15.1.20 CREATE TABLE Statement


```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    (create_definition,...)
    [table_options]
    [partition_options]

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [(create_definition,...)]
    [table_options]
    [partition_options]
    [IGNORE | REPLACE]
    [AS] query_expression

CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    { LIKE old_tbl_name | (LIKE old_tbl_name) }

create_definition: {
    col_name column_definition
  | {INDEX | KEY} [index_name] [index_type] (key_part,...)
      [index_option] ...
  | {FULLTEXT | SPATIAL} [INDEX | KEY] [index_name] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] PRIMARY KEY
      [index_type] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] UNIQUE [INDEX | KEY]
      [index_name] [index_type] (key_part,...)
      [index_option] ...
  | [CONSTRAINT [symbol]] FOREIGN KEY
      [index_name] (col_name,...)
      reference_definition
  | check_constraint_definition
```

```
}

column_definition: {
    data_type [NOT NULL | NULL] [DEFAULT {literal | (expr)} ]
      [VISIBLE | INVISIBLE]
      [AUTO_INCREMENT] [UNIQUE [KEY]] [[PRIMARY] KEY]
      [COMMENT 'string']
      [COLLATE collation_name]
      [COLUMN_FORMAT {FIXED | DYNAMIC | DEFAULT}]
      [ENGINE_ATTRIBUTE [=] 'string']
      [SECONDARY_ENGINE_ATTRIBUTE [=] 'string']
      [STORAGE {DISK | MEMORY}]
      [reference_definition]
      [check_constraint_definition]
  | data_type
      [COLLATE collation_name]
      [GENERATED ALWAYS] AS (expr)
      [VIRTUAL | STORED] [NOT NULL | NULL]
      [VISIBLE | INVISIBLE]
      [UNIQUE [KEY]] [[PRIMARY] KEY]
      [COMMENT 'string']
      [reference_definition]
      [check_constraint_definition]
}

data_type:
    (see Chapter 13, Data Types)

key_part: {col_name [(length)] | (expr)} [ASC | DESC]

index_type:
    USING {BTREE | HASH}

index_option: {
    KEY_BLOCK_SIZE [=] value
  | index_type
  | WITH PARSER parser_name
  | COMMENT 'string'
  | {VISIBLE | INVISIBLE}
  |ENGINE_ATTRIBUTE [=] 'string'
  |SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
}

check_constraint_definition:
    [CONSTRAINT [symbol]] CHECK (expr) [[NOT] ENFORCED]

reference_definition:
    REFERENCES tbl_name (key_part,...)
      [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
      [ON DELETE reference_option]
      [ON UPDATE reference_option]
```

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

table_options:
    table_option [[,] table_option] ...

table_option: {
    AUTOEXTEND_SIZE [=] value
  | AUTO_INCREMENT [=] value
  | AVG_ROW_LENGTH [=] value
  | [DEFAULT] CHARACTER SET [=] charset_name
  | CHECKSUM [=] {0 | 1}
  | [DEFAULT] COLLATE [=] collation_name
  | COMMENT [=] 'string'
  | COMPRESSION [=] {'ZLIB' | 'LZ4' | 'NONE'}
  | CONNECTION [=] 'connect_string'
  | {DATA | INDEX} DIRECTORY [=] 'absolute path to directory'
  | DELAY_KEY_WRITE [=] {0 | 1}
  | ENCRYPTION [=] {'Y' | 'N'}
  | ENGINE [=] engine_name
  | ENGINE_ATTRIBUTE [=] 'string'
  | INSERT_METHOD [=] { NO | FIRST | LAST }
  | KEY_BLOCK_SIZE [=] value
  | MAX_ROWS [=] value
  | MIN_ROWS [=] value
  | PACK_KEYS [=] {0 | 1 | DEFAULT}
  | PASSWORD [=] 'string'
  | ROW_FORMAT [=] {DEFAULT | DYNAMIC | FIXED | COMPRESSED | REDUNDANT | COMPACT}
  | START TRANSACTION
  | SECONDARY_ENGINE_ATTRIBUTE [=] 'string'
  | STATS_AUTO_RECALC [=] {DEFAULT | 0 | 1}
  | STATS_PERSISTENT [=] {DEFAULT | 0 | 1}
  | STATS_SAMPLE_PAGES [=] value
  | tablespace_option
  | UNION [=] (tbl_name[,tbl_name]...)
}

