

13.2.5. CREATE TABLE Syntax

13.2.5.1. Silent Column Specification Changes

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

Or:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
    [( ) LIKE old_tbl_name ( )];
```

create_definition:

```
    column_definition
    | [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
    | KEY [index_name] [index_type] (index_col_name,...)
    | INDEX [index_name] [index_type] (index_col_name,...)
    | [CONSTRAINT [symbol]] UNIQUE [INDEX]
      [index_name] [index_type] (index_col_name,...)
    | [FULLTEXT|SPATIAL] [INDEX] [index_name] (index_col_name,...)
    | [CONSTRAINT [symbol]] FOREIGN KEY
      [index_name] (index_col_name,...) [reference_definition]
    | CHECK (expr)
```

column_definition:

```
    col_name type [NOT NULL | NULL] [DEFAULT default_value]
      [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
      [COMMENT 'string'] [reference_definition]
```

type:

```
    TINYINT[(length)] [UNSIGNED] [ZEROFILL]
    | SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
    | MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
    | INT[(length)] [UNSIGNED] [ZEROFILL]
    | INTEGER[(length)] [UNSIGNED] [ZEROFILL]
    | BIGINT[(length)] [UNSIGNED] [ZEROFILL]
    | REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
    | DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
    | FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
    | DECIMAL(length,decimals) [UNSIGNED] [ZEROFILL]
    | NUMERIC(length,decimals) [UNSIGNED] [ZEROFILL]
    | DATE
    | TIME
    | TIMESTAMP
    | DATETIME
    | CHAR(length) [BINARY | ASCII | UNICODE]
    | VARCHAR(length) [BINARY]
    | TINYBLOB
    | BLOB
    | MEDIUMBLOB
    | LONGBLOB
    | TINYTEXT [BINARY]
    | TEXT [BINARY]
    | MEDIUMTEXT [BINARY]
    | LONGTEXT [BINARY]
```

```

| ENUM(value1,value2,value3,...)
| SET(value1,value2,value3,...)
| spatial_type

index_col_name:
    col_name [(length)] [ASC | DESC]

reference_definition:
    REFERENCES tbl_name [(index_col_name,...)]
                [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:
    {ENGINE|TYPE} = engine_name
    AUTO_INCREMENT = value
    AVG_ROW_LENGTH = value
    CHECKSUM = {0 | 1}
    COMMENT = 'string'
    MAX_ROWS = value
    MIN_ROWS = value
    PACK_KEYS = {0 | 1 | DEFAULT}
    PASSWORD = 'string'
    DELAY_KEY_WRITE = {0 | 1}
    ROW_FORMAT = {DEFAULT|DYNAMIC|FIXED|COMPRESSED|REDUNDANT|COMPACT}
    RAID_TYPE = { 1 | STRIPED | RAID0 }
                RAID_CHUNKS = value
                RAID_CHUNKSIZE = value
    UNION = (tbl_name[,tbl_name]...)
    INSERT_METHOD = { NO | FIRST | LAST }
    DATA DIRECTORY = 'absolute path to directory'
    INDEX DIRECTORY = 'absolute path to directory'
    [DEFAULT] CHARACTER SET charset_name [COLLATE collation_name]

