only read from within the trigger.
- `t4` is locked for writing because it might be updated within the trigger.

Table-Locking Restrictions and Conditions

You can safely use `KILL` to terminate a session that is waiting for a table lock. See Section 13.7.8.4, “KILL Statement”.

`LOCK TABLES` and `UNLOCK TABLES` cannot be used within stored programs.

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

The scope of a lock generated by `LOCK TABLES` is a single MySQL server. It is not compatible with NDB Cluster, which has no way of enforcing an SQL-level lock across multiple instances of `mysqld`. You can enforce locking in an API application instead. See Section 23.2.7.10, “Limitations Relating to Multiple NDB Cluster Nodes”, for more information.

The following statements are prohibited while a `LOCK TABLES` statement is in effect: `CREATE TABLE`, `CREATE TABLE ... LIKE`, `CREATE VIEW`, `DROP VIEW`, and DDL statements on stored functions and procedures and events.

For some operations, system tables in the `mysql` database must be accessed. For example, the `HELP` statement requires the contents of the server-side help tables, and `CONVERT_TZ()` might need to read the time zone tables. The server implicitly locks the system tables for reading as necessary so that you need not lock them explicitly. These tables are treated as just described:

```
mysql.help_category
mysql.help_keyword
mysql.help_relation
mysql.help_topic
mysql.time_zone
mysql.time_zone_leap_second
mysql.time_zone_name
mysql.time_zone_transition
mysql.time_zone_transition_type
```

If you want to explicitly place a `WRITE` lock on any of those tables with a `LOCK TABLES` statement, the table must be the only one locked; no other table can be locked with the same statement.

Normally, you do not need to lock tables, because all single `UPDATE` statements are atomic; no other session can interfere with any other currently executing SQL statement. However, there are a few cases when locking tables may provide an advantage:

- If you are going to run many operations on a set of `MyISAM` tables, it is much faster to lock the tables you are going to use. Locking `MyISAM` tables speeds up inserting, updating, or deleting on them because MySQL does not flush the key cache for the locked tables until `UNLOCK TABLES` is called. Normally, the key cache is flushed after each SQL statement.

  The downside to locking the tables is that no session can update a `READ`-locked table (including the one holding the lock) and no session can access a `WRITE`-locked table other than the one holding the lock.

- If you are using tables for a nontransactional storage engine, you must use `LOCK TABLES` if you want to ensure that no other session modifies the tables between a `SELECT` and an `UPDATE`. The example shown here requires `LOCK TABLES` to execute safely:

```
LOCK TABLES trans READ, customer WRITE;
SELECT SUM(value) FROM trans WHERE customer_id=some_id;
UPDATE customer
  SET total_value=sum_from_previous_statement
  WHERE customer_id=some_id;
UNLOCK TABLES;
```

  Without `LOCK TABLES`, it is possible that another session might insert a new row in the `trans` table between execution of the `SELECT` and `UPDATE` statements.

You can avoid using `LOCK TABLES` in many cases by using relative updates (`UPDATE customer SET value=value+new_value`) or the `LAST_INSERT_ID()` function.

You can also avoid locking tables in some cases by using the user-level advisory lock functions `GET_LOCK()` and `RELEASE_LOCK()`. These locks are saved in a hash table in the server and implemented with `pthread_mutex_lock()` and `pthread_mutex_unlock()` for high speed. See Section 12.15, “Locking Functions”.

See Section 8.11.1, “Internal Locking Methods”, for more information on locking policy.

13.3.7 SET TRANSACTION Statement

```
SET [GLOBAL | SESSION] TRANSACTION
    transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic: {
    ISOLATION LEVEL level
  | access_mode
}

level: {
     REPEATABLE READ
   | READ COMMITTED
   | READ UNCOMMITTED
   | SERIALIZABLE
}

access_mode: {
     READ WRITE
   | READ ONLY
}
```

This statement specifies transaction characteristics. It takes a list of one or more characteristic values separated by commas. Each characteristic value sets the transaction isolation level or access mode. The isolation level is used for operations on `InnoDB` tables. The access mode specifies whether transactions operate in read/write or read-only mode.

In addition, `SET TRANSACTION` can include an optional `GLOBAL` or `SESSION` keyword to indicate the scope of the statement.

- Transaction Isolation Levels

- Transaction Access Mode
- Transaction Characteristic Scope

Transaction Isolation Levels

To set the transaction isolation level, use an `ISOLATION LEVEL level` clause. It is not permitted to specify multiple `ISOLATION LEVEL` clauses in the same `SET TRANSACTION` statement.

The default isolation level is `REPEATABLE READ`. Other permitted values are `READ COMMITTED`, `READ UNCOMMITTED`, and `SERIALIZABLE`. For information about these isolation levels, see Section 15.7.2.1, “Transaction Isolation Levels”.

Transaction Access Mode

To set the transaction access mode, use a `READ WRITE` or `READ ONLY` clause. It is not permitted to specify multiple access-mode clauses in the same `SET TRANSACTION` statement.

By default, a transaction takes place in read/write mode, with both reads and writes permitted to tables used in the transaction. This mode may be specified explicitly using `SET TRANSACTION` with an access mode of `READ WRITE`.

If the transaction access mode is set to `READ ONLY`, changes to tables are prohibited. This may enable storage engines to make performance improvements that are possible when writes are not permitted.

In read-only mode, it remains possible to change tables created with the `TEMPORARY` keyword using DML statements. Changes made with DDL statements are not permitted, just as with permanent tables.