partition_options:
    PARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY [ALGORITHM={1 | 2}] (column_list)
        | RANGE{(expr) | COLUMNS(column_list)}
        | LIST{(expr) | COLUMNS(column_list)} }
    [PARTITIONS num]
    [SUBPARTITION BY
        { [LINEAR] HASH(expr)
        | [LINEAR] KEY [ALGORITHM={1 | 2}] (column_list) }
      [SUBPARTITIONS num]
    ]
    [(partition_definition [, partition_definition] ...)]

partition_definition:
```

```
    PARTITION partition_name
        [VALUES
            {LESS THAN {(expr | value_list) | MAXVALUE}
            |
            IN (value_list)}]
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]
        [(subpartition_definition [, subpartition_definition] ...)]

subpartition_definition:
    SUBPARTITION logical_name
        [[STORAGE] ENGINE [=] engine_name]
        [COMMENT [=] 'string' ]
        [DATA DIRECTORY [=] 'data_dir']
        [INDEX DIRECTORY [=] 'index_dir']
        [MAX_ROWS [=] max_number_of_rows]
        [MIN_ROWS [=] min_number_of_rows]
        [TABLESPACE [=] tablespace_name]

tablespace_option:
    TABLESPACE tablespace_name [STORAGE DISK]
  | [TABLESPACE tablespace_name] STORAGE MEMORY

query_expression:
    SELECT ...   (Some valid select or union statement)
```

`CREATE TABLE` creates a table with the given name. You must have the `CREATE` privilege for the table.

By default, tables are created in the default database, using the `InnoDB` storage engine. An error occurs if the table exists, if there is no default database, or if the database does not exist.

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

For information about the physical representation of a table, see Section 15.1.20.1, “Files Created by CREATE TABLE”.

There are several aspects to the `CREATE TABLE` statement, described under the following topics in this section:

- Table Name
- Temporary Tables

- Table Cloning and Copying
- Column Data Types and Attributes
- Indexes, Foreign Keys, and CHECK Constraints
- Table Options
- Table Partitioning

Table Name

- ***tbl_name***

  The table name can be specified as ***db_name.tbl_name*** to create the table in a specific database. This works regardless of whether there is a default database, assuming that the database exists. If you use quoted identifiers, quote the database and table names separately. For example, write `` `mydb`.`mytbl` ``, not `` `mydb.mytbl` ``.

  Rules for permissible table names are given in Section 11.2, “Schema Object Names”.

- `IF NOT EXISTS`

  Prevents an error from occurring if the table exists. However, there is no verification that the existing table has a structure identical to that indicated by the `CREATE TABLE` statement.

Temporary Tables

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. For more information, see Section 15.1.20.2, “CREATE TEMPORARY TABLE Statement”.

Table Cloning and Copying

- `LIKE`

  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;
  ```

  For more information, see Section 15.1.20.3, “CREATE TABLE ... LIKE Statement”.

- `[AS]` ***query_expression***

To create one table from another, add a SELECT statement at the end of the CREATE TABLE statement:

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

For more information, see Section 15.1.20.4, “CREATE TABLE ... SELECT Statement”.

- IGNORE | REPLACE

  The IGNORE and REPLACE options indicate how to handle rows that duplicate unique key values when copying a table using a SELECT statement.

  For more information, see Section 15.1.20.4, “CREATE TABLE ... SELECT Statement”.