select_statement:
    [IGNORE | REPLACE] [AS] SELECT ... (Some legal select statement)

```

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

Rules for allowable table names are given in [Section 9.2, “Database, Table, Index, Column, and Alias Names”](#). By default, the table is created in the current database. An error occurs if the table exists, if there is no current database, or if the database does not exist.

In MySQL 3.22 or later, the table name can be specified as *db_name.tbl_name* to create the table in a specific database. This works whether or not there is a current database. If you use quoted identifiers, quote the database and table names separately. For example, ``mydb`.`mytbl`` is legal, but ``mydb.mytbl`` is not.

From MySQL 3.23 on, you can use the TEMPORARY keyword when creating a table. A TEMPORARY table is visible only to the current connection, and is dropped automatically

when the connection is closed. This means that two different connections 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.) From MySQL 4.0.2 on, you must have the `CREATE TEMPORARY TABLES` privilege to be able to create temporary tables.

In MySQL 3.23 or later, you can use the keywords `IF NOT EXISTS` so that an error does not occur if the table exists. Note that there is no verification that the existing table has a structure identical to that indicated by the `CREATE TABLE` statement. Also, if you use `IF NOT EXISTS` in a `CREATE TABLE ... SELECT` statement, any records selected by the `SELECT` part are inserted regardless of whether the table already exists.

MySQL represents each table by an `.frm` table format (definition) file in the database directory. The storage engine for the table might create other files as well. In the case of `MyISAM` tables, the storage engine creates data and index files. Thus, for each `MyISAM` table `tbl_name`, there are three disk files:

| File | Purpose |
|---------------------------|--------------------------------|
| <code>tbl_name.frm</code> | Table format (definition) file |
| <code>tbl_name.MYD</code> | Data file |
| <code>tbl_name.MYI</code> | Index file |

The files created by each storage engine to represent tables are described in [Chapter 14, MySQL Storage Engines and Table Types](#).

For general information on the properties of the various column types, see [Chapter 11, Column Types](#). For information about spatial column types, see [Chapter 18, Spatial Extensions in MySQL](#).

- If neither `NULL` nor `NOT NULL` is specified, the column is treated as though `NULL` had been specified.
- An integer 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`. See [Section 24.2.3.33, "mysql_insert_id\(\)](#)".

As of MySQL 4.1.1, specifying the `NO_AUTO_VALUE_ON_ZERO` flag for the `--sql-mode` server option or the `sql_mode` system variable allows you to store `0` in `AUTO_INCREMENT` columns as `0` without generating a new sequence value. See [Section 5.3.1, "mysqld Command-Line Options"](#).

Note: There can be only one `AUTO_INCREMENT` column per table, it must be indexed, and it cannot have a `DEFAULT` value. As of MySQL 3.23, 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 don't accidentally get an `AUTO_INCREMENT` column that contains `0`.

For MyISAM and BDB tables, you can specify an `AUTO_INCREMENT` secondary column in a multiple-column key. See [Section 3.6.9, “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
```

- As of MySQL 4.1, character column definitions can include a `CHARACTER SET` attribute to specify the character set and, optionally, a collation for the column. For details, see [Chapter 10, Character Set Support](#). `CHARSET` is a synonym for `CHARACTER SET`.

```
CREATE TABLE t (c CHAR(20) CHARACTER SET utf8 COLLATE utf8_bin);
```

Also as of 4.1, MySQL interprets length specifications in character column definitions in characters. (Earlier versions interpret them in bytes.)

- `NULL` values are handled differently for `TIMESTAMP` columns than for other column types. Before MySQL 4.1.6, you cannot store a literal `NULL` in a `TIMESTAMP` column; setting the column to `NULL` sets it to the current date and time. Because `TIMESTAMP` columns behave this way, the `NULL` and `NOT NULL` attributes do not apply in the normal way and are ignored if you specify them. On the other hand, to make it easier for MySQL clients to use `TIMESTAMP` columns, the server reports that such columns can be assigned `NULL` values (which is true), even though `TIMESTAMP` never actually contains a `NULL` value. You can see this when you use `DESCRIBE tbl_name` to get a description of your table.

Note that setting a `TIMESTAMP` column to 0 is not the same as setting it to `NULL`, because 0 is a valid `TIMESTAMP` value.

- The `DEFAULT` clause specifies a default value for a column. With one exception, the default value must be a constant; it cannot be a function or an expression. This means, for example, that you cannot set the default for a date column to be the value of a function such as `NOW()` or `CURRENT_DATE`. The exception is that you can specify `CURRENT_TIMESTAMP` as the default for a `TIMESTAMP` column as of MySQL 4.1.2. See [Section 11.3.1.2, “TIMESTAMP Properties as of MySQL 4.1”](#).

