

13.1.4. INSERT Syntax

[13.1.4.1. INSERT ... SELECT Syntax](#)

[13.1.4.2. INSERT DELAYED Syntax](#)

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name [(col_name,...)]
  VALUES ({expr | DEFAULT},...),(...),...
  [ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Or:

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name
  SET col_name={expr | DEFAULT}, ...
  [ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

Or:

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
  [INTO] tbl_name [(col_name,...)]
  SELECT ...
  [ ON DUPLICATE KEY UPDATE col_name=expr, ... ]
```

INSERT inserts new rows into an existing table. The INSERT ... VALUES and INSERT ... SET forms of the statement insert rows based on explicitly specified values. The INSERT ... SELECT form inserts rows selected from another table or tables. The INSERT ... VALUES form with multiple value lists is supported in MySQL 3.22.5 or later. The INSERT ... SET syntax is supported in MySQL 3.22.10 or later. INSERT ... SELECT is discussed further in See [Section 13.1.4.1, "INSERT ... SELECT Syntax"](#).

tbl_name is the table into which rows should be inserted. The columns for which the statement provides values can be specified as follows:

- The column name list or the SET clause indicates the columns explicitly.
- If you do not specify the column list for INSERT ... VALUES or INSERT ... SELECT, values for every column in the table must be provided in the VALUES() list or by the SELECT. If you don't know the order of the columns in the table, use DESCRIBE *tbl_name* to find out.

Column values can be given in several ways:

- If you are not running in strict mode, any column not explicitly given a value is set to its default (explicit or implicit) value. For example, if you specify a column list that doesn't name all the columns in the table, unnamed columns are set to their default values. Default value assignment is described in [Section 13.2.5, "CREATE TABLE Syntax"](#). See [Section 1.7.6.2, "Constraints on Invalid Data"](#).

If you want INSERT statements to generate an error unless you explicitly specify values for all columns that don't have a default value, you should use STRICT mode. See [Section 5.3.2, "The Server SQL Mode"](#).

- You can use the keyword `DEFAULT` to explicitly set a column to its default value. (New in MySQL 4.0.3.) This makes it easier to write `INSERT` statements that assign values to all but a few columns, because it allows you to avoid writing an incomplete `VALUES` list that does not include a value for each column in the table. Otherwise, you would have to write out the list of column names corresponding to each value in the `VALUES` list.

As of MySQL 4.1.0, you can use `DEFAULT(col_name)` as a more general form that can be used in expressions to produce a column's default value.

- If both the column list and the `VALUES` list are empty, `INSERT` creates a row with each column set to its default value:

```
mysql> INSERT INTO tbl_name () VALUES();
```

In `STRICT` mode you will get an error if a column doesn't have a default value. In not `strict` mode, MySQL will use the implicit default value for any column with a not defined default value.

- You can specify an expression `expr` to provide a column value. This might involve type conversion if the type of the expression does not match the type of the column, and conversion of a given value can result in different inserted values depending on the column type. For example, inserting the string `'1999.0e-2'` into an `INT`, `FLOAT`, `DECIMAL(10,6)`, or `YEAR` column results in the values 1999, 19.9921, 19.992100, and 1999. The reason the value stored in the `INT` and `YEAR` columns is 1999 is that the string-to-integer conversion looks only at as much of the initial part of the string as may be considered a valid integer or year. For the floating-point and fixed-point columns, the string-to-floating-point conversion considers the entire string as a valid floating-point value.

An expression `expr` can refer to any column that was set earlier in a value list. For example, you can do this because the value for `col2` refers to `col1`, which has previously been assigned:

```
mysql> INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);
```

But you cannot do this because the value for `col1` refers to `col2`, which is assigned after `col1`:

```
mysql> INSERT INTO tbl_name (col1,col2) VALUES(col2*2,15);
```

One exception involves columns that contain `AUTO_INCREMENT` values. Because the `AUTO_INCREMENT` value is generated after other value assignments, any reference to an `AUTO_INCREMENT` column in the assignment returns a 0.

The `INSERT` statement supports the following modifiers:

- If you specify the `DELAYED` keyword, the server puts the row or rows to be inserted into a buffer, and the client issuing the `INSERT DELAYED` statement then can continue on. If the table is busy, the server holds the rows. When the table becomes free, it begins inserting rows, checking periodically to see whether there are new read requests for the table. If there are, the delayed row queue is suspended until the table becomes free again. See [Section 13.1.4.2, "INSERT DELAYED Syntax"](#). `DELAYED`

was added in MySQL 3.22.5.

- If you specify the `LOW_PRIORITY` keyword, execution of the `INSERT` is delayed until no other clients are reading from the table. This includes other clients that began reading while existing clients are reading, and while the `INSERT LOW_PRIORITY` statement is waiting. It is possible, therefore, for a client that issues an `INSERT LOW_PRIORITY` statement to wait for a very long time (or even forever) in a read-heavy environment. (This is in contrast to `INSERT DELAYED`, which lets the client continue at once.) See [Section 13.1.4.2, “INSERT DELAYED Syntax”](#). Note that `LOW_PRIORITY` should normally not be used with `MyISAM` tables because doing so disables concurrent inserts. See [Section 14.1, “The MyISAM Storage Engine”](#). `LOW_PRIORITY` was added in MySQL 3.22.5.
- If you specify the `HIGH_PRIORITY` keyword, it overrides the effect of the `--low-priority-updates` option if the server was started with that option. It also causes concurrent inserts not to be used. `HIGH_PRIORITY` was added in MySQL 3.23.11.
- The rows-affected value for an `INSERT` can be obtained using the `mysql_affected_rows()` C API function. See [Section 24.2.3.1, “mysql_affected_rows\(\)”](#).
- If you specify the `IGNORE` keyword in an `INSERT` statement, errors that occur while executing the statement are treated as warnings instead. For example, without `IGNORE`, a row that duplicates an existing `UNIQUE` index or `PRIMARY KEY` value in the table causes a duplicate-key error and the statement is aborted. With `IGNORE`, the error is ignored and the row is not inserted. Data conversions that would trigger errors abort the statement if `IGNORE` is not specified. With `IGNORE`, invalid values are adjusted to the closest value values and inserted; warnings are produced but the statement does not abort. You can determine with the `mysql_info()` C API function how many rows were inserted into the table.

If you specify the `ON DUPLICATE KEY UPDATE` clause (new in MySQL 4.1.0), and a row is inserted that would cause a duplicate value in a `UNIQUE` index or `PRIMARY KEY`, an `UPDATE` of the old row is performed. For example, if column `a` is declared as `UNIQUE` and contains the value `1`, the following two statements have identical effect:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
      -> ON DUPLICATE KEY UPDATE c=c+1;

mysql> UPDATE table SET c=c+1 WHERE a=1;
```