Column Data Types and Attributes

There is a hard limit of 4096 columns per table, but the effective maximum may be less for a given table and depends on the factors discussed in Section 10.4.7, “Limits on Table Column Count and Row Size”.

- ***data_type***

  data_type represents the data type in a column definition. For a full description of the syntax available for specifying column data types, as well as information about the properties of each type, see Chapter 13, *Data Types*.

  - Some attributes do not apply to all data types. AUTO_INCREMENT applies only to integer and floating-point types. Prior to MySQL 8.0.13, DEFAULT does not apply to the BLOB, TEXT, GEOMETRY, and JSON types.

  - Character data types (CHAR, VARCHAR, the TEXT types, ENUM, SET, and any synonyms) can include CHARACTER SET to specify the character set for the column. CHARSET is a synonym for CHARACTER SET. A collation for the character set can be specified with the COLLATE attribute, along with any other attributes. For details, see Chapter 12, *Character Sets, Collations, Unicode*. Example:

    ```
    CREATE TABLE t (c CHAR(20) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin);
    ```

    MySQL 8.0 interprets length specifications in character column definitions in characters. Lengths for BINARY and VARBINARY are in bytes.

  - For CHAR, VARCHAR, BINARY, and VARBINARY columns, indexes can be created that use only the leading part of column values, using ***col_name*(*length*)** syntax to specify an index prefix

length. BLOB and TEXT columns also can be indexed, but a prefix length *must* be given. Prefix lengths are given in characters for nonbinary string types and in bytes for binary string types. That is, index entries consist of the first ***length*** characters of each column value for CHAR, VARCHAR, and TEXT columns, and the first ***length*** bytes of each column value for BINARY, VARBINARY, and BLOB columns. Indexing only a prefix of column values like this can make the index file much smaller. For additional information about index prefixes, see Section 15.1.15, “CREATE INDEX Statement”.

Only the InnoDB and MyISAM storage engines support indexing on BLOB and TEXT columns. For example:

```
CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
```

If a specified index prefix exceeds the maximum column data type size, CREATE TABLE handles the index as follows:

- For a nonunique index, either an error occurs (if strict SQL mode is enabled), or the index length is reduced to lie within the maximum column data type size and a warning is produced (if strict SQL mode is not enabled).
- For a unique index, an error occurs regardless of SQL mode because reducing the index length might enable insertion of nonunique entries that do not meet the specified uniqueness requirement.

- JSON columns cannot be indexed. You can work around this restriction by creating an index on a generated column that extracts a scalar value from the JSON column. See Indexing a Generated Column to Provide a JSON Column Index, for a detailed example.

- NOT NULL | NULL

  If neither NULL nor NOT NULL is specified, the column is treated as though NULL had been specified.

  In MySQL 8.0, only the InnoDB, MyISAM, and MEMORY storage engines support indexes on columns that can have NULL values. In other cases, you must declare indexed columns as NOT NULL or an error results.

- DEFAULT

  Specifies a default value for a column. For more information about default value handling, including the case that a column definition includes no explicit DEFAULT value, see Section 13.6, “Data Type Default Values”.

If the NO_ZERO_DATE or NO_ZERO_IN_DATE SQL mode is enabled and a date-valued default is not correct according to that mode, CREATE TABLE produces a warning if strict SQL mode is not enabled and an error if strict mode is enabled. For example, with NO_ZERO_IN_DATE enabled, c1 DATE DEFAULT '2010-00-00' produces a warning.

- VISIBLE, INVISIBLE

  Specify column visibility. The default is VISIBLE if neither keyword is present. A table must have at least one visible column. Attempting to make all columns invisible produces an error. For more information, see Section 15.1.20.10, “Invisible Columns”.

  The VISIBLE and INVISIBLE keywords are available as of MySQL 8.0.23. Prior to MySQL 8.0.23, all columns are visible.

