3

Working with Data in MySQL

VIRTUALLY EVERYTHING YOU DO in MySQL involves data in some way or another because the purpose of a database management system is, by definition, to manage data. Even a simple `SELECT 1` statement involves expression evaluation to produce an integer data value.

Every data value in MySQL has a type. For example, `37.4` is a number and `'abc'` is a string. Sometimes data types are explicit, such as when you issue a `CREATE TABLE` statement that specifies the type for each column you define as part of the table:

```
CREATE TABLE mytbl
(
    int_col  INT,       # integer-valued column
    str_col  CHAR(20),  # string-valued column
    date_col DATE       # date-valued column
);
```

Other times data types are implicit, such as when you refer to literal values in an expression, pass values to a function, or use the value returned from a function. The following `INSERT` statement does all of those things:

```
INSERT INTO mytbl (int_col,str_col,date_col)
VALUES(14,CONCAT('a','b'),20050115);
```

The statement performs the following operations, all of which involve data types:

- It assigns the integer value `14` to the integer column `int_col`.
- It passes the string values `'a'` and `'b'` to the `CONCAT()` string-concatenation function. `CONCAT()` returns the string value `'ab'`, which is assigned to the string column `str_col`.
- It assigns the integer value `20050115` to the date column `date_col`. The assignment involves a type mismatch, but the integer value can reasonably be interpreted as a date value, so MySQL performs an automatic type conversion that converts the integer `20050115` to the date `'2005-01-15'`.

To use MySQL effectively, it's essential to understand how MySQL handles data. This chapter describes the types of data values that MySQL can handle, and discusses the issues involved in working with those types:

- The general categories of data values that MySQL can represent, including the `NULL` value.
- The specific data types MySQL provides for table columns, and the properties that characterize each data type. Some of MySQL's data types are fairly generic, such as the `BLOB` string type. Others behave in special ways that you should understand to avoid being surprised. These include the `TIMESTAMP` data type and integer types that have the `AUTO_INCREMENT` attribute.
- MySQL's capabilities for working with different character sets.

  Note: Support for multiple character sets was introduced beginning with MySQL 4.1, but underwent quite a bit of development during the early 4.1 releases. For best results, avoid early releases and use a recent 4.1 release instead.
- How to choose data types appropriately for your table columns. It's important to know how to pick the best type for your purposes when you create a table, and when to choose one type over another when several related types might be applicable to the kind of values you want to store.
- MySQL's rules for expression evaluation. You can use a wide range of operators and functions in expressions to retrieve, display, and manipulate data. Expression evaluation includes rules governing type conversion that come into play when a value of one type is used in a context requiring a value of another type. It's important to understand when type conversion happens and how it works; some conversions don't make sense and result in meaningless values. Assigning the string `'13'` to an integer column results in the value `13`, but assigning the string `'abc'` to that column results in the value `0` because `'abc'` doesn't look like a number. Worse, if you perform a comparison without knowing the conversion rules, you can do considerable damage, such as updating or deleting every row in a table when you intend to affect only a few specific rows. MySQL 5.0 introduces "strict" data-handling mode, which enables you to cause bad data values to be rejected.

Two appendixes provide additional information that supplements the discussion in this chapter about MySQL's data types, operators, and functions. These are Appendix B, "Data Type Reference," and Appendix C, "Operator and Function Reference."

The examples shown throughout this chapter use the `CREATE TABLE` and `ALTER TABLE` statements extensively to create and alter tables. These statements should be reasonably familiar to you because we have used them in Chapter 1, "Getting Started with MySQL and SQL," and Chapter 2, "MySQL SQL Syntax and Use." See also Appendix E, "SQL Syntax Reference."

MySQL supports several table types, each of which is managed by a different storage engine, and which differ in their properties. In some cases, a column with a given data type behaves differently for different storage engines, so the way you intend to use a column might determine or influence which storage engine to choose when you create a table. This chapter refers to storage engines on occasion, but a more detailed description of the available engines and their characteristics can be found in Chapter 2.

Data handling also depends in some cases on how default values are defined and on the current SQL mode. For general background on setting the SQL mode, see "The Server SQL Mode," in Chapter 2. In the current chapter, default value handing is covered in "Specifying Column Default Values." Strict mode and the rules for treatment of bad data are covered in "How MySQL Handles Invalid Data Values."

Categories of Data Values

MySQL knows about several general categories in which data values can be represented. These include numbers, string values, temporal values such as dates and times, spatial values, and the `NULL` value.

Numeric Values

Numbers are values such as `48` or `193.62`. MySQL understands numbers specified as integers (which have no fractional part) and floating-point or fixed-point values (which may have a fractional part). Integers can be specified in decimal or hexadecimal format.

An integer consists of a sequence of digits with no decimal point. In numeric contexts, an integer can be specified as a hexadecimal constant and is treated as a 64-bit integer. For example, 0x10 is 16 decimal. Hexadecimal values are treated as strings by default, so their syntax is given in the next section, "String Values."

A floating-point or fixed-point number consists of a sequence of digits, a decimal point, and another sequence of digits. The sequence of digits before or after the decimal point may be empty, but not both.

MySQL understands scientific notation. This is indicated by immediately following an integer or floating-point number with '`e`' or '`E`', a sign character ('`+`' or '`-`'), and an integer exponent. `1.34E+12` and `43.27e-1` are legal numbers in scientific notation. The number `1.34E12` is also legal even though it is missing an optional sign character before the exponent.

Hexadecimal numbers cannot be used in scientific notation; the '`e`' that begins the exponent part is also a legal hex digit and thus would be ambiguous.

Any number can be preceded by a plus or minus sign character ('`+`' or '`-`'), to indicate a positive or negative value.

As of MySQL 5.0.3, bit-field values can be written as `b'`*`val`*`'`, where *`val`* consists of one or more binary digits (0 or 1). For example, `b'1001'` is 9 decimal. This notation coincides with the introduction of the `BIT` data type, but bit-field values can be used more generally in other contexts.

```
mysql> SELECT b'101010' + 0;
+---------------+
| b'101010' + 0 |
+---------------+
|            42 |
+---------------+
mysql> SELECT CHAR(b'1100001');
+------------------+
| CHAR(b'1100001') |
+------------------+
| a                |
+------------------+
```

String Values

Strings are values such as `'Madison, Wisconsin'`, `'patient shows improvement'`, or even `'12345'` (which looks like a number, but isn't). Usually, you can use either single or double quotes to surround a string value, but there are two reasons to stick with single quotes:

- The SQL standard specifies single quotes, so statements that use single-quoted strings are more portable to other database engines.
- If the `ANSI_QUOTES` SQL mode is enabled, it treats the double quote as an identifier quoting character, not as a string quoting character. This means that a double-quoted value must refer to something like a database or table.

For the examples that use the double quote as a string quoting character in the discussion that follows, assume that `ANSI_QUOTES` mode is not enabled.

MySQL recognizes several escape sequences within strings that indicate special characters, as shown in Table 3.1. Each sequence begins with a backslash character ('\') to signify a temporary escape from the usual rules for character interpretation. Note that a NUL byte is not the same as the SQL `NULL` value; NUL is a zero-valued byte, whereas `NULL` in SQL signifies the absence of a value.

Table 3.1 **String Escape Sequences**

Sequence	Meaning
`\0`	NUL (zero-valued byte)
`\'`	Single quote
`\"`	Double quote
`\b`	Backspace
`\n`	Newline (linefeed)
`\r`	Carriage return
`\t`	Tab
`\\`	Single backslash
`\Z`	Ctrl-Z (Windows EOF character)

The escape sequences shown in the table are case sensitive, and any character not listed in the table is interpreted as itself if preceded by a backslash. For example, `\t` is a tab, but `\T` is an ordinary '`T`' character.

The table shows how to escape single or double quotes using backslash sequences, but you actually have several options for including quote characters within string values:

- Double the quote character if the string itself is quoted using the same character:

  ```
  'I can''t'
  "He said, ""I told you so."""
  ```

- Quote the string with the other quote character. In this case, you do not double the quote characters within the string:

  ```
  "I can't"
  'He said, "I told you so."'
  ```

- Escape the quote character with a backslash; this works regardless of the quote characters used to quote the string:

  ```
  'I can\'t'
  "I can\'t"
  "He said, \"I told you so.\""
  'He said, \"I told you so.\"'
  ```

To turn off the special meaning of backslash and treat it as an ordinary character, enable the `NO_BACKSLASH_ESCAPES` SQL mode, which is available as of MySQL 5.0.2.

As an alternative to using quotes for writing string values, you can use two forms of hexadecimal notation. The first consists of '`0x`' followed by one or more hexadecimal digits ('`0`' through '`9`' and '`a`' through '`f`'). For example, `0x0a` is 10 decimal, and `0xffff` is 65535 decimal. The non-decimal hex digits ('`a`' through '`f`') can be specified in uppercase or lowercase, but the leading '`0x`' cannot be given as '`0X`'. That is, `0x0a` and `0x0A` are legal hexadecimal values, but `0X0a` and `0X0A` are not. In string contexts, pairs of hexadecimal digits are interpreted as 8-bit numeric byte values in the range from 0 to 255, and the result is used as a string. In numeric contexts, a hexadecimal constant is treated as a number. The following statement illustrates the interpretation of a hex constant in each type of context:

```
mysql> SELECT 0x61626364, 0x61626364+0;
+------------+--------------+
| 0x61626364 | 0x61626364+0 |
+------------+--------------+
| abcd       |   1633837924 |
+------------+--------------+
```

If a hexadecimal value written using `0x` notation has an odd number of hex digits, MySQL treats it as though the value has a leading zero. For example, `0xa` is treated as `0x0a`.