The `READ WRITE` and `READ ONLY` access modes also may be specified for an individual transaction using the `START TRANSACTION` statement.

Transaction Characteristic Scope

You can set transaction characteristics globally, for the current session, or for the next transaction only:

- With the `GLOBAL` keyword:
  - The statement applies globally for all subsequent sessions.
  - Existing sessions are unaffected.
- With the `SESSION` keyword:
  - The statement applies to all subsequent transactions performed within the current session.
  - The statement is permitted within transactions, but does not affect the current ongoing transaction.
  - If executed between transactions, the statement overrides any preceding statement that sets the next-transaction value of the named characteristics.
- Without any `SESSION` or `GLOBAL` keyword:
  - The statement applies only to the next single transaction performed within the session.
  - Subsequent transactions revert to using the session value of the named characteristics.
  - The statement is not permitted within transactions:

```
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
ERROR 1568 (25001): Transaction characteristics can't be changed
while a transaction is in progress
```

A change to global transaction characteristics requires the `CONNECTION_ADMIN` privilege (or the deprecated `SUPER` privilege). Any session is free to change its session characteristics (even in the middle of a transaction), or the characteristics for its next transaction (prior to the start of that transaction).

To set the global isolation level at server startup, use the `--transaction-isolation=level` option on the command line or in an option file. Values of *`level`* for this option use dashes rather than spaces, so the permissible values are `READ-UNCOMMITTED`, `READ-COMMITTED`, `REPEATABLE-READ`, or `SERIALIZABLE`.

Similarly, to set the global transaction access mode at server startup, use the `--transaction-read-only` option. The default is `OFF` (read/write mode) but the value can be set to `ON` for a mode of read only.

For example, to set the isolation level to `REPEATABLE READ` and the access mode to `READ WRITE`, use these lines in the `[mysqld]` section of an option file:

```
[mysqld]
transaction-isolation = REPEATABLE-READ
transaction-read-only = OFF
```

At runtime, characteristics at the global, session, and next-transaction scope levels can be set indirectly using the `SET TRANSACTION` statement, as described previously. They can also be set directly using the `SET` statement to assign values to the `transaction_isolation` and `transaction_read_only` system variables:

- `SET TRANSACTION` permits optional `GLOBAL` and `SESSION` keywords for setting transaction characteristics at different scope levels.
- The `SET` statement for assigning values to the `transaction_isolation` and `transaction_read_only` system variables has syntaxes for setting these variables at different scope levels.

The following tables show the characteristic scope level set by each `SET TRANSACTION` and variable-assignment syntax.

Table 13.9 SET TRANSACTION Syntax for Transaction Characteristics

Syntax	Affected Characteristic Scope
`SET GLOBAL TRANSACTION` *`transaction_characteristic`*	Global
`SET SESSION TRANSACTION` *`transaction_characteristic`*	Session
`SET TRANSACTION` *`transaction_characteristic`*	Next transaction only

Table 13.10 SET Syntax for Transaction Characteristics

Syntax	Affected Characteristic Scope
`SET GLOBAL` *`var_name = value`*	Global
`SET @@GLOBAL.`*`var_name = value`*	Global
`SET PERSIST` *`var_name = value`*	Global
`SET @@PERSIST.`*`var_name = value`*	Global

Syntax	Affected Characteristic Scope
`SET PERSIST_ONLY var_name = value`	No runtime effect
`SET @@PERSIST_ONLY.var_name = value`	No runtime effect
`SET SESSION var_name = value`	Session
`SET @@SESSION.var_name = value`	Session
`SET var_name = value`	Session
`SET @@var_name = value`	Next transaction only

It is possible to check the global and session values of transaction characteristics at runtime:

```
SELECT @@GLOBAL.transaction_isolation, @@GLOBAL.transaction_read_only;
SELECT @@SESSION.transaction_isolation, @@SESSION.transaction_read_only;
```

13.3.8 XA Transactions

Support for XA transactions is available for the `InnoDB` storage engine. The MySQL XA implementation is based on the X/Open CAE document *Distributed Transaction Processing: The XA Specification*. This document is published by The Open Group and available at http://www.opengroup.org/public/pubs/catalog/c193.htm. Limitations of the current XA implementation are described in Section 13.3.8.3, “Restrictions on XA Transactions”.

On the client side, there are no special requirements. The XA interface to a MySQL server consists of SQL statements that begin with the `XA` keyword. MySQL client programs must be able to send SQL statements and to understand the semantics of the XA statement interface. They do not need be linked against a recent client library. Older client libraries also work.

Among the MySQL Connectors, MySQL Connector/J 5.0.0 and higher supports XA directly, by means of a class interface that handles the XA SQL statement interface for you.

XA supports distributed transactions, that is, the ability to permit multiple separate transactional resources to participate in a global transaction. Transactional resources often are RDBMSs but may be other kinds of resources.

A global transaction involves several actions that are transactional in themselves, but that all must either complete successfully as a group, or all be rolled back as a group. In essence, this extends ACID properties “up a level” so that multiple ACID transactions can be executed in concert as components of a global operation that also has ACID properties. (As with nondistributed transactions, `SERIALIZABLE` may be preferred if your applications are sensitive to read phenomena. `REPEATABLE READ` may not be sufficient for distributed transactions.)