- AUTO_INCREMENT

  An integer or floating-point column can have the additional attribute AUTO_INCREMENT. When you insert a value of NULL (recommended) or 0 into an indexed AUTO_INCREMENT column, the column is set to the next sequence value. Typically this is ***value***+1, where ***value*** is the largest value for the column currently in the table. AUTO_INCREMENT sequences begin with 1.

  To retrieve an AUTO_INCREMENT value after inserting a row, use the LAST_INSERT_ID() SQL function or the mysql_insert_id() C API function. See Section 14.15, “Information Functions”, and mysql_insert_id().

  If the NO_AUTO_VALUE_ON_ZERO SQL mode is enabled, you can store 0 in AUTO_INCREMENT columns as 0 without generating a new sequence value. See Section 7.1.11, “Server SQL Modes”.

  There can be only one AUTO_INCREMENT column per table, it must be indexed, and it cannot have a DEFAULT value. An AUTO_INCREMENT column works properly only if it contains only positive values. Inserting a negative number is regarded as inserting a very large positive number. This is done to avoid precision problems when numbers “wrap” over from positive to negative and also to ensure that you do not accidentally get an AUTO_INCREMENT column that contains 0.

  For MyISAM tables, you can specify an AUTO_INCREMENT secondary column in a multiple-column key. See Section 5.6.9, “Using AUTO_INCREMENT”.

  To make MySQL compatible with some ODBC applications, you can find the AUTO_INCREMENT value for the last inserted row with the following query:

```
SELECT * FROM tbl_name WHERE auto_col IS NULL
```

This method requires that sql_auto_is_null variable is not set to 0. See Section 7.1.8, “Server System Variables”.

For information about InnoDB and AUTO_INCREMENT, see Section 17.6.1.6, “AUTO_INCREMENT Handling in InnoDB”. For information about AUTO_INCREMENT and MySQL Replication, see Section 19.5.1.1, “Replication and AUTO_INCREMENT”.

- COMMENT

  A comment for a column can be specified with the COMMENT option, up to 1024 characters long. The comment is displayed by the SHOW CREATE TABLE and SHOW FULL COLUMNS statements. It is also shown in the COLUMN_COMMENT column of the Information Schema COLUMNS table.

- COLUMN_FORMAT

  In NDB Cluster, it is also possible to specify a data storage format for individual columns of NDB tables using COLUMN_FORMAT. Permissible column formats are FIXED, DYNAMIC, and DEFAULT. FIXED is used to specify fixed-width storage, DYNAMIC permits the column to be variable-width, and DEFAULT causes the column to use fixed-width or variable-width storage as determined by the column's data type (possibly overridden by a ROW_FORMAT specifier).

  For NDB tables, the default value for COLUMN_FORMAT is FIXED.

  In NDB Cluster, the maximum possible offset for a column defined with COLUMN_FORMAT=FIXED is 8188 bytes. For more information and possible workarounds, see Section 25.2.7.5, “Limits Associated with Database Objects in NDB Cluster”.

  COLUMN_FORMAT currently has no effect on columns of tables using storage engines other than NDB. MySQL 8.0 silently ignores COLUMN_FORMAT.

- ENGINE_ATTRIBUTE and SECONDARY_ENGINE_ATTRIBUTE options (available as of MySQL 8.0.21) are used to specify column 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.

- STORAGE

  For NDB tables, it is possible to specify whether the column is stored on disk or in memory by using a STORAGE clause. STORAGE DISK causes the column to be stored on disk, and STORAGE MEMORY causes in-memory storage to be used. The CREATE TABLE statement used must still include a TABLESPACE clause:

```
mysql> CREATE TABLE t1 (
    ->     c1 INT STORAGE DISK,
    ->     c2 INT STORAGE MEMORY
    -> ) ENGINE NDB;
ERROR 1005 (HY000): Can't create table 'c.t1' (errno: 140)

mysql> CREATE TABLE t1 (
    ->     c1 INT STORAGE DISK,
    ->     c2 INT STORAGE MEMORY
    -> ) TABLESPACE ts_1 ENGINE NDB;
Query OK, 0 rows affected (1.06 sec)
```

  For NDB tables, STORAGE DEFAULT is equivalent to STORAGE MEMORY.