The rows-affected value is 1 if the row is inserted as a new record and 2 if an existing record is updated.

Note: If column `b` is unique too, the `INSERT` would be equivalent to this `UPDATE` statement instead:

```
mysql> UPDATE table SET c=c+1 WHERE a=1 OR b=2 LIMIT 1;
```

If `a=1 OR b=2` matches several rows, only *one* row is updated! In general, you should try to avoid using the `ON DUPLICATE KEY` clause on tables with multiple `UNIQUE` keys.

As of MySQL 4.1.1, you can use the `VALUES(col_name)` function in the `UPDATE` clause to

refer to column values from the `INSERT` part of the `INSERT ... UPDATE` statement. In other words, `VALUES(col_name)` in the `UPDATE` clause refers to the value of `col_name` that would be inserted if no duplicate-key conflict occurred. This function is especially useful in multiple-row inserts. The `VALUES()` function is meaningful only in `INSERT ... UPDATE` statements and returns `NULL` otherwise.

Example:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3),(4,5,6)
      -> ON DUPLICATE KEY UPDATE c=VALUES(a)+VALUES(b);
```

That statement is identical to the following two statements:

```
mysql> INSERT INTO table (a,b,c) VALUES (1,2,3)
      -> ON DUPLICATE KEY UPDATE c=3;
mysql> INSERT INTO table (a,b,c) VALUES (4,5,6)
      -> ON DUPLICATE KEY UPDATE c=9;
```

When you use `ON DUPLICATE KEY UPDATE`, the `DELAYED` option is ignored.

You can find the value used for an `AUTO_INCREMENT` column by using the `LAST_INSERT_ID()` function. From within the C API, use the `mysql_insert_id()` function. However, note that the two functions do not behave quite identically under all circumstances. The behavior of `INSERT` statements with respect to `AUTO_INCREMENT` columns is discussed further in [Section 12.8.3, "Information Functions"](#) and [Section 24.2.3.33, "mysql_insert_id\(\)"](#).

If you use an `INSERT ... VALUES` statement with multiple value lists or `INSERT ... SELECT`, the statement returns an information string in this format:

```
Records: 100 Duplicates: 0 Warnings: 0
```

`Records` indicates the number of rows processed by the statement. (This is not necessarily the number of rows actually inserted. `Duplicates` can be non-zero.) `Duplicates` indicates the number of rows that couldn't be inserted because they would duplicate some existing unique index value. `Warnings` indicates the number of attempts to insert column values that were problematic in some way. `Warnings` can occur under any of the following conditions:

- Inserting `NULL` into a column that has been declared `NOT NULL`. For multiple-row `INSERT` statements or `INSERT INTO ... SELECT` statements, the column is set to the implicit default value for the column data type. This is `0` for numeric types, the empty string (`' '`) for string types, and the ```zero``` value for date and time types. `INSERT INTO ... SELECT` statements are handled the same way as multiple-row inserts because the server does not examine the result set from the `SELECT` to see whether it returns a single row. (For a single-row `INSERT`, no warning occurs when `NULL` is inserted into a `NOT NULL` column. Instead, the statement fails with an error.)
- Setting a numeric column to a value that lies outside the column's range. The value is clipped to the closest endpoint of the range.
- Assigning a value such as `'10.34 a'` to a numeric column. The trailing non-numeric text is stripped off and the remaining numeric part is inserted. If the string value has

no leading numeric part, the column is set to 0.

- Inserting a string into a string column (CHAR, VARCHAR, TEXT, or BLOB) that exceeds the column's maximum length. The value is truncated to the column's maximum length.
- Inserting a value into a date or time column that is illegal for the column type. The column is set to the appropriate zero value for the type.

If you are using the C API, the information string can be obtained by invoking the `mysql_info()` function. See [Section 24.2.3.31, "mysql_info\(\)"](#).