String values may also be specified using the standard SQL notation `X'val'`, where *`val`* consists of pairs of hexadecimal digits. As with `0x` notation, such values are interpreted as strings, but may be used as numbers in a numeric context:

```
mysql> SELECT X'61626364', X'61626364'+0;
+-------------+---------------+
| X'61626364' | X'61626364'+0 |
+-------------+---------------+
| abcd        |    1633837924 |
+-------------+---------------+
```

Unlike `0x` notation, the leading 'X' is not case sensitive:

```
mysql> SELECT X'61', x'61';
+-------+-------+
| X'61' | x'61' |
+-------+-------+
| a     | a     |
+-------+-------+
```

Properties of Binary and Non-Binary Strings

String values fall into two general categories, binary and non-binary:

- A binary string is a sequence of bytes. These bytes are interpreted without respect to any concept of character set. A binary string has no special comparison or sorting properties. Comparisons are done byte by byte based on numeric byte values. Trailing spaces are significant in comparisons.
- A non-binary string is a sequence of characters. It is associated with a character set, which determines the allowable characters that may be used and how MySQL interprets the string contents. Character sets have one or more collating (sorting) orders. The particular collation used for a string determines the ordering of characters in the character set, which affects comparison operations. Trailing spaces are not significant in comparisons. The default character set and collation are `latin1` and `latin1_swedish_ci`.

Character units vary in their storage requirements. A single-byte character set such as `latin1` uses one byte per character, but there also are multi-byte character sets in which some or all characters require more than one byte. For example, both of the Unicode character sets available in MySQL are multi-byte. `ucs2` is a double-byte character set in which each character requires two bytes. `utf8` is a variable-length multi-byte character set with characters that take from one to three bytes.

To find out which character sets and collations are available in your server as it currently is configured, use these two statements:

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | ISO 8859-1 West European    | latin1_swedish_ci   |      1 |
...
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
...
+----------+-----------------------------+---------------------+--------+
mysql> SHOW COLLATION;
+----------------------+----------+-----+---------+----------+---------+
| Collation            | Charset  | Id  | Default | Compiled | Sortlen |
+----------------------+----------+-----+---------+----------+---------+
| big5_chinese_ci      | big5     |   1 | Yes     | Yes      |       1 |
| big5_bin             | big5     |  84 |         | Yes      |       1 |
...
| latin1_german1_ci    | latin1   |   5 |         |          |       0 |
| latin1_swedish_ci    | latin1   |   8 | Yes     | Yes      |       1 |
| latin1_danish_ci     | latin1   |  15 |         |          |       0 |
| latin1_german2_ci    | latin1   |  31 |         | Yes      |       2 |
| latin1_bin           | latin1   |  47 |         | Yes      |       1 |
| latin1_general_ci    | latin1   |  48 |         |          |       0 |
| latin1_general_cs    | latin1   |  49 |         |          |       0 |
| latin1_spanish_ci    | latin1   |  94 |         |          |       0 |
...
+----------------------+----------+-----+---------+----------+---------+
```

As shown by the output from `SHOW COLLATION`, each collation is specific to a given character set, but a given character set might have several collations. Collation names usually consist of a character set name, a language, and an additional suffix. For example, `utf8_icelandic_ci` is a collation for the `utf8` Unicode character set in which comparisons follow Icelandic sorting rules and characters are compared in case-insensitive fashion. Collation suffixes have the following meanings:

- `_ci` indicates a case-insensitive collation.
- `_cs` indicates a case-sensitive collation.
- `_bin` indicates a binary collation. That is, comparisons are based on character code values without reference to any language. For this reason, `_bin` collation names do not include any language name. Examples: `latin1_bin` and `utf8_bin`.

The sorting properties for binary and non-binary strings differ as follows:

- Binary strings are processed byte by byte in comparisons based solely on the numeric value of each byte. One implication of this property is that binary values appear to be case sensitive, but that actually is a side effect of the fact that uppercase and lowercase versions of a character have different numeric byte values. There isn't really any notion of lettercase for binary strings. Lettercase is a function of collation, which applies only to character (non-binary) strings.
- Non-binary strings are processed character by character in comparisons, and the relative value of each character is determined by the collating sequence that is used for the character set. For most collations, uppercase and lowercase versions of a given letter have the same collating value, so non-binary string comparisons typically are not case sensitive. However, that is not true for case-sensitive or binary collations.

Because collations are used for comparison and sorting, they affect many operations:

- Comparisons operators: `<`, `<=`, `=`, `<>`, `>=`, `>`, and `LIKE`.
- Sorting: `ORDER BY`, `MIN()`, and `MAX()`.
- Grouping: `GROUP BY` and `DISTINCT`.

To determine the character set or collation of a string, you can use the `CHARSET()` and `COLLATION()` functions.

Quoted string literals are interpreted according to the current server settings. The default character set and collation are `latin1` and `latin1_swedish_ci`:

```
mysql> SELECT CHARSET('abcd'), COLLATION('abcd');
+-----------------+-------------------+
| CHARSET('abcd') | COLLATION('abcd') |
+-----------------+-------------------+
| latin1          | latin1_swedish_ci |
+-----------------+-------------------+
```

MySQL treats hexadecimal constants as binary strings by default:

```
mysql> SELECT CHARSET(0x0123), COLLATION(0x123);
+-----------------+------------------+
| CHARSET(0x0123) | COLLATION(0x123) |
+-----------------+------------------+
| binary          | binary           |
+-----------------+------------------+
```

Two forms of notation can be used to force a string literal to be interpreted with a given character set:

- A string constant can be designated for interpretation with a given character set using the following notation, where *`charset`* is the name of a supported character set:

  `_charset str`

The _*charset* notation is called a "character set introducer." The string can be written as a quoted string or as a hexadecimal value. The following examples show how to cause strings to be interpreted in the `latin2` and `utf8` character sets:

```
_latin2 'abc'
_latin2 0x616263
_latin2 X'616263'
_utf8 'def'
_utf8 0x646566
_utf8 X'646566'
```

For quoted strings, whitespace is optional between the introducer and the following string. For hexadecimal values, whitespace is required.

- The notation `N'`*str*`'` is equivalent to `_utf8'`*str*`'`. `N` must be followed immediately by a quoted literal string with no intervening whitespace.

Introducer notation works for literal quoted strings or hexadecimal constants, but not for string expressions or column values. However, any string or string expression can be used to produce a string in a designated character set using the `CONVERT()` function:

```
CONVERT(str USING charset);
```

Introducers and `CONVERT()` are not the same. An introducer does not change the string value; it merely modifies how the string is interpreted. `CONVERT()` takes a string argument and produces a new string in the desired character set. To see the difference between introducers and `CONVERT()`, consider the following two statements that refer to the `ucs2` double-byte character set:

```
mysql> SET @s1 = _ucs2 'ABCD';
mysql> SET @s2 = CONVERT('ABCD' USING ucs2);
```

Assume that the default character set is `latin1` (a single-byte character set). The first statement interprets each pair of characters in the string `'ABCD'` as a single double-byte `ucs2` character, resulting in a two-character `ucs2` string. The second statement converts each character of the string `'ABCD'` to the corresponding `ucs2` character, resulting in a four-character `ucs2` string.

What is the "length" of each string? It depends. If you measure with `CHAR_LENGTH()`, you get the length in characters. If you measure with `LENGTH()`, you get the length in bytes:

```
mysql> SELECT CHAR_LENGTH(@s1), LENGTH(@s1), CHAR_LENGTH(@s2), LENGTH(@s2);
+------------------+-------------+------------------+-------------+
| CHAR_LENGTH(@s1) | LENGTH(@s1) | CHAR_LENGTH(@s2) | LENGTH(@s2) |
+------------------+-------------+------------------+-------------+
|                2 |           4 |                4 |           8 |
+------------------+-------------+------------------+-------------+
```

Here is a somewhat subtle point: A binary string is not the same thing as a non-binary string that has a binary collation. The binary string has no character set. It is interpreted with byte semantics and comparisons use single-byte numeric codes. A non-binary string with a binary collation has character semantics and comparisons use numeric character values that might be based on multiple bytes per character.

Here's one way to see the difference between binary and non-binary strings with regard to lettercase. Create a binary string and a non-binary string that has a binary collation, and then pass each string to the `UPPER()` function:

```
mysql> SET @s1 = BINARY 'abcd';
mysql> SET @s2 = _latin1 'abcd' COLLATE latin1_bin;
mysql> SELECT UPPER(@s1), UPPER(@s2);
+------------+------------+
| UPPER(@s1) | UPPER(@s2) |
+------------+------------+
| abcd       | ABCD       |
+------------+------------+
```

Why doesn't `UPPER()` convert the binary string to uppercase? This occurs because it has no character set, so there is no way to know which byte values correspond to uppercase or lowercase characters. To use a binary string with functions such as `UPPER()` and `LOWER()`, you must first convert it to a non-binary string:

```
mysql> SELECT @s1,  UPPER(CONVERT(@s1 USING latin1));
+------+----------------------------------+
| @s1  | UPPER(CONVERT(@s1 USING latin1)) |
+------+----------------------------------+
| abcd | ABCD                             |
+------+----------------------------------+
```

Character Set–Related System Variables

The server maintains several system variables that are involved in various aspects of character set support. Six of these variables refer to character sets and three refer to collations. Each of the collation variables is linked to a corresponding character set variable.

- `character_set_system` indicates the character set used for storing identifiers. This is always `utf8`.
- `character_set_server` and `collation_server` indicate the server's default character set and collation.
- `character_set_database` and `collation_database` indicate the character set and collation of the default database. These are read-only and set automatically by the server whenever you select a default database. If there is no default database, they're set to the server's default character set and collation. These variables come into play when you create a table but specify no explicit character set or collation. In this case, the table defaults are taken from the database defaults.

- The remaining variables influence how communication occurs between the client and the server:
    - `character_set_client` indicates the character set used for SQL statements that the client sends to the server.
    - `character_set_results` indicates the character set used for results that the server returns to the client. "Results" include data values and also metadata such as column names.
    - `character_set_connection` is used by the server. When it receives a statement string from the client, it converts the string from `character_set_client` to `character_set_connection` and works with the statement in the latter character set. (There is an exception: Any literal string in the statement that is preceded by a character set introducer is interpreted using the character set indicated by the introducer.) `collation_connection` is used for comparisons between literal strings within statement strings.

Very likely you'll find that most character set and collation variables are set to the same value by default. For example, the following output indicates that client/server communication takes place using the `latin1` character set:

```
mysql> SHOW VARIABLES LIKE 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | latin1 |
| character_set_connection | latin1 |
| character_set_database   | latin1 |
| character_set_results    | latin1 |
| character_set_server     | latin1 |
| character_set_system     | utf8   |
+--------------------------+--------+
mysql> SHOW VARIABLES LIKE 'collation\_%';
+----------------------+-------------------+
| Variable_name        | Value             |
+----------------------+-------------------+
| collation_connection | latin1_swedish_ci |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+----------------------+-------------------+
```

A client that wants to talk to the server using another character set can change the communication-related variables. For example, if you want to use `utf8`, change three variables:

```
mysql> SET character_set_client = utf8;
mysql> SET character_set_results = utf8;
mysql> SET character_set_connection = utf8;
```

However, it's more convenient to use a `SET NAMES` statement for this purpose. The following statement is equivalent to the preceding three `SET` statements:

```
mysql> SET NAMES 'utf8';
```

One restriction on setting the communication character set is that you cannot use `ucs2`.

Many client programs support a `--default-character-set` option that produces the same effect as a `SET NAMES` statement by informing the server of the desired communication character set.

For variables that come in pairs (a character set variable and a collation variable), the members of the pair are linked in the following ways:

- Setting the character set variable also sets the associated collation variable to the default collation for the character set.
- Setting the collation variable also sets the associated character set variable to the character set implied by the first part of the collation name.

For example, setting `character_set_connection` to `utf8` sets `collation_connection` to `utf8_general_ci`. Setting `collation_connection` to `latin1_spanish_ci` sets `character_set_connection` to `latin1`.

Date and Time (Temporal) Values

Dates and times are values such as `'2005-06-17'` or `'12:30:43'`. MySQL also understands combined date/time values, such as `'2005-06-17 12:30:43'`. Take special note of the fact that MySQL represents dates in year-month-day order. This often surprises newcomers to MySQL, although this is standard SQL format (also known as "ISO 8601" format). You can display date values any way you like using the `DATE_FORMAT()` function, but the default display format lists the year first, and input values must be specified with the year first.

Spatial Values

MySQL 4.1 and up supports spatial values, although currently only for MyISAM tables. This capability allows representation of values such as points, lines, and polygons. For example, the following statement uses the text representation of a point value with X and Y coordinates of (10, 20) to create a `POINT` and assigns the result to a user-defined variable:

```
SET @pt = POINTFROMTEXT('POINT(10 20)');
```

The `NULL` Value

`NULL` is something of a "typeless" value. Generally, it's used to mean "no value," "unknown value," "missing value," "out of range," "not applicable," "none of the above," and so forth. You can insert `NULL` values into tables, retrieve them from tables, and test whether a value is `NULL`. However, you cannot perform arithmetic on `NULL` values; if you try, the result is `NULL`. Also, many functions return `NULL` if you invoke them with a `NULL` argument.

MySQL Data Types

Each table in a database contains one or more columns. When you create a table using a `CREATE TABLE` statement, you specify a data type for each column. A data type is more specific than a general category such as "number" or "string." For a column, the data type is the means by which you precisely characterize the kind of values the column may contain, such as `SMALLINT` or `VARCHAR(32)`. This in turn determines how MySQL treats those values. For example, if you have numeric values, you could store them using either a numeric or string column, but MySQL will treat the values somewhat differently depending on what type you use. Each data type has several characteristics:

- What kind of values it can represent.
- How much space values take up, and whether the values are fixed-length (all values of the type take the same amount of space) or variable-length (the amount of space depends on the particular value being stored)
- How values of the type are compared and sorted
- Whether the type can be indexed

The following discussion surveys MySQL's data types briefly, and then describes in more detail the syntax for defining them and the properties that characterize each type, such as their range and storage requirements. The type specifications are shown as you use them in `CREATE TABLE` statements. Optional information is indicated by square brackets (`[]`). For example, the syntax `MEDIUMINT[(`*`M`*`)]` indicates that the maximum display width, specified as `(`*`M`*`)`, is optional. On the other hand, for `VARCHAR(`*`M`*`)`, the lack of brackets indicates that `(`*`M`*`)` is required.

Overview of Data Types

MySQL provides data types for values from all the general data value categories except the `NULL` value. `NULL` spans all types in the sense that the property of whether a column may contain `NULL` values is treated as a type attribute.

MySQL has numeric data types for integer, floating-point, and fixed-point values, as shown in Table 3.2. Numeric columns can be signed or unsigned. A special attribute allows sequential integer column values to be generated automatically, which is useful in applications that require a series of unique identification numbers.

Table 3.2 **Numeric Data Types**

Type Name	Meaning
`TINYINT`	A very small integer
`SMALLINT`	A small integer
`MEDIUMINT`	A medium-sized integer

Table 3.2 **Continued**

Type Name	Meaning
`INT`	A standard integer
`BIGINT`	A large integer
`FLOAT`	A single-precision floating-point number
`DOUBLE`	A double-precision floating-point number
`DECIMAL`	A fixed-point number
`BIT`	A bit field

Table 3.3 shows the MySQL string data types. Strings can hold anything, even arbitrary binary data such as images or sounds. Strings can be compared according to whether or not they are case sensitive. In addition, you can perform pattern matching on strings. (Actually, in MySQL, you can even perform pattern matching on numeric types, but it's more commonly done with string types.)

Table 3.3 **String Data Types**

Type Name	Meaning
`CHAR`	A fixed-length non-binary string
`VARCHAR`	A variable-length non-binary string
`BINARY`	A fixed-length binary string
`VARBINARY`	A variable-length binary string
`TINYBLOB`	A very small `BLOB` (binary large object)
`BLOB`	A small `BLOB`
`MEDIUMBLOB`	A medium-sized `BLOB`
`LONGBLOB`	A large `BLOB`
`TINYTEXT`	A very small non-binary string
`TEXT`	A small non-binary string
`MEDIUMTEXT`	A medium-sized non-binary string
`LONGTEXT`	A large non-binary string
`ENUM`	An enumeration; each column value may be assigned one enumeration member
`SET`	A set; each column value may be assigned zero or more set members

Table 3.4 shows the MySQL date and types, where *`CC`*, *`YY`*, *`MM`*, *`DD`* *`hh`*, *`mm`*, and *`ss`* represent century, year, month, day, hour, minute, and second, respectively. For temporal values, MySQL provides types for dates and times (either combined or separate) and timestamps (a special type that allows you to track when changes were last made to a record). There is also a type for efficiently representing year values when you don't need an entire date.

Table 3.4 **Date and Time Data Types**

Type Name	Meaning
`DATE`	A date value, in `'`*`CCYY-MM-DD`*`'` format
`TIME`	A time value, in `'`*`hh:mm:ss`*`'` format
`DATETIME`	A date and time value, in `'`*`CCYY-MM-DD hh:mm:ss`*`'` format
`TIMESTAMP`	A timestamp value, in `'`*`CCYY-MM-DD hh:mm:ss`*`'` format
`YEAR`	A year value, in *`CCYY`* format

Table 3.5 shows the MySQL spatial data types. These represent various kinds of geometrical or geographical values.

Table 3.5 **Spatial Data Types**

Type Name	Meaning
`GEOMETRY`	A spatial value of any type
`POINT`	A point (a pair of X,Y coordinates)
`LINESTRING`	A curve (one or more `POINT` values)
`POLYGON`	A polygon
`GEOMETRYCOLLECTION`	A collection of `GEOMETRY` values
`MULTILINESTRING`	A collection of `LINESTRING` values
`MULTIPOINT`	A collection of `POINT` values
`MULTIPOLYGON`	A collection of `POLYGON` values

Defining Column Types in Table Definitions

To create a table, issue a `CREATE TABLE` statement that includes a list of the columns in the table. Here's an example that creates a table named `mytbl` with three columns named `f`, `c`, and `i`:

```
CREATE TABLE mytbl
(
    f FLOAT(10,4),
    c CHAR(15) NOT NULL DEFAULT 'none',
    i TINYINT UNSIGNED NULL
);
```

Each column has a name and a type. Various attributes may be associated with the type. The syntax for defining a column is as follows:

`col_name col_type` `[`*`col_attributes`*`] [`*`general_attributes`*`]`

The name of the column, *`col_name`*, is always first in the definition and must be a legal identifier. The precise rules for identifier syntax are given in "MySQL Naming Rules," in Chapter 2. Briefly summarized, column identifiers may be up to 64 characters long, and may consist of alphanumeric characters from the system character set, as well as the underscore and dollar sign characters ('`_`' and '`$`'). Keywords such as `SELECT`, `DELETE`, and `CREATE` normally are reserved and cannot be used. However, you can include other characters within an identifier or use a reserved word as an identifier if you are willing to put up with the bother of quoting it whenever you refer to it. To quote an identifier, enclose it within backtick ('`` ` ``') characters. If the `ANSI_QUOTES` SQL mode is enabled, you also can quote identifiers within double quote ('`"`') characters.

`col_type` indicates the column data type; that is, the specific kind of values the column can hold. Some type specifiers indicate the maximum length of the values you store in the column. For others, the length is implied by the type name. For example, `CHAR(10)` specifies an explicit length of 10 characters, whereas `TINYTEXT` values have an implicit maximum length of 255 characters. Some of the type specifiers allow you to indicate a maximum display width (how many characters to use for displaying values). For floating-point and fixed-point types, you can specify the number of significant digits and number of decimal places.

Following the column's data type, you may specify optional type-specific attributes as well as more general attributes. These attributes function as type modifiers. They cause MySQL to change the way it treats column values in some way:

- The type-specific attributes that are allowable depend on the data type you choose. For example, `UNSIGNED` and `ZEROFILL` are allowable only for numeric types, and `CHARACTER SET` and `COLLATE` are allowable only for non-binary string types.
- The general attributes may be given for any data type, with a few exceptions. You may specify `NULL` or `NOT NULL` to indicate whether a column can hold `NULL` values. For most data types, you can specify a `DEFAULT` clause to define a default value for the column. Default value handling is described in the next section.

If multiple column attributes are present, there are some constraints on the order in which they may appear. In general, you should be safe if you specify data type-specific attributes such as `UNSIGNED` or `ZEROFILL` before general attributes such as `NULL` or `NOT NULL`.

Specifying Column Default Values

For all but `BLOB` and `TEXT` types, spatial types, or columns with the `AUTO_INCREMENT` attribute, you can specify a `DEFAULT` *`def_value`* clause to indicate that a column should be assigned the value *`def_value`* when a new row is created that does not explicitly specify the column's value. With some limited exceptions for `TIMESTAMP` columns, *`def_value`* must be a constant. It cannot be an expression or refer to other columns.

If a column definition includes no explicit `DEFAULT` clause and the column can take `NULL` values, its default value is `NULL`. Otherwise, the handling of a missing `DEFAULT` clause is version dependent:

- Before MySQL 5.0.2, MySQL defines the column with a `DEFAULT` clause that specifies the implicit default value. The implicit default depends on the column data type:
    - For numeric columns, the default is 0, except for integer columns that have the `AUTO_INCREMENT` attribute. For `AUTO_INCREMENT`, the default is the next number in the column sequence.
    - For date and time types except `TIMESTAMP`, the default is the "zero" value for the type (for example, `'0000-00-00'` for `DATE`). For `TIMESTAMP`, the default is the current date and time for the first `TIMESTAMP` column in a table, and the "zero" value for any following `TIMESTAMP` columns. (The `TIMESTAMP` defaults actually are more complex and are discussed in "The `TIMESTAMP` Data Type" later in this chapter.)
    - For string types other than `ENUM`, the default is the empty string. For `ENUM`, the default is the first enumeration element. For `SET`, the default when the column cannot contain `NULL` is actually the empty set, but that is equivalent to the empty string.
- From MySQL 5.0.2 on, the column is created without any `DEFAULT` clause. That is, it has no default value. This affects how the column is handled if new rows that do not specify a value for the column are inserted into the table:
    - When strict mode is not in effect, the column is set to the implicit default for its data type.
    - When strict mode is in effect, an error occurs if the table is transactional. The statement aborts and rolls back. For non-transactional tables, an error occurs and the statement aborts if the row is the first row inserted by the statement. If it is not the first row, you can elect to have the statement abort or to have the column set to its implicit default with a warning. The choice depends on which strict mode setting is in effect. See "How MySQL Handles Invalid Data Values" for details.

You can use the `SHOW CREATE TABLE` statement to see which columns have a `DEFAULT` value and what they are.

Numeric Data Types

MySQL's numeric data types group into three general classifications:

- Integer types are used for numbers that have no fractional part, such as `43`, `-3`, `0`, or `-798432`. You can use integer columns for data represented by whole numbers, such as weight to the nearest pound, height to the nearest inch, number of stars in a galaxy, number of people in a household, or number of bacteria in a petri dish.
- Floating-point and fixed-point types are used for numbers that may have a fractional part, such as `3.14159`, `-.00273`, `-4.78`, or `39.3E+4`. You can use these data types for values that may have a fractional part or that are extremely large or small.

Some types of data you might represent as floating-point values are average crop yield, distances, money values, unemployment rates, or stock prices.

- The `BIT` type is used for specifying bit-field values.

Floating-point values may be assigned to integer columns, but will be rounded to the nearest integer. Conversely, integer values may be assigned to floating-point or fixed-point columns. They are treated as having a fractional part of zero.

When you specify a number, you should not include commas as a separator. For example, `12345678.90` is legal, but `12,345,678.90` is not.

Table 3.6 shows the name and range of each numeric type, and Table 3.7 shows the amount of storage required for values of each type.

Table 3.6 **Numeric Data Type Ranges**

Type Specification	**Range**
`TINYINT[(`*`M`*`)]`	Signed values: –128 to 127 (-2^7 to 2^7-1); Unsigned values: 0 to 255 (0 to 2^8-1)
`SMALLINT[(`*`M`*`)]`	Signed values: –32768 to 32767 (-2^{15} to $2^{15}-1$); Unsigned values: 0 to 65535 (0 to $2^{16}-1$)
`MEDIUMINT[(`*`M`*`)]`	Signed values: –8388608 to 8388607 (-2^{23} to $2^{23}-1$); Unsigned values: 0 to 16777215 (0 to $2^{24}-1$)
`INT[(`*`M`*`)]`	Signed values: –2147683648 to 2147483647 (-2^{31} to $2^{31}-1$); Unsigned values: 0 to 4294967295 (0 to $2^{32}-1$)
`BIGINT[(`*`M`*`)]`	Signed values: –9223372036854775808 to 9223372036854775807 (-2^{63} to $2^{63}-1$); Unsigned values: 0 to 18446744073709551615 (0 to $2^{64}-1$)
`FLOAT[(`*`M,D`*`)]`	Minimum non-zero values: ±1.175494351E–38; Maximum non-zero values: ±3.402823466E+38
`DOUBLE[(`*`M,D`*`)]`	Minimum non-zero values: ±2.2250738585072014E–308; Maximum non-zero values: ±1.7976931348623157E+308
`DECIMAL([`*`M`*`[,`*`D`*`]])`	Varies; range depends on *M* and *D*
`BIT[(`*`M`*`)]`	0 to 2^M-1

Table 3.7 **Numeric Data Type Storage Requirements**

Type Specification	**Storage Required**
`TINYINT[(`*`M`*`)]`	1 byte
`SMALLINT[(`*`M`*`)]`	2 bytes
`MEDIUMINT[(`*`M`*`)]`	3 bytes
`INT[(`*`M`*`)]`	4 bytes
`BIGINT[(`*`M`*`)]`	8 bytes
`FLOAT[(`*`M,D`*`)]`	4 bytes

Table 3.7 **Continued**

Type Specification	Storage Required
`DOUBLE[(M,D)]`	8 bytes
`DECIMAL([M[,D]])`	*M*+2 bytes
`BIT[(M)]`	Varies depending on `M`

Integer Data Types

MySQL provides five integer types: `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, and `BIGINT`. `INTEGER` is a synonym for `INT`. These types vary in the range of values they can represent and in the amount of storage space they require. (The larger the range, the more storage is required.) Integer columns can be defined as `UNSIGNED` to disallow negative values; this shifts the range for the column upward to begin at 0.

When you define an integer column, you can specify an optional display size *M*. If given, *M* should be an integer from 1 to 255. It represents the number of characters used to display values for the column. For example, `MEDIUMINT(4)` specifies a `MEDIUMINT` column with a display width of 4. If you define an integer column without an explicit width, a default width is assigned. The defaults are the lengths of the "longest" values for each type. Note that displayed values are not chopped to fit within *M* characters. If the printable representation of a particular value requires more than *M* characters, MySQL displays the full value.

The display size *M* for an integer column relates only to the number of characters used to display column values. It has *nothing* to do with the number of bytes of storage space required. For example, `BIGINT` values require 8 bytes of storage regardless of the display width. It is not possible to magically cut the required storage space for a `BIGINT` column in half by defining it as `BIGINT(4)`. Nor does *M* have anything to do with the range of values allowed. If you define a column as `INT(3)`, that doesn't restrict it to a maximum value of 999.

The following statement creates a table to illustrate the default values of *M* and *D* for integer data types:

```
CREATE TABLE mytbl
(
    itiny     TINYINT,
    itiny_u   TINYINT UNSIGNED,
    ismall    SMALLINT,
    ismall_u  SMALLINT UNSIGNED,
    imedium   MEDIUMINT,
    imedium_u MEDIUMINT UNSIGNED,
    ireg      INT,
    ireg_u    INT UNSIGNED,
    ibig      BIGINT,
    ibig_u    BIGINT UNSIGNED
);
```

If you issue a `DESCRIBE mytbl` statement after creating the table, the number following each type name shows the value that MySQL uses by default in the absence of an explicit display width specifier:

```
mysql> DESCRIBE mytbl;
+-----------+-----------------------+------+-----+---------+-------+
| Field     | Type                  | Null | Key | Default | Extra |
+-----------+-----------------------+------+-----+---------+-------+
| itiny     | tinyint(4)            | YES  |     | NULL    |       |
| itiny_u   | tinyint(3) unsigned   | YES  |     | NULL    |       |
| ismall    | smallint(6)           | YES  |     | NULL    |       |
| ismall_u  | smallint(5) unsigned  | YES  |     | NULL    |       |
| imedium   | mediumint(9)          | YES  |     | NULL    |       |
| imedium_u | mediumint(8) unsigned | YES  |     | NULL    |       |
| ireg      | int(11)               | YES  |     | NULL    |       |
| ireg_u    | int(10) unsigned      | YES  |     | NULL    |       |
| ibig      | bigint(20)            | YES  |     | NULL    |       |
| ibig_u    | bigint(20) unsigned   | YES  |     | NULL    |       |
+-----------+-----------------------+------+-----+---------+-------+
```

Floating-Point and Fixed-Point Data Types

MySQL provides two floating-point types (`FLOAT`, `DOUBLE`), and one fixed-point type (`DECIMAL`). Synonymous types are `DOUBLE PRECISION` for `DOUBLE`, and `NUMERIC` and `FIXED` for `DECIMAL`. The `REAL` type is a synonym for `DOUBLE` by default. If the `REAL_AS_DEFAULT` SQL mode is enabled, `REAL` type is a synonym for `FLOAT`.

Ranges for these types differ from ranges for integer types in the sense that there is not only a maximum value that a floating-point type can represent, but also a minimum non-zero value. The minimum values provide a measure of how precise the type is, which is often important for recording scientific data. (There are, of course, corresponding negative maximum and minimum values.)

Floating-point and fixed-point types can be defined as `UNSIGNED`. Unlike the integer types, defining a floating-point or fixed-point type as `UNSIGNED` doesn't shift the type's range upward, it merely eliminates the negative end.

For each floating-point or fixed-point type, you may specify a maximum number of significant digits *`M`* and the number of decimal places *`D`*. The value of *`M`* should be from 1 to 255. The value of *`D`* may be from 0 to 30. If *`M`* is not greater than *`D`*, it is adjusted up to a value of *`D`*+1. *`M`* and *`D`* correspond to the concepts of "precision" and "scale" with which you may be familiar.

For `FLOAT` and `DOUBLE`, *`M`* and *`D`* are optional. If you omit both from the column definition, values are stored to the full precision allowed by your hardware.

For `DECIMAL`, *`M`* and *`D`* are optional. If *`D`* is omitted, it defaults to 0. If *`M`* is omitted as well, it defaults to 10. In other words, the following equivalences hold:

```
DECIMAL = DECIMAL(10) = DECIMAL(10,0)
DECIMAL(n) = DECIMAL(n,0)
```

FLOAT(*p*) syntax also is allowed. However, whereas *p* stands for the required number of bits of precision in standard SQL, it is treated differently in MySQL. *p* may range from 0 to 53 and is used only to determine whether the column stores single-precision or double-precision values. For *p* values from 0 to 24, the column is treated as single precision. For values from 25 to 53, the column is treated as double precision. That is, the column is treated as a FLOAT or DOUBLE with no *M* or *D* values.

The DECIMAL type is a fixed-point type. It differs from FLOAT and DOUBLE in that DECIMAL values actually are stored as strings and have a fixed number of decimals. The significance of this fact is that DECIMAL values are not subject to roundoff error the way that FLOAT and DOUBLE columns are—a property that makes DECIMAL especially applicable for storing currency values. The corresponding tradeoff is that DECIMAL values are not as efficient as floating-point values stored in native format that the processor can operate on directly. Also, be aware that the fixed-point properties of DECIMAL apply only to storage and retrieval. Calculations on DECIMAL values might be done using floating-point operations.

MySQL handles DECIMAL values according to the standard SQL specification, with one extension. Standard SQL requires that a type of DECIMAL(*M*,*D*) must be able to represent any value with *M* digits and *D* decimal places. For example, DECIMAL(4,2) must be able to represent values from –99.99 to 99.99. Because the sign character and decimal point must still be stored, this requires an extra two bytes, so DECIMAL(*M*,*D*) values use *M*+2 bytes. For DECIMAL(4,2), six bytes are needed for the "widest" value (–99.99).

The MySQL extension to standard SQL occurs at the positive end of the range. The sign byte is not needed to hold a sign character, so MySQL uses it to extend the range beyond that required by the SQL standard. In other words, for DECIMAL(4,2), the maximum value that can be stored in the six bytes available is 999.99.

There are two special conditions that reduce the DECIMAL storage requirement of *M*+2 bytes to a lesser value:

- If *D* is 0, DECIMAL values have no fractional part, and no byte need be allocated to store the decimal point. This reduces the required storage by one byte.
- If a DECIMAL column is UNSIGNED, no sign character need be stored, also reducing the required storage by one byte.

The maximum possible range for DECIMAL is the same as for DOUBLE, but the effective range is determined by the values of *M* and *D*. If you vary *M* and hold *D* fixed, the range becomes larger as *M* becomes larger. If you hold *M* fixed and vary *D*, the range becomes smaller as *D* becomes larger, although the precision increases. These properties are shown by Table 3.8 and Table 3.9.

Table 3.8 **How M Affects the Range of DECIMAL(M,D)**

Type Specification	Range
DECIMAL(4,1)	–999.9 to 9999.9
DECIMAL(5,1)	–9999.9 to 99999.9
DECIMAL(6,1)	–99999.9 to 999999.9

Table 3.9 **How `D` Affects the Range of `DECIMAL(M,D)`**

Type Specification	Range
`DECIMAL(4,0)`	–9999 to 99999
`DECIMAL(4,1)`	–999.9 to 9999.9
`DECIMAL(4,2)`	–99.99 to 999.99

The `BIT` Data Type

The `BIT` data type was introduced in MySQL 5.0.3 as a type for holding bit-field values. When you define a `BIT` column, you can specify an optional maximum width *`M`* that indicates the "width" of the column in bits. *`M`* should be an integer from 1 to 64. If omitted, *`M`* defaults to 1.

Values retrieved from a `BIT` column are not displayed in printable form by default. To display a printable representation of bit-field values, add zero. The `BIN()` function also can be useful for display bit-field values or the result of computations on them.

```
mysql> CREATE TABLE t (b BIT(3));  # holds values from 0 to 7
mysql> INSERT INTO t (b) VALUES(0),(b'11'),(b'101'),(b'111');
mysql> SELECT BIN(b+0), BIN(b & b'101'), BIN(b | b'101') FROM t;
+----------+-----------------+-----------------+
| BIN(b+0) | BIN(b & b'101') | BIN(b | b'101') |
+----------+-----------------+-----------------+
| 0        | 0               | 101             |
| 11       | 1               | 111             |
| 101      | 101             | 101             |
| 111      | 101             | 111             |
+----------+-----------------+-----------------+
```

Numeric Data Type Attributes

The `UNSIGNED` attribute disallows negative values. It can be used with all numeric types except `BIT`, but is most often used with integer types. Making an integer column `UNSIGNED` doesn't change the "size" of the underlying data type's range; it just shifts the range upward. Consider this table definition:

```
CREATE TABLE mytbl
(
    itiny   TINYINT,
    itiny_u TINYINT UNSIGNED
);
```

`itiny` and `itiny_u` both are `TINYINT` columns with a range of 256 values, but differ in the set of allowable values. The range of `itiny` is –128 to 127, whereas the range of `itiny_u` is shifted up, resulting in a range of 0 to 255.

`UNSIGNED` is useful for columns into which you plan to store information that doesn't take on negative values, such as population counts or attendance figures. Were you to use a signed column for such values, you would use only half of the data type's range. By making the column UNSIGNED, you effectively double your usable range. For example, if you use the column for sequence numbers, it will take twice as long to run out of values if you make it UNSIGNED.

You also can specify UNSIGNED for floating-point and fixed-point columns, although the effect is slightly different than for integer columns. The range does not shift upward; instead, the upper end remains unchanged and the lower end becomes zero.

The SIGNED attribute is allowed for all numeric types that allow UNSIGNED. However, it has no effect because such types are signed by default. SIGNED serves simply to indicate explicitly in a column definition that the column allows negative values.

The ZEROFILL attribute may be specified for all numeric types except BIT. It causes displayed values for the column to be padded with leading zeros to the display width. You can use ZEROFILL when you want to make sure column values always display using a given number of digits. Actually, it's more accurate to say "a given minimum number of digits" because values wider than the display width are displayed in full without being chopped. You can see this by issuing the following statements:

```
mysql> DROP TABLE IF EXISTS mytbl;
mysql> CREATE TABLE mytbl (my_zerofill INT(5) ZEROFILL);
mysql> INSERT INTO mytbl VALUES(1),(100),(10000),(1000000);
mysql> SELECT my_zerofill FROM mytbl;
+-------------+
| my_zerofill |
+-------------+
|       00001 |
|       00100 |
|       10000 |
|     1000000 |
+-------------+
```

Note that the final value, which is wider than the column's display width, is displayed in full.

If you specify the ZEROFILL attribute for a column, it automatically becomes UNSIGNED as well.

One other attribute, AUTO_INCREMENT, is intended only for use with integer data types. Specify the AUTO_INCREMENT attribute when you want to generate a series of unique identifier values. When you insert NULL into an AUTO_INCREMENT column, MySQL generates the next sequence value and stores it in the column. Normally, unless you take steps to cause otherwise, AUTO_INCREMENT values begin at 1 and increase by 1 for each new row. The sequence may be affected if you delete rows from the table. That is, sequence values might be reused; it is storage engine-dependent whether this occurs.

You can have at most one AUTO_INCREMENT column in a table. The column should have the NOT NULL constraint, and it must be indexed. Generally, an AUTO_INCREMENT column is indexed as a PRIMARY KEY or UNIQUE index. Also, because sequence values always are positive, you normally define the column `UNSIGNED` as well. For example, you can define an `AUTO_INCREMENT` column in any of the following ways:

```
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE);
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT, PRIMARY KEY (i));
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT, UNIQUE (i));
```

The first two forms specify the index information as part of the column definition. The second two specify the index as a separate clause of the `CREATE TABLE` statement. Using a separate clause is optional if the index includes only the `AUTO_INCREMENT` column. If you want to create a multiple-column index that includes the `AUTO_INCREMENT` column, you must use a separate clause. (For an example of this, see "`AUTO_INCREMENT` for MyISAM Tables.")

It is always allowable to define an `AUTO_INCREMENT` column explicitly as `NOT NULL`, but if you omit `NOT NULL`, MySQL adds it automatically.

"Working with Sequences" discusses the behavior of `AUTO_INCREMENT` columns further.

Following the attributes just described, which are specific to numeric columns, you may specify `NULL` or `NOT NULL`. If you do not specify `NULL` or `NOT NULL` for a numeric column, it allows `NULL` by default.

You also can specify a default value using the `DEFAULT` attribute. The following table contains three `INT` columns, having default values of `-1`, `1`, and `NULL`:

```
CREATE TABLE t
(
    i1 INT DEFAULT -1,
    i2 INT DEFAULT 1,
    i3 INT DEFAULT NULL
);
```

The rules that MySQL uses for assigning a default value if you specify no `DEFAULT` clause are given in "Specifying Column Default Values."

Choosing Numeric Data Types

When you choose a type for a numeric column, consider the range of values that you need to represent and choose the smallest type that will cover the range. Choosing a larger type wastes space, leading to tables that are unnecessarily large and that cannot be processed as efficiently as if you had chosen a smaller type. For integer values, `TINYINT` is the best if the range of values in your data is small, such as a person's age or number of siblings. `MEDIUMINT` can represent millions of values and can be used for many more types of values, at some additional cost in storage space. `BIGINT` has the largest range of all but requires twice as much storage as the next smallest integer type (`INT`) and should be used only when really necessary. For floating-point values, `DOUBLE` takes twice as much space as `FLOAT`. Unless you need exceptionally high precision or an extremely large range of values, you can probably represent your data at half the storage cost by using `FLOAT` instead of `DOUBLE`.

Every numeric column's range of values is determined by its type. If you attempt to insert a value that lies outside the column's range, the result depends on whether strict mode is enabled. If it is, an out of range value results in an error. If strict mode is not enabled, truncation occurs: MySQL clips the value to the appropriate endpoint of the range and uses the result.

Value truncation occurs according to the range of the data type, not the display width. For example, a `SMALLINT(3)` column has a display width of 3 and a range from –32768 to 32767. The value `12345` is wider than the display width but within the range of the column, so it is inserted without clipping and retrieved as `12345`. The value `99999` is outside the range, so it is clipped to `32767` when inserted. Subsequent retrievals return the value `32767`.

In general, values assigned to a floating-point or fixed-point column are rounded to the number of decimals indicated by the column specification. If you store `1.23456` in a `FLOAT(8,1)` column, the result is `1.2`. If you store the same value in a `FLOAT(8,4)` column, the result is `1.2346`. This means you should define floating-point columns with a sufficient number of decimals to store values as precise as you require. If you need accuracy to thousandths, don't define a type with only two decimal places.

String Data Types

MySQL provides several data types for storing string values. Strings are often used for text values like these:

```
'N. Bertram, et al.'
'Pencils (no. 2 lead)'
'123 Elm St.'
'Monograph Series IX'
```

But strings are actually "generic" types in a sense because you can use them to represent any value. For example, you can use binary string types to hold binary data, such as images, sounds, or compressed output from `gzip`.

Table 3.10 shows all the types provided by MySQL for defining string-valued columns, and the maximum size and storage requirements of each type. The `BLOB` and `TEXT` types each have several variants that are distinguished by the maximum size of values they can hold.

Some types hold binary strings (byte strings) and others hold non-binary strings (character strings). Thus, size and storage requirements are given in number of bytes per value for binary string types and number of characters for non-binary string types. For example, `BINARY(20)` holds 20 characters, whereas `CHAR(20)` holds 20 bytes. The differences between byte and character semantics for binary and non-binary strings are characterized in "String Values." Each of the binary string types for byte strings has a corresponding non-binary type for character strings, as shown in Table 3.11.

Each of the non-binary string types, as well as `ENUM` and `SET`, can be assigned a character set and collation. The MyISAM, MEMORY, and InnoDB storage engines include support for using multiple character sets within a single table. Character set assignment is discussed in "String Data Type Attributes."

Some string types are fixed-length. For a given column, each value requires the same amount of storage. Other string types are variable-length. The amount of storage taken by a value varies from row to row and depends on the length of the values actually stored in the column. This length is represented by *L* in the table for variable-length types. The extra bytes required in addition to *L* are the number of bytes needed to store the length of the value. MySQL handles variable-length values by storing both the content of the value and its length. These extra bytes are treated as an unsigned integer. There is a correspondence between a variable-length type's maximum length, the number of extra bytes required for that type, and the range of the unsigned integer type that uses the same number of bytes. For example, a `MEDIUMBLOB` value may be up to 2^{24}–1 bytes long and requires 3 bytes to record the length. The 3-byte integer type `MEDIUMINT` has a maximum unsigned value of 2^{24}–1. That's not a coincidence.

Table 3.10 **String Data Types**

Type Specification	Maximum Size	Storage Required
`BINARY[(M)]`	*M* bytes	*M* bytes
`VARBINARY(M)`	*M* bytes	*L* + 1 or 2 bytes
`CHAR[(M)]`	*M* characters	*M* characters
`VARCHAR(M)`	*M* characters	*L* characters + 1 or 2 bytes
`TINYBLOB`	2^{8}–1 bytes	*L* + 1 bytes
`BLOB`	2^{16}–1 bytes	*L* + 2 bytes
`MEDIUMBLOB`	2^{24}–1 bytes	*L* + 3 bytes
`LONGBLOB`	2^{32}–1 bytes	*L* + 4 bytes
`TINYTEXT`	2^{8}–1 characters	*L* characters + 1 byte
`TEXT`	2^{16}–1 characters	*L* characters + 2 bytes
`MEDIUMTEXT`	2^{24}–1 characters	*L* characters + 3 bytes
`LONGTEXT`	2^{32}–1 characters	*L* characters + 4 bytes
`ENUM('value1','value2',...)`	65,535 members	1 or 2 bytes
`SET('value1','value2',...)`	64 members	1, 2, 3, 4, or 8 bytes

The number of extra bytes for `VARBINARY` and `VARCHAR` is always one prior to MySQL 5.0.3 because the maximum column length is 255. As of MySQL 5.0.3, the column length can be up to 65,535, and two extra bytes are required for lengths greater than 255.

Table 3.11 **Corresponding Binary and Non-Binary String Types**

Binary String Type	Non-Binary String Type
`BINARY`	`CHAR`
`VARBINARY`	`VARCHAR`
`BLOB`	`TEXT`

Values for all string types except ENUM and SET are stored as a sequence of bytes and interpreted either as bytes or characters depending on whether the type holds binary or non-binary strings. Values that are too long are chopped to fit. (In strict mode, an error occurs instead unless the chopped characters are spaces.) But string types range from very small to very large, with the largest type able to hold nearly 4GB of data, so you should be able to find something long enough to avoid truncation of your information. (The effective maximum column size actually is imposed by the maximum packet size of the client/server communication protocol, which is 1GB.)

For ENUM and SET, the column definition includes a list of legal string values, but ENUM and SET values are stored internally as numbers, as detailed later in "The ENUM and SET Data Types." Attempting to store a value other than those in the list causes the value to be converted to '' (the empty string) unless strict mode is enabled. In strict mode, an error occurs instead.

The CHAR and VARCHAR Data Types

CHAR and VARCHAR are two of the most commonly used string types. They both hold non-binary strings, and thus are associated with a character set and collation.

Some significant changes were made in MySQL 5.0.3 with regard to the VARCHAR data type:

- The maximum length for VARCHAR was increased from 255 to 65,535.
- Trailing spaces no longer are stripped when VARCHAR values are stored.
- There is no automatic conversion between CHAR and VARCHAR columns. In earlier versions, this occurs under some circumstances, as described later in this section.

The primary differences between CHAR and VARCHAR lie in whether they have a fixed or variable length, and in how trailing spaces are treated:

- CHAR is a fixed-length type, whereas VARCHAR is a variable-length type.
- Values retrieved from CHAR columns have trailing spaces removed. For a CHAR(*M*) column, values that are shorter than *M* characters are padded to a length of *M* when stored, but trailing spaces are stripped when the values are retrieved.
- For a VARCHAR(*M*) column, handling of trailing spaces is version-dependent. Before MySQL 5.0.3, VARCHAR values have trailing spaces stripped when they are stored. No space stripping occurs at retrieval time because that has already occurred. This trailing space removal differs from standard SQL behavior. As of MySQL 5.0.3, trailing spaces in VARCHAR values are retained both for storage and retrieval.

CHAR columns can be defined with a maximum length *M* from 0 to 255. *M* is optional for CHAR and defaults to 1 if missing. Note that CHAR(0) is legal. A CHAR(0) column can be used to represent on/off values if you allow it to be NULL. Values in such a column can have one of two values: NULL or the empty string. A CHAR(0) column takes very little storage space in the table—only a single bit. It can be useful as a placeholder when you want to define a column but don't want to allocate space for it if you're not sure yet how wide to make it. You can use ALTER TABLE to widen the column later.

VARCHAR columns can be defined with a maximum length *M* from 0 to 255 before MySQL 5.0.3. As of MySQL 5.0.3, the length can be from 0 to 65,535. However, the actual maximum length of a VARCHAR column in practice may be less than 65,535, depending on storage engine internal row-size limits, the column character set, and the number of other columns in the table.

Keep in mind two general principles when choosing between CHAR and VARCHAR data types:

- If your values all are *M* characters long, a VARCHAR(*M*) column actually will use more space than a CHAR(*M*) column due to the extra byte or bytes required to record the length of values. On the other hand, if your values vary in length, VARCHAR columns have the advantage of taking less space. A CHAR(*M*) column always takes *M* characters, even if it is empty or NULL.
- If you're using MyISAM tables and your values don't vary much in length, CHAR is a better choice than VARCHAR because the MyISAM storage engine can process fixed-length rows more efficiently than variable-length rows. See "Data Type Choices and Query Efficiency," in Chapter 4, "Query Optimization."

Before MySQL 5.0.3, you cannot mix CHAR and VARCHAR within the same table, with a few limited exceptions. Depending on the circumstances, MySQL will even convert columns from one type to another when you create a table, something that other databases do not do. The principles that govern these conversions are as follows:

- Table rows are fixed-length only if all the columns in the table are fixed-length types.
- If even a single column has a variable length, table rows become variable length as well.
- If table rows are variable-length, MySQL converts any fixed-length columns in the table to their variable-length equivalents when doing so will save space.

What this means is that if you have VARCHAR, BLOB, or TEXT columns in a table, you cannot also have CHAR columns; MySQL silently converts them to VARCHAR. Suppose that you create a table like this:

```
CREATE TABLE mytbl
(
    c1 CHAR(10),
    c2 VARCHAR(10)
);
```

If you issue a DESCRIBE statement, the output is as follows:

```
mysql> DESCRIBE mytbl;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| c1    | varchar(10) | YES  |     | NULL    |       |
| c2    | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

Notice that the presence of the VARCHAR column causes MySQL to convert c1 to VARCHAR as well. If you try using ALTER TABLE to convert c1 to CHAR, it won't work. The only way to convert a VARCHAR column to CHAR is to convert all VARCHAR columns in the table at the same time:

```
mysql> ALTER TABLE mytbl MODIFY c1 CHAR(10), MODIFY c2 CHAR(10);
mysql> DESCRIBE mytbl;
+-------+----------+------+-----+---------+-------+
| Field | Type     | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| c1    | char(10) | YES  |     | NULL    |       |
| c2    | char(10) | YES  |     | NULL    |       |
+-------+----------+------+-----+---------+-------+
```

The BLOB and TEXT data types are variable-length like VARCHAR, but they have no fixed-length equivalent, so you cannot use CHAR columns in the same table as BLOB or TEXT columns. CHAR columns will be converted to VARCHAR. Even using ALTER TABLE to convert all VARCHAR columns to CHAR at the same time will fail. The presence of a BLOB or TEXT column requires that the rows be variable length, so MySQL will not convert the VARCHAR columns.