  The STORAGE clause has no effect on tables using storage engines other than NDB. The STORAGE keyword is supported only in the build of **mysqld** that is supplied with NDB Cluster; it is not recognized in any other version of MySQL, where any attempt to use the STORAGE keyword causes a syntax error.

- GENERATED ALWAYS

  Used to specify a generated column expression. For information about generated columns, see Section 15.1.20.8, “CREATE TABLE and Generated Columns”.

  Stored generated columns can be indexed. InnoDB supports secondary indexes on virtual generated columns. See Section 15.1.20.9, “Secondary Indexes and Generated Columns”.

Indexes, Foreign Keys, and CHECK Constraints

Several keywords apply to creation of indexes, foreign keys, and CHECK constraints. For general background in addition to the following descriptions, see Section 15.1.15, “CREATE INDEX Statement”, Section 15.1.20.5, “FOREIGN KEY Constraints”, and Section 15.1.20.6, “CHECK Constraints”.

- CONSTRAINT ***symbol***

  The CONSTRAINT ***symbol*** clause may be given to name a constraint. If the clause is not given, or a ***symbol*** is not included following the CONSTRAINT keyword, MySQL automatically generates a constraint name, with the exception noted below. The ***symbol*** value, if used, must be unique per

schema (database), per constraint type. A duplicate ***symbol*** results in an error. See also the discussion about length limits of generated constraint identifiers at Section 11.2.1, “Identifier Length Limits”.

> **Note**
>
> If the `CONSTRAINT` ***symbol*** clause is not given in a foreign key definition, or a ***symbol*** is not included following the `CONSTRAINT` keyword, MySQL uses the foreign key index name up to MySQL 8.0.15, and automatically generates a constraint name thereafter.

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. Consequently, names for each type of constraint must be unique per schema, but constraints of different types can have the same name.

- `PRIMARY KEY`

  A unique index where all key columns must be defined as `NOT NULL`. If they are not explicitly declared as `NOT NULL`, MySQL declares them so implicitly (and silently). A table can have only one `PRIMARY KEY`. The name of a `PRIMARY KEY` is always `PRIMARY`, which thus cannot be used as the name for any other kind of index.

  If you do not have a `PRIMARY KEY` and an application asks for the `PRIMARY KEY` in your tables, MySQL returns the first `UNIQUE` index that has no `NULL` columns as the `PRIMARY KEY`.

  In `InnoDB` tables, keep the `PRIMARY KEY` short to minimize storage overhead for secondary indexes. Each secondary index entry contains a copy of the primary key columns for the corresponding row. (See Section 17.6.2.1, “Clustered and Secondary Indexes”.)

  In the created table, a `PRIMARY KEY` is placed first, followed by all `UNIQUE` indexes, and then the nonunique indexes. This helps the MySQL optimizer to prioritize which index to use and also more quickly to detect duplicated `UNIQUE` keys.

  A `PRIMARY KEY` can be a multiple-column index. However, you cannot create a multiple-column index using the `PRIMARY KEY` key attribute in a column specification. Doing so only marks that single column as primary. You must use a separate `PRIMARY KEY(`***key_part, ...***`)` clause.

  If a table has a `PRIMARY KEY` or `UNIQUE NOT NULL` index that consists of a single column that has an integer type, you can use `_rowid` to refer to the indexed column in `SELECT` statements, as described in Unique Indexes.

In MySQL, the name of a PRIMARY KEY is PRIMARY. For other indexes, if you do not assign a name, the index is assigned the same name as the first indexed column, with an optional suffix (_2, _3, ...) to make it unique. You can see index names for a table using SHOW INDEX FROM ***tbl_name***. See Section 15.7.7.22, “SHOW INDEX Statement”.