Prior to MySQL 5.0.2, if a column definition includes no explicit `DEFAULT` value, MySQL determines the default value as follows:

If the column can take `NULL` as a value, the column is defined with an explicit `DEFAULT NULL` clause.

If the column cannot take `NULL` as the value, MySQL defines the column with an explicit `DEFAULT` clause, using the implicit default value for the column data type. Implicit defaults are defined as follows:

- For numeric types other than those declared with the `AUTO_INCREMENT` attribute, the default is 0. For an `AUTO_INCREMENT` column, the default value is the next value in the sequence.
- For date and time types other than `TIMESTAMP`, the default is the appropriate

``zero" value for the type. For the first `TIMESTAMP` column in a table, the default value is the current date and time. See [Section 11.3, "Date and Time Types"](#).

- For string types other than `ENUM`, the default value is the empty string. For `ENUM`, the default is the first enumeration value.

`BLOB` and `TEXT` columns cannot be assigned a default value.

As of MySQL 5.0.2, if a column definition includes no explicit `DEFAULT` value, MySQL determines the default value as follows:

If the column can take `NULL` as a value, the column is defined with an explicit `DEFAULT NULL` clause. This is the same as before 5.0.2.

If the column cannot take `NULL` as the value, MySQL defines the column with no explicit `DEFAULT` clause. For data entry, if an `INSERT` or `REPLACE` statement includes no value for the column, MySQL handles the column according to the SQL mode in effect at the time:

- If strict mode is not enabled, MySQL sets the column to the implicit default value for the column data type.
- If strict mode is enabled, an error occurs for transactional tables and the statement is rolled back. For non-transactional tables, an error occurs, but if this happens for the the second or subsequent row of a multiple-row statement, the preceding rows will have been inserted.

Suppose a table `t` is defined as follows:

```
CREATE TABLE t (i INT NOT NULL);
```

In this case, `i` has no explicit default, so in strict mode all of the following statements produce an error in strict mode and no row is inserted. For non strict mode, only the third statement produces an error; the implicit default is inserted for the first two, but the third fails because `DEFAULT(i)` cannot produce a value:

```
INSERT INTO t VALUES();  
INSERT INTO t VALUES(DEFAULT);  
INSERT INTO t VALUES(DEFAULT(i));
```

See [Section 5.3.2, "The Server SQL Mode"](#).

For a given table, you can use the `SHOW CREATE TABLE` statement to see which columns have an explicit `DEFAULT` clause.

- A comment for a column can be specified with the `COMMENT` option. The comment is displayed by the `SHOW CREATE TABLE` and `SHOW FULL COLUMNS` statements. This option is operational as of MySQL 4.1. (It is allowed but ignored in earlier versions.)
- From MySQL 4.1.0 on, the attribute `SERIAL` can be used as an alias for `BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE`.