Some examples of distributed transactions:

- An application may act as an integration tool that combines a messaging service with an RDBMS. The application makes sure that transactions dealing with message sending, retrieval, and processing that also involve a transactional database all happen in a global transaction. You can think of this as “transactional email.”
- An application performs actions that involve different database servers, such as a MySQL server and an Oracle server (or multiple MySQL servers), where actions that involve multiple servers must happen as part of a global transaction, rather than as separate transactions local to each server.
- A bank keeps account information in an RDBMS and distributes and receives money through automated teller machines (ATMs). It is necessary to ensure that ATM actions are correctly reflected in the accounts, but this cannot be done with the RDBMS alone. A global transaction manager integrates the ATM and database resources to ensure overall consistency of financial transactions.

Applications that use global transactions involve one or more Resource Managers and a Transaction Manager:

- A Resource Manager (RM) provides access to transactional resources. A database server is one kind of resource manager. It must be possible to either commit or roll back transactions managed by the RM.
- A Transaction Manager (TM) coordinates the transactions that are part of a global transaction. It communicates with the RMs that handle each of these transactions. The individual transactions within a global transaction are “branches” of the global transaction. Global transactions and their branches are identified by a naming scheme described later.

The MySQL implementation of XA enables a MySQL server to act as a Resource Manager that handles XA transactions within a global transaction. A client program that connects to the MySQL server acts as the Transaction Manager.

To carry out a global transaction, it is necessary to know which components are involved, and bring each component to a point when it can be committed or rolled back. Depending on what each component reports about its ability to succeed, they must all commit or roll back as an atomic group. That is, either all components must commit, or all components must roll back. To manage a global transaction, it is necessary to take into account that any component or the connecting network might fail.

The process for executing a global transaction uses two-phase commit (2PC). This takes place after the actions performed by the branches of the global transaction have been executed.

1. In the first phase, all branches are prepared. That is, they are told by the TM to get ready to commit. Typically, this means each RM that manages a branch records the actions for the branch in stable storage. The branches indicate whether they are able to do this, and these results are used for the second phase.
2. In the second phase, the TM tells the RMs whether to commit or roll back. If all branches indicated when they were prepared that they were able to commit, all branches are told to commit. If any branch indicated when it was prepared that it was not able to commit, all branches are told to roll back.

In some cases, a global transaction might use one-phase commit (1PC). For example, when a Transaction Manager finds that a global transaction consists of only one transactional resource (that is, a single branch), that resource can be told to prepare and commit at the same time.

13.3.8.1 XA Transaction SQL Statements

To perform XA transactions in MySQL, use the following statements:

```
XA {START|BEGIN} xid [JOIN|RESUME]

XA END xid [SUSPEND [FOR MIGRATE]]

XA PREPARE xid

XA COMMIT xid [ONE PHASE]

XA ROLLBACK xid

XA RECOVER [CONVERT XID]
```

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.

Each XA statement begins with the `XA` keyword, and most of them require an *`xid`* value. An *`xid`* is an XA transaction identifier. It indicates which transaction the statement applies to. *`xid`* values are supplied by the client, or generated by the MySQL server. An *`xid`* value has from one to three parts:

```
xid: gtrid [, bqual [, formatID ]]
```

gtrid is a global transaction identifier, *bqual* is a branch qualifier, and *formatID* is a number that identifies the format used by the *gtrid* and *bqual* values. As indicated by the syntax, *bqual* and *formatID* are optional. The default *bqual* value is `''` if not given. The default *formatID* value is 1 if not given.

gtrid and *bqual* must be string literals, each up to 64 bytes (not characters) long. *gtrid* and *bqual* can be specified in several ways. You can use a quoted string (`'ab'`), hex string (`X'6162'`, `0x6162`), or bit value (`b'nnnn'`).

formatID is an unsigned integer.

The *gtrid* and *bqual* values are interpreted in bytes by the MySQL server's underlying XA support routines. However, while an SQL statement containing an XA statement is being parsed, the server works with some specific character set. To be safe, write *gtrid* and *bqual* as hex strings.

xid values typically are generated by the Transaction Manager. Values generated by one TM must be different from values generated by other TMs. A given TM must be able to recognize its own *xid* values in a list of values returned by the `XA RECOVER` statement.

`XA START` *xid* starts an XA transaction with the given *xid* value. Each XA transaction must have a unique *xid* value, so the value must not currently be used by another XA transaction. Uniqueness is assessed using the *gtrid* and *bqual* values. All following XA statements for the XA transaction must be specified using the same *xid* value as that given in the `XA START` statement. If you use any of those statements but specify an *xid* value that does not correspond to some existing XA transaction, an error occurs.

Beginning with MySQL 8.0.31, `XA START`, `XA BEGIN`, `XA END`, `XA COMMIT`, and `XA ROLLBACK` statements are not filtered by the default database when the server is running with `--replicate-do-db` or `--replicate-ignore-db`.

One or more XA transactions can be part of the same global transaction. All XA transactions within a given global transaction must use the same *gtrid* value in the *xid* value. For this reason, *gtrid* values must be globally unique so that there is no ambiguity about which global transaction a given XA transaction is part of. The *bqual* part of the *xid* value must be different for each XA transaction within a global transaction. (The requirement that *bqual* values be different is a limitation of the current MySQL XA implementation. It is not part of the XA specification.)

The `XA RECOVER` statement returns information for those XA transactions on the MySQL server that are in the `PREPARED` state. (See Section 13.3.8.2, “XA Transaction States”.) The output includes a row for each such XA transaction on the server, regardless of which client started it.

`XA RECOVER` requires the `XA_RECOVER_ADMIN` privilege. This privilege requirement prevents users from discovering the XID values for outstanding prepared XA transactions other than their own. It does not affect normal commit or rollback of an XA transaction because the user who started it knows its XID.