- KEY | INDEX

  KEY is normally a synonym for INDEX. The key attribute PRIMARY KEY can also be specified as just KEY when given in a column definition. This was implemented for compatibility with other database systems.

- UNIQUE

  A UNIQUE index creates a constraint such that all values in the index must be distinct. An error occurs if you try to add a new row with a key value that matches an existing row. For all engines, a UNIQUE index permits multiple NULL values for columns that can contain NULL. If you specify a prefix value for a column in a UNIQUE index, the column values must be unique within the prefix length.

  If a table has a PRIMARY KEY or UNIQUE NOT NULL index that consists of a single column that has an integer type, you can use _rowid to refer to the indexed column in SELECT statements, as described in Unique Indexes.

- FULLTEXT

  A FULLTEXT index is a special type of index used for full-text searches. Only the InnoDB and MyISAM storage engines support FULLTEXT indexes. They can be created only from CHAR, VARCHAR, and TEXT columns. Indexing always happens over the entire column; column prefix indexing is not supported and any prefix length is ignored if specified. See Section 14.9, “Full-Text Search Functions”, for details of operation. A WITH PARSER clause can be specified as an ***index_option*** value to associate a parser plugin with the index if full-text indexing and searching operations need special handling. This clause is valid only for FULLTEXT indexes. InnoDB and MyISAM support full-text parser plugins. See Full-Text Parser Plugins and Writing Full-Text Parser Plugins for more information.

- SPATIAL

  You can create SPATIAL indexes on spatial data types. Spatial types are supported only for InnoDB and MyISAM tables, and indexed columns must be declared as NOT NULL. See Section 13.4, “Spatial Data Types”.

- FOREIGN KEY

MySQL supports foreign keys, which let you cross-reference related data across tables, and foreign key constraints, which help keep this spread-out data consistent. For definition and option information, see ***reference_definition***, and ***reference_option***.

Partitioned tables employing the InnoDB storage engine do not support foreign keys. See Section 26.6, “Restrictions and Limitations on Partitioning”, for more information.

- CHECK

  The CHECK clause enables the creation of constraints to be checked for data values in table rows. See Section 15.1.20.6, “CHECK Constraints”.

- ***key_part***

  - A ***key_part*** specification can end with ASC or DESC to specify whether index values are stored in ascending or descending order. The default is ascending if no order specifier is given.

  - Prefixes, defined by the ***length*** attribute, can be up to 767 bytes long for InnoDB tables that use the REDUNDANT or COMPACT row format. The prefix length limit is 3072 bytes for InnoDB tables that use the DYNAMIC or COMPRESSED row format. For MyISAM tables, the prefix length limit is 1000 bytes.

    Prefix *limits* are measured in bytes. However, prefix *lengths* for index specifications in CREATE TABLE, ALTER TABLE, and CREATE INDEX statements are interpreted as number of characters for nonbinary string types (CHAR, VARCHAR, TEXT) and number of bytes for binary string types (BINARY, VARBINARY, BLOB). Take this into account when specifying a prefix length for a nonbinary string column that uses a multibyte character set.

  - Beginning with MySQL 8.0.17, the ***expr*** for a ***key_part*** specification can take the form (CAST ***json_path*** AS ***type*** ARRAY) to create a multi-valued index on a JSON column. Multi-Valued Indexes, provides detailed information regarding creation of, usage of, and restrictions and limitations on multi-valued indexes.

- ***index_type***

  Some storage engines permit you to specify an index type when creating an index. The syntax for the ***index_type*** specifier is USING ***type_name***.

  Example:

```
CREATE TABLE lookup
  (id INT, INDEX USING BTREE (id))
  ENGINE = MEMORY;
```

The preferred position for USING is after the index column list. It can be given before the column list, but support for use of the option in that position is deprecated and you should expect it to be removed in a future MySQL release.