- `KEY` is normally a synonym for `INDEX`. From MySQL 4.1, 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.
- In MySQL, a `UNIQUE` index is one in which all values in the index must be distinct. An error occurs if you try to add a new row with a key that matches an existing row. The exception to this is that if a column in the index is allowed to contain `NULL` values, it can contain multiple `NULL` values. This exception does not apply to `BDB` tables, for which an indexed column allows only a single `NULL`.
- A `PRIMARY KEY` is a unique `KEY` 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`. If you don't 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 the created table, a `PRIMARY KEY` is placed first, followed by all `UNIQUE` indexes, and then the non-unique 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(index_col_name, ...)` clause.
- If a `PRIMARY KEY` or `UNIQUE` index consists of only one column that has an integer type, you can also refer to the column as `_rowid` in `SELECT` statements (new in MySQL 3.23.11).
- In MySQL, the name of a `PRIMARY KEY` is `PRIMARY`. For other indexes, if you don't 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 13.5.4.10, "SHOW INDEX Syntax"](#).
- From MySQL 4.1.0 on, some storage engines allow 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;
```

For details about `USING`, see [Section 13.2.4, "CREATE INDEX Syntax"](#).

For more information about how MySQL uses indexes, see [Section 7.4.5, "How MySQL Uses Indexes"](#).

- Only the `MyISAM`, `InnoDB`, `BDB`, and (as of MySQL 4.0.2) `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.

- With `col_name(length)` syntax in an index specification, you can create an index that uses only the first `length` characters of a CHAR or VARCHAR column. Indexing only a prefix of column values like this can make the index file much smaller. See [Section 7.4.3, “Column Indexes”](#).

The MyISAM and (as of MySQL 4.0.14) InnoDB storage engines also support indexing on BLOB and TEXT columns. When indexing a BLOB or TEXT column, you *must* specify a prefix length for the index. For example:

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

Prefixes can be up to 255 bytes long (or 1000 bytes for MyISAM and InnoDB tables as of MySQL 4.1.2). Note that prefix limits are measured in bytes, whereas the prefix length in CREATE TABLE statements is interpreted as number of characters. Take this into account when specifying a prefix length for a column that uses a multi-byte character set.

- An `index_col_name` specification can end with ASC or DESC. These keywords are allowed for future extensions for specifying ascending or descending index value storage. Currently they are parsed but ignored; index values are always stored in ascending order.
- When you use ORDER BY or GROUP BY with a TEXT or BLOB column, the server sorts values using only the initial number of bytes indicated by the `max_sort_length` system variable. See [Section 11.4.3, “The BLOB and TEXT Types”](#).
- In MySQL 3.23.23 or later, you can create special FULLTEXT indexes. They are used for full-text search. Only the MyISAM table type supports FULLTEXT indexes. They can be created only from CHAR, VARCHAR, and TEXT columns. Indexing always happens over the entire column; partial indexing is not supported and any prefix length is ignored if specified. See [Section 12.6, “Full-Text Search Functions”](#) for details of operation.
- In MySQL 4.1 or later, you can create SPATIAL indexes on spatial column types. Spatial types are supported only for MyISAM tables and indexed columns must be declared as NOT NULL. See [Chapter 18, Spatial Extensions in MySQL](#).
- In MySQL 3.23.44 or later, InnoDB tables support checking of foreign key constraints. See [Chapter 15, The InnoDB Storage Engine](#). Note that the FOREIGN KEY syntax in InnoDB is more restrictive than the syntax presented for the CREATE TABLE statement at the beginning of this section: The columns of the referenced table must always be explicitly named. InnoDB supports both ON DELETE and ON UPDATE actions on foreign keys as of MySQL 3.23.50 and 4.0.8, respectively. For the precise syntax, see [Section 15.7.4, “FOREIGN KEY Constraints”](#).

For other storage engines, MySQL Server parses the FOREIGN KEY and REFERENCES syntax in CREATE TABLE statements, but without further action being taken. The CHECK clause is parsed but ignored by all storage engines. See [Section 1.7.5.5, “Foreign Keys”](#).

- For MyISAM and ISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte. The maximum record length in bytes can be calculated as follows:

```

row length = 1
             + (sum of column lengths)
             + (number of NULL columns + delete_flag + 7)/8
             + (number of variable-length columns)

```

delete_flag is 1 for tables with static record format. Static tables use a bit in the row record for a flag that indicates whether the row has been deleted. *delete_flag* is 0 for dynamic tables because the flag is stored in the dynamic row header.

These calculations do not apply for InnoDB tables, for which storage size is no different for NULL columns than for NOT NULL columns.

The *table_options* part of the CREATE TABLE syntax can be used in MySQL 3.23 and above.

The ENGINE and TYPE options specify the storage engine for the table. ENGINE was added in MySQL 4.0.18 (for 4.0) and 4.1.2 (for 4.1). It is the preferred option name as of those versions, and TYPE has become deprecated. TYPE is supported throughout the 4.x series, but likely will be removed in MySQL 5.1.

The ENGINE and TYPE options take the following values:

| Storage Engine | Description |
|----------------|--|
| ARCHIVE | The archiving storage engine. See Section 14.7, “The ARCHIVE Storage Engine” . |
| BDB | Transaction-safe tables with page locking. See Section 14.4, “The BDB (BerkeleyDB) Storage Engine” . |
| BerkeleyDB | An alias for BDB. |
| CSV | Tables that store rows in comma-separated values format. See Section 14.8, “The csv Storage Engine” . |
| EXAMPLE | An example engine. See Section 14.5, “The EXAMPLE Storage Engine” . |
| FEDERATED | Storage engine that accesses remote tables. See Section 14.6, “The FEDERATED Storage Engine” . |
| HEAP | The data for this table is stored only in memory. See Section 14.3, “The MEMORY (HEAP) Storage Engine” . |
| ISAM | The original MySQL storage engine. See Section 14.9, “The ISAM Storage Engine” . |
| InnoDB | Transaction-safe tables with row locking and foreign keys. See Chapter 15, The InnoDB Storage Engine . |
| MEMORY | An alias for HEAP. (Actually, as of MySQL 4.1, MEMORY is the preferred term.) |
| MERGE | A collection of MyISAM tables used as one table. See Section 14.2, “The MERGE Storage Engine” . |
| | |

| | |
|------------|---|
| MRG_MyISAM | An alias for MERGE. |
| MyISAM | The binary portable storage engine that is the improved replacement for ISAM. See Section 14.1, “The MyISAM Storage Engine” . |
| NDB | Alias for NDBCLUSTER. |
| NDBCLUSTER | Clustered, fault-tolerant, memory-based tables. See Chapter 16, MySQL Cluster . |

See [Chapter 14, MySQL Storage Engines and Table Types](#).

If a storage engine is specified that is not available, MySQL uses MyISAM instead. For example, if a table definition includes the ENGINE=BDB option but the MySQL server does not support BDB tables, the table is created as a MyISAM table. This makes it possible to have a replication setup where you have transactional tables on the master but tables created on the slave are non-transactional (to get more speed). In MySQL 4.1.1, a warning occurs if the storage engine specification is not honored.

The other table options are used to optimize the behavior of the table. In most cases, you don't have to specify any of them. The options work for all storage engines unless otherwise indicated:

- AUTO_INCREMENT

The initial AUTO_INCREMENT value for the table. This works for MyISAM only, for MEMORY as of MySQL 4.1, and for InnoDB as of MySQL 5.0.3. 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.

- 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 records.

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 a table is 4GB (or 2GB if your operating system only supports 2GB tables). The reason for this is just to keep down the pointer sizes to make the index smaller and faster if you don't really need big files. If you want all your tables to be able to grow above the 4GB limit and are willing to have your smaller tables slightly slower and larger than necessary, you may increase the default pointer size by setting the `myisam_data_pointer_size` system variable, which was added in MySQL 4.1.2. See [Section 5.3.3, “Server System Variables”](#).

- 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.)

- `COMMENT`

A comment for your table, up to 60 characters long.

- `MAX_ROWS`

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

- `MIN_ROWS`

The minimum number of rows you plan to store in the table.

- `PACK_KEYS`

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` (MySQL 4.0) tells the storage engine to only pack long `CHAR/VARCHAR` columns. (`MyISAM` and `ISAM` only.)

If you don't use `PACK_KEYS`, the default is to only pack strings, 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 big 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 isn't 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`

Encrypt the `.frm` file with a password. This option doesn't do anything in the standard MySQL version.

- `DELAY_KEY_WRITE`