The exception to the prohibition on mixing fixed-length and variable-length columns is that CHAR columns shorter than four characters are not converted to VARCHAR. For example, MySQL will not change the CHAR column in the following table to VARCHAR:

```
CREATE TABLE mytbl
(
    c1 CHAR(2),
    c2 VARCHAR(10)
);
```

You can see this from the output of DESCRIBE:

```
mysql> DESCRIBE mytbl;
+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| c1    | char(2)     | YES  |     | NULL    |       |
| c2    | varchar(10) | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

There is a reason for not converting columns that are shorter than four characters: On average, any savings you might gain by not storing trailing spaces are offset by the extra byte needed in a `VARCHAR` column to record the length of each value. In fact, if all your columns are short, MySQL will convert any that you define as `VARCHAR` to `CHAR`. MySQL does this because the conversion decreases storage requirements on average and, for MyISAM tables, improves performance by making table rows fixed-length. Suppose that you create a table with the following specification:

```
CREATE TABLE mytbl
(
    c0 VARCHAR(0),
    c1 VARCHAR(1),
    c2 VARCHAR(2),
    c3 VARCHAR(3),
    c4 VARCHAR(4)
);
```

`DESCRIBE` reveals that MySQL silently changes all the `VARCHAR` columns shorter than four characters to `CHAR`:

```
mysql> DESCRIBE mytbl;
+-------+------------+------+-----+---------+-------+
| Field | Type       | Null | Key | Default | Extra |
+-------+------------+------+-----+---------+-------+
| c0    | char(0)    | YES  |     | NULL    |       |
| c1    | char(1)    | YES  |     | NULL    |       |
| c2    | char(2)    | YES  |     | NULL    |       |
| c3    | char(3)    | YES  |     | NULL    |       |
| c4    | varchar(4) | YES  |     | NULL    |       |
+-------+------------+------+-----+---------+-------+
```

As of MySQL 5.0.3, automatic conversion between `CHAR` and `VARCHAR` no longer occurs. A table can freely mix `CHAR` and `VARCHAR` columns.

The `BINARY` and `VARBINARY` Data Types

The `BINARY` and `VARBINARY` types are similar to `CHAR` and `VARCHAR`, with the following differences:

- `CHAR` and `VARCHAR` are non-binary types that store characters and have a character set and collation. Comparisons are based on the collating sequence.
- `BINARY` and `VARBINARY` are binary types that store bytes and have no character set or collation. Comparisons are based on numeric byte values.

Trailing space removal is the same for `BINARY` as for `CHAR`, and the same for `VARBINARY` as for `VARCHAR`. Also, prior to MySQL 5.0.3, `BINARY` columns may be converted to `VARBINARY` or vice versa when you create a table. The rules for when this occurs are the same as for conversion between `CHAR` and `VARCHAR`, as described in the previous section.

The BLOB and TEXT Data Types

A "BLOB" is a binary large object—basically, a container that can hold anything you want to toss into it, and that you can make about as big as you want. In MySQL, the BLOB type is really a family of types (TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB). These types are identical except in the maximum amount of information they can hold (see Table 3.10). BLOB columns store binary strings. They are useful for storing data that may grow very large or that may vary widely in size from row to row. Some examples are compressed data, encrypted data, images, and sounds.

MySQL also has a family of TEXT types (TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT). These are similar to the corresponding BLOB types, except that TEXT types store non-binary strings rather than binary strings. That is, they store characters rather than bytes, and are associated with a character set and collation. This results in the general differences between binary and non-binary strings that were described earlier in "String Values." For example, in comparison operations, BLOB values are compared in byte units and TEXT values are compared in character units using the column collation.

BLOB or TEXT columns sometimes can be indexed, depending on the storage engine you're using:

- MyISAM, InnoDB, and BDB tables support BLOB and TEXT indexing. However, you must specify a prefix size to be used for the index. This avoids creating index entries that might be huge and thereby defeat any benefits to be gained by that index. The exception is that prefixes are not used for FULLTEXT indexes on TEXT columns. FULLTEXT searches are based on the entire content of the indexed columns, so any prefix you specify is ignored.
- MEMORY tables do not support BLOB and TEXT indexes. This is because the MEMORY engine does not support BLOB or TEXT columns at all.

BLOB or TEXT columns may require special care:

- Due to the typical large variation in the size of BLOB and TEXT values, tables containing them are subject to high rates of fragmentation if many deletes and updates are done. If you're using a MyISAM table to store BLOB or TEXT values, you can run OPTIMIZE TABLE periodically to reduce fragmentation and maintain good performance. See Chapter 4 for more information.
- The max_sort_length system variable influences BLOB and TEXT comparison and sorting operations. Only the first max_sort_length bytes of each value are used. (For TEXT columns that use a multi-byte character set, this means that comparisons might involve fewer than max_sort_length characters.) If this causes a problem with the default max_sort_length value of 1024, you might want to increase the value before performing comparisons.
- If you're using very large values, you might need to configure the server to increase the value of the max_allowed_packet parameter. See Chapter 11, "General MySQL Administration," for more information. You will also need to increase the packet size for any client that wants to use very large values. The mysql and mysqldump clients support setting this value directly using a startup option.

The ENUM and SET Data Types

ENUM and SET are special string data types for which values must be chosen from a fixed (predefined) list of allowable strings. The primary difference between them is that ENUM column values must consist of exactly one member of the list of values, whereas SET column values may contain any or all members of the list. In other words, ENUM is used for values that are mutually exclusive, whereas SET allows multiple choices from the list.

The ENUM data type defines an enumeration. ENUM columns may be assigned values consisting of exactly one member chosen from a list of values specified at table-creation time. You can define an enumeration to have up to 65,535 members. Enumerations are commonly used to represent category values. For example, values in a column defined as ENUM('N','Y') can be either 'N' or 'Y'. Or you can use ENUM for such things as available sizes or colors for a product or for answers to multiple-choice questions in a survey or questionnaire where a single response must be selected:

```
employees ENUM('less than 100','100-500','501-1500','more than 1500')
color ENUM('red','green','blue','black')
size ENUM('S','M','L','XL','XXL')
vote ENUM('Yes','No','Undecided')
```

If you are processing selections from a Web page that includes mutually exclusive radio buttons, you can use an ENUM to represent the options from which a visitor to your site chooses. For example, if you run an online pizza ordering service, ENUM columns can be used to represent the type of crust and size of pizza a customer orders:

```
crust ENUM('thin','regular','pan style','deep dish')
size ENUM('small','medium','large')
```

If enumeration categories represent counts, it's important to choose your categories properly when you create the enumeration. For example, when recording white blood cell counts from a laboratory test, you might group the counts into categories like this:

```
wbc ENUM('0-100','101-300','>300')
```

If any given test result is provided as an exact count, you can record the value in the wbc column using the category into which the count falls. But you cannot recover the original count if you decide you want to convert the column from a category-based ENUM to an integer column based on exact count. If you really need the exact count, use an integer column instead. You can group integer values into categories when you retrieve them using the CASE construct. For example, if wbc is defined as an integer column, you can select it as a category like this:

```
SELECT CASE WHEN wbc <= 100 THEN '0-100'
            WHEN wbc <= 300 THEN '101-300'
            ELSE '>300' END AS 'wbc category'
FROM ...
```

The SET type is similar to ENUM in the sense that when you create a SET column, you specify a list of legal set members. But unlike ENUM, each column value may consist of any number of members from the set. The set may have up to 64 members. You can use a `SET` when you have a fixed set of values that are not mutually exclusive as they are in an `ENUM` column. For example, you might use a `SET` to represent options available for an automobile:

```
SET('luggage rack','cruise control','air conditioning','sun roof')
```

Then particular `SET` values would represent those options actually ordered by customers:

```
'cruise control,sun roof'
'luggage rack,air conditioning'
'luggage rack,cruise control,air conditioning'
'air conditioning'
''
```

The final value shown (the empty string) means that the customer ordered no options. This is a legal value for any `SET` column.

`SET` column definitions are written as a list of individual strings separated by commas to indicate what the set members are. A `SET` column value, on the other hand, is written as a single string. If the value consists of multiple set members, the members are separated within the string by commas. This means you shouldn't use a string containing a comma as a `SET` member.

Other uses for `SET` columns might be for representing information such as patient diagnoses or results from selections on Web pages. For a diagnosis, there may be a standard list of symptoms to ask a patient about, and the patient might exhibit any or all of them:

```
SET('dizziness','shortness of breath','cough')
```

For an online pizza service, the Web page for ordering could have a set of check boxes for ingredients that a customer wants as toppings on a pizza, several of which might be chosen:

```
SET('pepperoni','sausage','mushrooms','onions','ripe olives')
```

The way you define the legal value list for an `ENUM` or `SET` column is significant in several ways:

- The list determines the possible legal values for the column, as has already been discussed.
- If an `ENUM` or `SET` column has a collation that is not case sensitive, you can insert legal values in any lettercase and they will be recognized. However, the lettercase of the strings as specified in the column definition determines the lettercase of column values when they are retrieved later. For example, if you have an `ENUM('Y','N')` column and you store `'y'` and `'n'` in it, the values are displayed as `'Y'` and `'N'` when you retrieve them. If the column has a case sensitive or binary collation, you must insert values using exactly the lettercase used in the column definition or the values will not be recognized as legal. On the other hand, you can have distinct elements that differ only in lettercase, something that is not true when you use a collation that is not case sensitive.

- The order of values in an ENUM definition is the order used for sorting. The order of values in a SET definition also determines sort order, although the relationship is more complicated because column values may contain multiple set members.
- When MySQL displays a SET value that consists of multiple set members, the order in which it lists the members within the value is determined by the order in which they appear in the SET column definition.

ENUM and SET are classified as string types because enumeration and set members are specified as strings when you create columns of these types. However, the ENUM and SET types actually have a split personality: The members are stored internally as numbers and you can work with them as such. This means that ENUM and SET types are more efficient than other string types because they often can be handled using numeric operations rather than string operations. It also means that ENUM and SET values can be used in either string or numeric contexts. Finally, ENUM and SET columns can cause confusion if you use them in string context but expect them to behave as numbers, or vice versa.

MySQL sequentially numbers ENUM members in the column definition beginning with 1. (The value 0 is reserved for the error member, which is represented in string form by the empty string.) The number of enumeration values determines the storage size of an ENUM column. One byte can represent 256 values and two bytes can represent 65,536 values. (Compare this to the ranges of the one-byte and two-byte integer types TINYINT UNSIGNED and SMALLINT UNSIGNED.) Thus, counting the error member, the maximum number of enumeration members is 65,536 and the storage size depends on whether there are more than 256 members. You can specify a maximum of 65,535 (not 65,536) members in the ENUM definition because MySQL reserves a spot for the error member as an implicit member of every enumeration. When you assign an illegal value to an ENUM column, MySQL assigns the error member. (In strict mode, an error occurs instead.)

Here's an example you can try using the mysql client. It demonstrates that you can retrieve ENUM values in either string or numeric form (which shows the numeric ordering of enumeration members and also that the NULL value has no number in the ordering):