- ***index_option***

  index_option values specify additional options for an index.

  - 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 information about the table-level KEY_BLOCK_SIZE attribute, see Table Options.

  - WITH PARSER

    The WITH PARSER option can be used only with FULLTEXT indexes. It associates a parser plugin with the index if full-text indexing and searching operations need special handling. InnoDB and MyISAM support full-text parser plugins. If you have a MyISAM table with an associated full-text parser plugin, you can convert the table to InnoDB using ALTER TABLE.

  - COMMENT

    Index definitions can include an optional comment of up to 1024 characters.

    You can set the InnoDB MERGE_THRESHOLD value for an individual index using the ***index_option*** COMMENT clause. See Section 17.8.11, “Configuring the Merge Threshold for Index Pages”.

  - VISIBLE, INVISIBLE

    Specify index visibility. Indexes are visible by default. An invisible index is not used by the optimizer. Specification of index visibility applies to indexes other than primary keys (either explicit or implicit). For more information, see Section 10.3.12, “Invisible Indexes”.

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

  For more information about permissible ***index_option*** values, see Section 15.1.15, “CREATE INDEX Statement”. For more information about indexes, see Section 10.3.1, “How MySQL Uses Indexes”.

- ***reference_definition***

  For ***reference_definition*** syntax details and examples, see Section 15.1.20.5, “FOREIGN KEY Constraints”.

  InnoDB and NDB tables support checking of foreign key constraints. The columns of the referenced table must always be explicitly named. Both ON DELETE and ON UPDATE actions on foreign keys are supported. For more detailed information and examples, see Section 15.1.20.5, “FOREIGN KEY Constraints”.

  For other storage engines, MySQL Server parses and ignores the FOREIGN KEY syntax in CREATE TABLE statements.

  > **Important**
  >
  > For users familiar with the ANSI/ISO SQL Standard, please note that no storage engine, including InnoDB, recognizes or enforces the MATCH clause used in referential integrity constraint definitions. Use of an explicit MATCH clause does not have the specified effect, and also causes ON DELETE and 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 15.1.20.5, “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 17, *The InnoDB Storage Engine*, and in particular Section 17.1, “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 18.2, “The MyISAM Storage Engine”.
`MEMORY`	The data for this storage engine is stored only in memory. See Section 18.3, “The MEMORY Storage Engine”.
`CSV`	Tables that store rows in comma-separated values format. See Section 18.4, “The CSV Storage Engine”.
`ARCHIVE`	The archiving storage engine. See Section 18.5, “The ARCHIVE Storage Engine”.
`EXAMPLE`	An example engine. See Section 18.9, “The EXAMPLE Storage Engine”.
`FEDERATED`	Storage engine that accesses remote tables. See Section 18.8, “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 18.7, “The MERGE Storage Engine”.

Storage Engine	Description
NDB	Clustered, fault-tolerant, memory-based tables, supporting transactions and foreign keys. Also known as NDBCLUSTER. See Chapter 25, *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 7.1.11, “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 17.6.3.9, “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 7.1.8, “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 17.8.11, “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 15.1.20.12, “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 17.9.2, “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 17.6.1.2, “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 10.12.2.2, “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 7.1.8, “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 17.13, “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 18.7, “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 17.9.1, “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 17.9, “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 17.10, “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 18.2.3, “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 15.1.1, “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 17.8.10.1, “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 17.8.10.1, “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 17.8.10.1, “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 17.6.3.3, “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 25.6.11, “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 18.7, “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 26, *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 15.1.9, “ALTER TABLE Statement”. For more detailed descriptions and examples, see Section 26.3, “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 26.2.4, “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 26.2.4.1, “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 26.2.4.1, “LINEAR HASH Partitioning”, and Section 26.2.5, “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 15.1.9.1, “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 26.2.1, “RANGE Partitioning”, and Section 26.4, “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 26, *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 26.2.6, “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 26.6.3, “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.