Set this to 1 if you want to delay key updates for the table until the table is closed.

(MyISAM only.)

- **ROW_FORMAT**

Defines how the rows should be stored. Currently this option works only with MyISAM tables. The option value can `FIXED` or `DYNAMIC` for static or variable-length row format. **mysampack** sets the type to `COMPRESSED`. See [Section 14.1.3, "MyISAM Table Storage Formats"](#).

Starting with MySQL/InnoDB-5.0.3, InnoDB records are stored in a more compact format (`ROW_FORMAT=COMPACT`) by default. The old format can be requested by specifying `ROW_FORMAT=REDUNDANT`.

- **RAID_TYPE**

Note: This information applies only before MySQL 5.0. RAID support has been removed as of MySQL 5.0.

The `RAID_TYPE` option can help you to exceed the 2GB/4GB limit for the MyISAM data file (not the index file) on operating systems that don't support big files. This option is unnecessary and not recommended for filesystems that support big files.

You can get more speed from the I/O bottleneck by putting RAID directories on different physical disks. The only allowed `RAID_TYPE` is `STRIPED`. `1` and `RAID0` are aliases for `STRIPED`.

If you specify the `RAID_TYPE` option for a MyISAM table, specify the `RAID_CHUNKS` and `RAID_CHUNKSIZE` options as well. The maximum `RAID_CHUNKS` value is 255. MyISAM creates `RAID_CHUNKS` subdirectories named `00`, `01`, `02`, ... `09`, `0a`, `0b`, ... in the database directory. In each of these directories, MyISAM creates a file `tbl_name.MYD`. When writing data to the data file, the RAID handler maps the first `RAID_CHUNKSIZE*1024` bytes to the first file, the next `RAID_CHUNKSIZE*1024` bytes to the next file, and so on.

`RAID_TYPE` works on any operating system, as long as you have built MySQL with the `--with-raid` option to **configure**. To determine whether a server supports RAID tables, use `SHOW VARIABLES LIKE 'have_raid'` to see whether the variable value is `YES`.

- **UNION**

`UNION` is used when you want to use a collection of identical tables as one. This works only with `MERGE` tables. See [Section 14.2, "The MERGE Storage Engine"](#).

For the moment, you must have `SELECT`, `UPDATE`, and `DELETE` privileges for the tables you map to a `MERGE` table. Originally, all used tables had to be in the same database as the `MERGE` table itself. This restriction has been lifted as of MySQL 4.1.1.

- **INSERT_METHOD**

If you want to insert data in a `MERGE` table, you have to specify with `INSERT_METHOD` into which table the row should be inserted. `INSERT_METHOD` is an option useful for

MERGE tables only. This option was introduced in MySQL 4.0.0. See [Section 14.2, “The MERGE Storage Engine”](#).

- DATA DIRECTORY , INDEX DIRECTORY

By using DATA DIRECTORY='directory' or INDEX DIRECTORY='directory' you can specify where the MyISAM storage engine should put a table's data file and index file. Note that the directory should be a full path to the directory (not a relative path).

These options work only for MyISAM tables from MySQL 4.0 on, when you are not using the --skip-symbolic-links option. Your operating system must also have a working, thread-safe realpath() call. See [Section 7.6.1.2, “Using Symbolic Links for Tables on Unix”](#).

As of MySQL 3.23, you can create one table from another by adding a SELECT statement at the end of the CREATE TABLE statement:

```
CREATE TABLE new_tbl 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))
-> TYPE=MyISAM SELECT b,c FROM test2;
```

This creates a MyISAM table with three columns, a, b, and c. 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.

If any errors occur while copying the data to the table, it is automatically dropped and not created.

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;
```

Some conversion of column types might occur. For example, the AUTO_INCREMENT attribute is not preserved, and VARCHAR columns can become CHAR columns.

When creating a table with CREATE ... 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;
```

As of MySQL 4.1, you can explicitly specify the type for a generated column:

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

In MySQL 4.1, you can also use LIKE to create an empty table based on the definition of another table, including any column attributes and indexes the original table has:

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

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

You can precede the SELECT by IGNORE or REPLACE to indicate how to handle records that duplicate unique key values. With IGNORE, new records that duplicate an existing record on a unique key value are discarded. With REPLACE, new records replace records that have the same unique key value. If neither IGNORE nor REPLACE is specified, duplicate unique key values result in an error.

To ensure that the update log/binary log can be used to re-create the original tables, MySQL does not allow concurrent inserts during CREATE TABLE ... SELECT.