`XA RECOVER` output rows look like this (for an example *xid* value consisting of the parts `'abc'`, `'def'`, and `7`):

```
mysql> XA RECOVER;
+----------+--------------+--------------+--------+
| formatID | gtrid_length | bqual_length | data   |
+----------+--------------+--------------+--------+
|        7 |            3 |            3 | abcdef |
+----------+--------------+--------------+--------+
```

The output columns have the following meanings:

- `formatID` is the *formatID* part of the transaction *xid*
- `gtrid_length` is the length in bytes of the *gtrid* part of the *xid*
- `bqual_length` is the length in bytes of the *bqual* part of the *xid*

- `data` is the concatenation of the *`gtrid`* and *`bqual`* parts of the *`xid`*

XID values may contain nonprintable characters. `XA RECOVER` permits an optional `CONVERT XID` clause so that clients can request XID values in hexadecimal.

13.3.8.2 XA Transaction States

An XA transaction progresses through the following states:

1. Use `XA START` to start an XA transaction and put it in the `ACTIVE` state.
2. For an `ACTIVE` XA transaction, issue the SQL statements that make up the transaction, and then issue an `XA END` statement. `XA END` puts the transaction in the `IDLE` state.
3. For an `IDLE` XA transaction, you can issue either an `XA PREPARE` statement or an `XA COMMIT ... ONE PHASE` statement:
   - `XA PREPARE` puts the transaction in the `PREPARED` state. An `XA RECOVER` statement at this point includes the transaction's *`xid`* value in its output, because `XA RECOVER` lists all XA transactions that are in the `PREPARED` state.
   - `XA COMMIT ... ONE PHASE` prepares and commits the transaction. The *`xid`* value is not listed by `XA RECOVER` because the transaction terminates.
4. For a `PREPARED` XA transaction, you can issue an `XA COMMIT` statement to commit and terminate the transaction, or `XA ROLLBACK` to roll back and terminate the transaction.

Here is a simple XA transaction that inserts a row into a table as part of a global transaction:

```
mysql> XA START 'xatest';
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO mytable (i) VALUES(10);
Query OK, 1 row affected (0.04 sec)

mysql> XA END 'xatest';
Query OK, 0 rows affected (0.00 sec)

mysql> XA PREPARE 'xatest';
Query OK, 0 rows affected (0.00 sec)

mysql> XA COMMIT 'xatest';
Query OK, 0 rows affected (0.00 sec)
```

In MySQL 8.0.28 and earlier, within the context of a given client connection, XA transactions and local (non-XA) transactions are mutually exclusive. For example, if `XA START` has been issued to begin an XA transaction, a local transaction cannot be started until the XA transaction has been committed or rolled back. Conversely, if a local transaction has been started with `START TRANSACTION`, no XA statements can be used until the transaction has been committed or rolled back.

MySQL 8.0.29 and later supports detached XA transactions, enabled by the `xa_detach_on_prepare` system variable (`ON` by default). Detached transactions are disconnected from the current session following execution of `XA PREPARE` (and can be committed or rolled back by another connection). This means that the current session is free to start a new local transaction or XA transaction without having to wait for the prepared XA transaction to be committed or rolled back.

When XA transactions are detached, a connection has no special knowledge of any XA transaction that it has prepared. If the current session tries to commit or roll back a given XA transaction (even one which it prepared) after another connection has already done so, the attempt is rejected with an invalid XID error (`ER_XAER_NOTA`) since the requested *`xid`* no longer exists.

Note

Detached XA transactions cannot use temporary tables.

When detached XA transactions are disabled (`xa_detach_on_prepare` set to `OFF`), an XA transaction remains connected until it is committed or rolled back by the originating connection, as described previously for MySQL 8.0.28 and earlier. Disabling detached XA transactions is not recommended for a MySQL server instance used in group replication; see Server Instance Configuration, for more information.

If an XA transaction is in the `ACTIVE` state, you cannot issue any statements that cause an implicit commit. That would violate the XA contract because you could not roll back the XA transaction. Trying to execute such a statement raises the following error:

```
ERROR 1399 (XAE07): XAER_RMFAIL: The command cannot be executed
when global transaction is in the ACTIVE state
```

Statements to which the preceding remark applies are listed at Section 13.3.3, “Statements That Cause an Implicit Commit”.

13.3.8.3 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.

Note

Prior to MySQL 5.7.7, XA transactions were not compatible with replication at all. This was because an XA transaction that was in `PREPARED` state would be rolled back on clean server shutdown or client disconnect. Similarly, an XA transaction that was in `PREPARED` state would still exist in `PREPARED` state in case the server was shutdown abnormally and then started again, but the contents of the transaction could not be written to the binary log. In both of these situations the XA transaction could not be replicated correctly.

13.4 Replication Statements

Replication can be controlled through the SQL interface using the statements described in this section. Statements are split into a group which controls source servers, a group which controls replica servers, and a group which can be applied to any replication servers.

13.4.1 SQL Statements for Controlling Source Servers

This section discusses statements for managing replication source servers. Section 13.4.2, “SQL Statements for Controlling Replica Servers”, discusses statements for managing replica servers.

In addition to the statements described here, the following `SHOW` statements are used with source servers in replication. For information about these statements, see Section 13.7.7, “SHOW Statements”.

- `SHOW BINARY LOGS`
- `SHOW BINLOG EVENTS`
- `SHOW MASTER STATUS`
- `SHOW REPLICAS` (or before MySQL 8.0.22, `SHOW SLAVE HOSTS`)

13.4.1.1 PURGE BINARY LOGS Statement

```
PURGE { BINARY | MASTER } LOGS {
    TO 'log_name'
  | BEFORE datetime_expr
}
```

The binary log is a set of files that contain information about data modifications made by the MySQL server. The log consists of a set of binary log files, plus an index file (see Section 5.4.4, “The Binary Log”).

The `PURGE BINARY LOGS` statement deletes all the binary log files listed in the log index file prior to the specified log file name or date. `BINARY` and `MASTER` are synonyms. Deleted log files also are removed from the list recorded in the index file, so that the given log file becomes the first in the list.

`PURGE BINARY LOGS` requires the `BINLOG_ADMIN` privilege. This statement has no effect if the server was not started with the `--log-bin` option to enable binary logging.

Examples:

```
PURGE BINARY LOGS TO 'mysql-bin.010';
PURGE BINARY LOGS BEFORE '2019-04-02 22:46:26';
```

The `BEFORE` variant's *`datetime_expr`* argument should evaluate to a `DATETIME` value (a value in *`'YYYY-MM-DD hh:mm:ss'`* format).

`PURGE BINARY LOGS` is safe to run while replicas are replicating. You need not stop them. If you have an active replica that currently is reading one of the log files you are trying to delete, this statement does not delete the log file that is in use or any log files later than that one, but it deletes any earlier log files. A warning message is issued in this situation. However, if a replica is not connected and you happen to purge one of the log files it has yet to read, the replica cannot replicate after it reconnects.

`PURGE BINARY LOGS` should not be issued while a `LOCK INSTANCE FOR BACKUP` statement is in effect for the instance, because it contravenes the rules of the backup lock by removing files from the server. From MySQL 8.0.28, this is disallowed.

To safely purge binary log files, follow this procedure:

1. On each replica, use `SHOW REPLICA STATUS` to check which log file it is reading.
2. Obtain a listing of the binary log files on the source with `SHOW BINARY LOGS`.
3. Determine the earliest log file among all the replicas. This is the target file. If all the replicas are up to date, this is the last log file on the list.

4. Make a backup of all the log files you are about to delete. (This step is optional, but always advisable.)

5. Purge all log files up to but not including the target file.

`PURGE BINARY LOGS TO` and `PURGE BINARY LOGS BEFORE` both fail with an error when binary log files listed in the `.index` file had been removed from the system by some other means (such as using `rm` on Linux). (Bug #18199, Bug #18453) To handle such errors, edit the `.index` file (which is a simple text file) manually to ensure that it lists only the binary log files that are actually present, then run again the `PURGE BINARY LOGS` statement that failed.

Binary log files are automatically removed after the server's binary log expiration period. Removal of the files can take place at startup and when the binary log is flushed. The default binary log expiration period is 30 days. You can specify an alternative expiration period using the `binlog_expire_logs_seconds` system variable. If you are using replication, you should specify an expiration period that is no lower than the maximum amount of time your replicas might lag behind the source.

13.4.1.2 RESET MASTER Statement

```
RESET MASTER [TO binary_log_file_index_number]
```

Warning

Use this statement with caution to ensure you do not lose any wanted binary log file data and GTID execution history.

`RESET MASTER` requires the `RELOAD` privilege.

For a server where binary logging is enabled (`log_bin` is `ON`), `RESET MASTER` deletes all existing binary log files and resets the binary log index file, resetting the server to its state before binary logging was started. A new empty binary log file is created so that binary logging can be restarted.

For a server where GTIDs are in use (`gtid_mode` is `ON`), issuing `RESET MASTER` resets the GTID execution history. The value of the `gtid_purged` system variable is set to an empty string (`''`), the global value (but not the session value) of the `gtid_executed` system variable is set to an empty string, and the `mysql.gtid_executed` table is cleared (see mysql.gtid_executed Table). If the GTID-enabled server has binary logging enabled, `RESET MASTER` also resets the binary log as described above. Note that `RESET MASTER` is the method to reset the GTID execution history even if the GTID-enabled server is a replica where binary logging is disabled; `RESET REPLICA` has no effect on the GTID execution history. For more information on resetting the GTID execution history, see Resetting the GTID Execution History.

Issuing `RESET MASTER` without the optional `TO` clause deletes all binary log files listed in the index file, resets the binary log index file to be empty, and creates a new binary log file starting at `1`. Use the optional `TO` clause to start the binary log file index from a number other than `1` after the reset.

Using `RESET MASTER` with the `TO` clause to specify a binary log file index number to start from simplifies failover by providing a single statement alternative to the `FLUSH BINARY LOGS` and `PURGE BINARY LOGS TO` statements. Check that you are using a reasonable value for the index number. If you enter an incorrect value, you can correct this by issuing another `RESET MASTER` statement with or without the `TO` clause. If you do not correct a value that is out of range, the server cannot be restarted.

The following example demonstrates `TO` clause usage:

```
RESET MASTER TO 1234;

SHOW BINARY LOGS;
+------------------+-----------+-----------+
| Log_name         | File_size | Encrypted |
+------------------+-----------+-----------+
| source-bin.001234 |       154 | No        |
+------------------+-----------+-----------+
```

Important

The effects of `RESET MASTER` without the `TO` clause differ from those of `PURGE BINARY LOGS` in 2 key ways:

1. `RESET MASTER` removes *all* binary log files that are listed in the index file, leaving only a single, empty binary log file with a numeric suffix of `.000001`, whereas the numbering is not reset by `PURGE BINARY LOGS`.
2. `RESET MASTER` is *not* intended to be used while any replicas are running. The behavior of `RESET MASTER` when used while replicas are running is undefined (and thus unsupported), whereas `PURGE BINARY LOGS` may be safely used while replicas are running.

See also Section 13.4.1.1, “PURGE BINARY LOGS Statement”.

`RESET MASTER` without the `TO` clause can prove useful when you first set up a source and replica, so that you can verify the setup as follows:

1. Start the source and replica, and start replication (see Section 17.1.2, “Setting Up Binary Log File Position Based Replication”).
2. Execute a few test queries on the source.
3. Check that the queries were replicated to the replica.
4. When replication is running correctly, issue `STOP REPLICA` followed by `RESET REPLICA` on the replica, then verify that no unwanted data from the test queries exists on the replica.
5. Issue `RESET MASTER` on the source to clean up the test queries.

After verifying the setup, resetting the source and replica and ensuring that no unwanted data or binary log files generated by testing remain on the source or replica, you can start the replica and begin replicating.

13.4.1.3 SET sql_log_bin Statement

```
SET sql_log_bin = {OFF|ON}
```

The `sql_log_bin` variable controls whether logging to the binary log is enabled for the current session (assuming that the binary log itself is enabled). The default value is `ON`. To disable or enable binary logging for the current session, set the session `sql_log_bin` variable to `OFF` or `ON`.

Set this variable to `OFF` for a session to temporarily disable binary logging while making changes to the source that you do not want replicated to the replica.

Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

It is not possible to set the session value of `sql_log_bin` within a transaction or subquery.

Setting this variable to `OFF` prevents new GTIDs from being assigned to transactions in the binary log. If you are using GTIDs for replication, this means that even when binary logging is later enabled again, the GTIDs written into the log from this point do not account for any transactions that occurred in the meantime, so in effect those transactions are lost.

`mysqldump` adds a `SET @@SESSION.sql_log_bin=0` statement to a dump file from a server where GTIDs are in use, which disables binary logging while the dump file is being reloaded. The statement prevents new GTIDs from being generated and assigned to the transactions in the dump file as they are executed, so that the original GTIDs for the transactions are used.

13.4.2 SQL Statements for Controlling Replica Servers

This section discusses statements for managing replica servers. Section 13.4.1, “SQL Statements for Controlling Source Servers”, discusses statements for managing source servers.

In addition to the statements described here, `SHOW REPLICA STATUS` and `SHOW RELAYLOG EVENTS` are also used with replicas. For information about these statements, see Section 13.7.7.35, “SHOW REPLICA STATUS Statement”, and Section 13.7.7.32, “SHOW RELAYLOG EVENTS Statement”.

13.4.2.1 CHANGE MASTER TO Statement

```
CHANGE MASTER TO option [, option] ... [ channel_option ]

option: {
    MASTER_BIND = 'interface_name'
  | MASTER_HOST = 'host_name'
  | MASTER_USER = 'user_name'
  | MASTER_PASSWORD = 'password'
  | MASTER_PORT = port_num
  | PRIVILEGE_CHECKS_USER = {'account' | NULL}
  | REQUIRE_ROW_FORMAT = {0|1}
  | REQUIRE_TABLE_PRIMARY_KEY_CHECK = {STREAM | ON | OFF}
  | ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = {OFF | LOCAL | uuid}
  | MASTER_LOG_FILE = 'source_log_name'
  | MASTER_LOG_POS = source_log_pos
  | MASTER_AUTO_POSITION = {0|1}
  | RELAY_LOG_FILE = 'relay_log_name'
  | RELAY_LOG_POS = relay_log_pos
  | MASTER_HEARTBEAT_PERIOD = interval
  | MASTER_CONNECT_RETRY = interval
  | MASTER_RETRY_COUNT = count
  | SOURCE_CONNECTION_AUTO_FAILOVER = {0|1}
  | MASTER_DELAY = interval
  | MASTER_COMPRESSION_ALGORITHMS = 'algorithm[,algorithm][,algorithm]'
  | MASTER_ZSTD_COMPRESSION_LEVEL = level
  | MASTER_SSL = {0|1}
  | MASTER_SSL_CA = 'ca_file_name'
  | MASTER_SSL_CAPATH = 'ca_directory_name'
  | MASTER_SSL_CERT = 'cert_file_name'
  | MASTER_SSL_CRL = 'crl_file_name'
  | MASTER_SSL_CRLPATH = 'crl_directory_name'
  | MASTER_SSL_KEY = 'key_file_name'
  | MASTER_SSL_CIPHER = 'cipher_list'
  | MASTER_SSL_VERIFY_SERVER_CERT = {0|1}
  | MASTER_TLS_VERSION = 'protocol_list'
  | MASTER_TLS_CIPHERSUITES = 'ciphersuite_list'
  | MASTER_PUBLIC_KEY_PATH = 'key_file_name'
  | GET_MASTER_PUBLIC_KEY = {0|1}
  | NETWORK_NAMESPACE = 'namespace'
  | IGNORE_SERVER_IDS = (server_id_list),
  | GTID_ONLY = {0|1}
}

channel_option:
    FOR CHANNEL channel

server_id_list:
    [server_id [, server_id] ... ]
```

`CHANGE MASTER TO` changes the parameters that the replica server uses for connecting to the source and for reading data from the source. It also updates the contents of the replication metadata repositories (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”). From MySQL 8.0.23, use `CHANGE REPLICATION SOURCE TO` in place of `CHANGE MASTER TO`, which is deprecated from that release. In releases before MySQL 8.0.23, use `CHANGE MASTER TO`.

`CHANGE MASTER TO` requires the `REPLICATION_SLAVE_ADMIN` privilege (or the deprecated `SUPER` privilege).

Options that you do not specify on a `CHANGE MASTER TO` statement retain their value, except as indicated in the following discussion. In most cases, there is therefore no need to specify options that do not change.

Values used for SOURCE_HOST and other CHANGE REPLICATION SOURCE TO options are checked for linefeed (\n or 0x0A) characters. The presence of such characters in these values causes the statement to fail with an error.

The optional FOR CHANNEL *channel* clause enables you to name which replication channel the statement applies to. Providing a FOR CHANNEL *channel* clause applies the CHANGE MASTER TO statement to a specific replication channel, and is used to add a new channel or modify an existing channel. For example, to add a new channel called channel2:

```
CHANGE MASTER TO MASTER_HOST=host1, MASTER_PORT=3002 FOR CHANNEL 'channel2'
```

If no clause is named and no extra channels exist, a CHANGE MASTER TO statement applies to the default channel, whose name is the empty string (""). When you have set up multiple replication channels, every CHANGE MASTER TO statement must name a channel using the FOR CHANNEL *channel* clause. See Section 17.2.2, "Replication Channels" for more information.

For some of the options of the CHANGE MASTER TO statement, you must issue a STOP SLAVE statement prior to issuing a CHANGE MASTER TO statement (and a START SLAVE statement afterwards). Sometimes, you only need to stop the replication SQL (applier) thread or the replication I/O (receiver) thread, not both:

- When the applier thread is stopped, you can execute CHANGE MASTER TO using any combination that is otherwise allowed of RELAY_LOG_FILE, RELAY_LOG_POS, and MASTER_DELAY options, even if the replication receiver thread is running. No other options may be used with this statement when the receiver thread is running.
- When the receiver thread is stopped, you can execute CHANGE MASTER TO using any of the options for this statement (in any allowed combination) *except* RELAY_LOG_FILE, RELAY_LOG_POS, MASTER_DELAY, or MASTER_AUTO_POSITION = 1 even when the applier thread is running.
- Both the receiver thread and the applier thread must be stopped before issuing a CHANGE MASTER TO statement that employs MASTER_AUTO_POSITION = 1, GTID_ONLY = 1, or ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS.

You can check the current state of the replication applier thread and replication receiver thread using SHOW SLAVE STATUS. Note that the Group Replication applier channel (group_replication_applier) has no receiver thread, only an applier thread.

CHANGE MASTER TO statements have a number of side-effects and interactions that you should be aware of beforehand:

- CHANGE MASTER TO causes an implicit commit of an ongoing transaction. See Section 13.3.3, "Statements That Cause an Implicit Commit".
- CHANGE MASTER TO causes the previous values for MASTER_HOST, MASTER_PORT, MASTER_LOG_FILE, and MASTER_LOG_POS to be written to the error log, along with other information about the replica's state prior to execution.
- If you are using statement-based replication and temporary tables, it is possible for a CHANGE MASTER TO statement following a STOP SLAVE statement to leave behind temporary tables on the replica. A warning (ER_WARN_OPEN_TEMP_TABLES_MUST_BE_ZERO) is issued whenever this occurs. You can avoid this in such cases by making sure that the value of the Replica_open_temp_tables or Slave_open_temp_tables system status variable is equal to 0 prior to executing such a CHANGE MASTER TO statement.
- When using a multithreaded replica (replica_parallel_workers > 0 or slave_parallel_workers > 0), stopping the replica can cause gaps in the sequence of transactions that have been executed from the relay log, regardless of whether the replica was stopped intentionally or otherwise. When such gaps exist, issuing CHANGE MASTER TO fails. The solution in this situation is to issue START SLAVE UNTIL SQL_AFTER_MTS_GAPS which ensures that the gaps are closed. From MySQL 8.0.26, the process of checking for gaps in the sequence of transactions is skipped entirely when GTID-based replication and GTID auto-positioning are in use,

because gaps in transactions can be resolved using GTID auto-positioning. In that situation, `CHANGE MASTER TO` can still be used.

The following options are available for `CHANGE MASTER TO` statements:

`ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS = {OFF | LOCAL | uuid}`

Makes the replication channel assign a GTID to replicated transactions that do not have one, enabling replication from a source that does not use GTID-based replication, to a replica that does. For a multi-source replica, you can have a mix of channels that use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, and channels that do not. The default is `OFF`, meaning that the feature is not used.

`LOCAL` assigns a GTID including the replica's own UUID (the `server_uuid` setting). *`uuid`* assigns a GTID including the specified UUID, such as the `server_uuid` setting for the replication source server. Using a nonlocal UUID lets you differentiate between transactions that originated on the replica and transactions that originated on the source, and for a multi-source replica, between transactions that originated on different sources. The UUID you choose only has significance for the replica's own use. If any of the transactions sent by the source do have a GTID already, that GTID is retained.

Channels specific to Group Replication cannot use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, but an asynchronous replication channel for another source on a server instance that is a Group Replication group member can do so. In that case, do not specify the Group Replication group name as the UUID for creating the GTIDs.

To set `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` to `LOCAL` or *`uuid`*, the replica must have `gtid_mode=ON` set, and this cannot be changed afterwards. This option is for use with a source that has binary log file position based replication, so `MASTER_AUTO_POSITION=1` cannot be set for the channel. Both the replication SQL thread and the replication I/O (receiver) thread must be stopped before setting this option.

Important

A replica set up with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on any channel cannot be promoted to replace the replication source server in the event that a failover is required, and a backup taken from the replica cannot be used to restore the replication source server. The same restriction applies to replacing or restoring other replicas that use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on any channel.

For further restrictions and information, see Section 17.1.3.6, “Replication From a Source Without GTIDs to a Replica With GTIDs”.

`GET_MASTER_PUBLIC_KEY = {0|1}`

Enables RSA key pair-based password exchange by requesting the public key from the source. The option is disabled by default.

This option applies to replicas that authenticate with the `caching_sha2_password` authentication plugin. For connections by accounts that authenticate using this plugin, the source does not send the public key unless requested, so it must be requested or specified in the client. If `MASTER_PUBLIC_KEY_PATH` is given and specifies a valid public key file, it takes precedence over `GET_MASTER_PUBLIC_KEY`. If you are using a replication user account that authenticates with the `caching_sha2_password` plugin (which is the default from MySQL 8.0), and you are not using a secure connection, you must specify either this option or the `MASTER_PUBLIC_KEY_PATH` option to provide the RSA public key to the replica.

`GTID_ONLY = {0|1}`

Stops the replication channel persisting file names and file positions in the replication metadata repositories. `GTID_ONLY` is available as of MySQL 8.0.27. The `GTID_ONLY` option is disabled by default for asynchronous replication channels, but it is enabled by default for Group Replication channels, and it cannot be disabled for them.

For replication channels with this setting, in-memory file positions are still tracked, and file positions can still be observed for debugging purposes in error messages and through interfaces such as `SHOW REPLICA STATUS` statements (where they are shown as being invalid if they are out of date). However, the writes and reads required to persist and check the file positions are avoided in situations where GTID-based replication does not actually require them, including the transaction queuing and application process.

This option can be used only if both the replication SQL (applier) thread and replication I/O (receiver) thread are stopped. To set `GTID_ONLY = 1` for a replication channel, GTIDs must be in use on the server (`gtid_mode = ON`), and row-based binary logging must be in use on the source (statement-based replication is not supported). The options `REQUIRE_ROW_FORMAT = 1` and `SOURCE_AUTO_POSITION = 1` must be set for the replication channel.

When `GTID_ONLY = 1` is set, the replica uses `replica_parallel_workers=1` if that system variable is set to zero for the server, so it is always technically a multi-threaded applier. This is because a multi-threaded applier uses saved positions rather than the replication metadata repositories to locate the start of a transaction that it needs to reapply.

If you disable `GTID_ONLY` after setting it, the existing relay logs are deleted and the existing known binary log file positions are persisted, even if they are stale. The file positions for the binary log and relay log in the replication metadata repositories might be invalid, and a warning is returned if this is the case. Provided that `SOURCE_AUTO_POSITION` is still enabled, GTID auto-positioning is used to provide the correct positioning.

If you also disable `SOURCE_AUTO_POSITION`, the file positions for the binary log and relay log in the replication metadata repositories are used for positioning if they are valid. If they are marked as invalid, you must provide a valid binary log file name and position (`SOURCE_LOG_FILE` and `SOURCE_LOG_POS`). If you also provide a relay log file name and position (`RELAY_LOG_FILE` and

RELAY_LOG_POS), the relay logs are preserved and the applier position is set to the stated position. GTID auto-skip ensures that any transactions already applied are skipped even if the eventual applier position is not correct.

`IGNORE_SERVER_IDS = (server_id_list)`

Makes the replica ignore events originating from the specified servers. The option takes a comma-separated list of 0 or more server IDs. Log rotation and deletion events from the servers are not ignored, and are recorded in the relay log.

In circular replication, the originating server normally acts as the terminator of its own events, so that they are not applied more than once. Thus, this option is useful in circular replication when one of the servers in the circle is removed. Suppose that you have a circular replication setup with 4 servers, having server IDs 1, 2, 3, and 4, and server 3 fails. When bridging the gap by starting replication from server 2 to server 4, you can include `IGNORE_SERVER_IDS = (3)` in the `CHANGE MASTER TO` statement that you issue on server 4 to tell it to use server 2 as its source instead of server 3. Doing so causes it to ignore and not to propagate any statements that originated with the server that is no longer in use.

If `IGNORE_SERVER_IDS` contains the server's own ID and the server was started with the `--replicate-same-server-id` option enabled, an error results.

Note

When global transaction identifiers (GTIDs) are used for replication, transactions that have already been applied are automatically ignored, so the `IGNORE_SERVER_IDS` function is not required and is deprecated. If `gtid_mode=ON` is set for the server, a deprecation warning is issued if you include the `IGNORE_SERVER_IDS` option in a `CHANGE MASTER TO` statement.

The source metadata repository and the output of `SHOW REPLICA STATUS` provide the list of servers that are currently ignored. For more information, see Section 17.2.4.2, “Replication Metadata Repositories”, and Section 13.7.7.35, “SHOW REPLICA STATUS Statement”.

If a `CHANGE MASTER TO` statement is issued without any `IGNORE_SERVER_IDS` option, any existing list is preserved. To clear the list of ignored servers, it is necessary to use the option with an empty list:

```
CHANGE MASTER TO IGNORE_SERVER_IDS = ();
```

`RESET REPLICA ALL` clears `IGNORE_SERVER_IDS`.

Note

A deprecation warning is issued if `SET GTID_MODE=ON` is issued when any channel has existing server IDs set with `IGNORE_SERVER_IDS`. Before starting