```
mysql> CREATE TABLE e_table (e ENUM('jane','fred','will','marcia'));
mysql> INSERT INTO e_table
    -> VALUES('jane'),('fred'),('will'),('marcia'),(''),(NULL);
mysql> SELECT e, e+0, e+1, e*3 FROM e_table;
+--------+------+------+------+
| e      | e+0  | e+1  | e*3  |
+--------+------+------+------+
| jane   |    1 |    2 |    3 |
| fred   |    2 |    3 |    6 |
| will   |    3 |    4 |    9 |
| marcia |    4 |    5 |   12 |
|        |    0 |    1 |    0 |
| NULL   | NULL | NULL | NULL |
+--------+------+------+------+
```

You can compare ENUM members either by name or number:

```
mysql> SELECT e FROM e_table WHERE e='will';
+------+
| e    |
+------+
| will |
+------+
mysql> SELECT e FROM e_table WHERE e=3;
+------+
| e    |
+------+
| will |
+------+
```

It is possible to define the empty string as a legal enumeration member, but this will only cause confusion. The string is assigned a non-zero numeric value, just as any other member listed in the definition. However, an empty string also is used for the error member that has a numeric value of 0, so it would correspond to two internal numeric element values. In the following example, assigning the illegal enumeration value 'x' to the ENUM column causes the error member to be assigned. This is distinguishable from the empty string member listed in the column definition only when retrieved in numeric form:

```
mysql> CREATE TABLE t (e ENUM('a','','b'));
mysql> INSERT INTO t VALUES('a'),(''),('b'),('x');
mysql> SELECT e, e+0 FROM t;
+------+------+
| e    | e+0  |
+------+------+
| a    |    1 |
|      |    2 |
| b    |    3 |
|      |    0 |
+------+------+
```

The numeric representation of SET columns is a little different than for ENUM columns. Set members are not numbered sequentially. Instead, members correspond to successive individual bits in the SET value. The first set member corresponds to bit 0, the second member corresponds to bit 1, and so on. In other words, the numeric values of SET members all are powers of two. The empty string corresponds to a numeric SET value of 0.

SET values are stored as bit values. Eight set members per byte can be stored this way, so the storage size for a SET column is determined by the number of set members, up to a maximum of 64 members. SET values take 1, 2, 3, 4, or 8 bytes for set sizes of 1 to 8, 9 to 16, 17 to 24, 25 to 32, and 33 to 64 members.

The representation of a SET as a set of bits is what allows a SET value to consist of multiple set members. Any combination of bits can be turned on in the value, so the value may consist of any combination of the strings in the `SET` definition that correspond to those bits.

The following example shows the relationship between the string and numeric forms of a `SET` column. The numeric value is displayed in both decimal and binary form:

```
mysql> CREATE TABLE s_table (s SET('table','lamp','chair','stool'));
mysql> INSERT INTO s_table
    -> VALUES('table'),('lamp'),('chair'),('stool'),(''),(NULL);
mysql> SELECT s, s+0, BIN(s+0) FROM s_table;
+-------+------+----------+
| s     | s+0  | BIN(s+0) |
+-------+------+----------+
| table |    1 | 1        |
| lamp  |    2 | 10       |
| chair |    4 | 100      |
| stool |    8 | 1000     |
|       |    0 | 0        |
| NULL  | NULL | NULL     |
+-------+------+----------+
```

If you assign to the column `s` a value of `'lamp,stool'`, MySQL stores it internally as 10 (binary 1010) because `'fred'` has a value of 2 (bit 1) and `'marcia'` has a value of 8 (bit 3).

When you assign values to `SET` columns, the substrings don't need to be listed in the same order that you used when you defined the column. However, when you retrieve the value later, members are displayed within the value in definition order. Also, if you assign to a `SET` column a value containing substrings that are not listed as set members, those strings drop out and the column is assigned a value consisting of the remaining substrings. When you retrieve the value later, the illegal substrings will not be present.

If you assign a value of `'chair,couch,table'` to the column `s` in `s_table`, two things happen:

- First, `'couch'` drops out because it's not a member of the set. This occurs because MySQL determines which bits correspond to each substring of the value to be assigned and turns them on in the stored value. `'couch'` corresponds to no bit and is ignored.
- Second, when you retrieve the value later, it appears as `'table,chair'`. On retrieval, MySQL constructs the string value from the numeric value by scanning the bits in order, which automatically reorders the substrings to the order used when the column was defined. This behavior also means that if you specify a set member more than once in a value, it will appear only once when you retrieve the value. If you assign `'lamp,lamp,lamp'` to a `SET` column, it will be simply `'lamp'` when retrieved.

In strict mode, use of an illegal `SET` member causes an error instead and the value is not stored. In the preceding example, assigning a value containing `'couch'` would cause an error and the assignment would fail.

The fact that MySQL reorders members in a `SET` value means that if you search for values using a string, you must list members in the proper order. If you insert `'chair,table'` and then search for `'chair,table'` you won't find the record; you must look for it as `'table,chair'`.

Sorting and indexing of `ENUM` and `SET` columns is done according to the internal (numeric) values of column values. The following example might appear to be incorrect because the values are not displayed in alphanumeric order:

```
mysql> SELECT e FROM e_table ORDER BY e;
+--------+
| e      |
+--------+
| NULL   |
|        |
| jane   |
| fred   |
| will   |
| marcia |
+--------+
```

You can better see what's going on by retrieving both the string and numeric forms of the `ENUM` values:

```
mysql> SELECT e, e+0 FROM e_table ORDER BY e;
+--------+------+
| e      | e+0  |
+--------+------+
| NULL   | NULL |
|        |    0 |
| jane   |    1 |
| fred   |    2 |
| will   |    3 |
| marcia |    4 |
+--------+------+
```

If you have a fixed set of values and you want them to sort in a particular order, you can exploit the `ENUM` sorting properties: Represent the values as an `ENUM` column in a table and list the enumeration values in the column definition in the order that you want them to be sorted. Suppose that you have a table representing personnel for a sports organization, such as a football team, and that you want to sort output by personnel position so that it comes out in a particular order, such as coaches, assistant coaches, quarterbacks, running backs, receivers, linemen, and so on. Define the column as an `ENUM` and list the enumeration elements in the order that you want to see them. Then column values automatically will come out in that order for sort operations.

For cases where you want an `ENUM` to sort in normal lexical order, you can convert the column to a non-`ENUM` string by using `CAST()` and sorting the result:

```
mysql> SELECT CAST(e AS CHAR) AS e_str FROM e_table ORDER BY e_str;
+--------+
| e_str  |
+--------+
| NULL   |
|        |
| fred   |
| jane   |
| marcia |
| will   |
+--------+
```

`CAST()` doesn't change the displayed values, but has the effect in this statement of performing an `ENUM`-to-string conversion that alters their sorting properties so they sort as strings.

String Data Type Attributes

The attributes unique to the string data types are `CHARACTER SET` and `COLLATE` for designating a character set and collating order. You can specify these as options for the table itself to set its defaults, or for individual columns to override the table defaults. (Actually, each database also has a default character set and collation, as does the server itself. These defaults sometimes come into play during table creation, as we'll see later.)

The `CHARACTER SET` and `COLLATION` attributes apply to the `CHAR`, `VARCHAR`, `TEXT`, `ENUM`, and `SET` data types. They do not apply to the binary string data types (`BINARY`, `VARBINARY`, and `BLOB`), because those types contain byte strings, not character strings.

When you specify the `CHARACTER SET` and `COLLATION` attributes, whether at the column, table, or database level, the following rules apply:

- The character set must be one that the server supports. To display the available character sets, use `SHOW CHARACTER SET`.
- If you specify both `CHARACTER SET` and `COLLATE`, the collation must be compatible with the character set. For example, with a character set of `latin2`, you could use a collation of `latin2_croatian_ci`, but not `latin1_bin`. To display the collations for each character set, use `SHOW COLLATION`.
- If you specify `CHARACTER SET` without `COLLATE`, the character set's default collation is used.
- If you specify `COLLATE` without `CHARACTER SET`, the character set is determined from the first part of the collation name.

To see how these rules apply, consider the following statement. It creates a table that uses several character sets:

```
CREATE TABLE mytbl
(
    c1  CHAR(10),
    c2  CHAR(40) CHARACTER SET latin2,
    c3  CHAR(10) COLLATE latin1_german1_ci,
    c4  BINARY(40)
) CHARACTER SET utf8;
```

The resulting table has `utf8` as its default character set. No `COLLATE` table option is given, so the default table collation is the default `utf8` collation (which happens to be `utf8_general_ci`). The definition for the `c1` column contains no `CHARACTER SET` or `COLLATE` attributes of its own, so the table defaults are used for it. On the other hand, the table-level character set and collation are not used for `c2`, `c3`, and `c4`: `c2` and `c3` have their own character set information, and `c4` has a binary string type, so the character set attributes do not apply. For `c2`, the collation is `latin2_general_ci`, the default collation for `latin2`. For `c3`, the character set is `latin1`, as can be determined from the collation name `latin1_german1_ci`.

To see character set information for an existing table, use `SHOW CREATE TABLE`:

```
mysql> SHOW CREATE TABLE mytbl\G
*************************** 1. row ***************************
       Table: mytbl
Create Table: CREATE TABLE `mytbl` (
  `c1` char(10) default NULL,
  `c2` char(40) character set latin2 default NULL,
  `c3` char(10) character set latin1 collate latin1_german1_ci default NULL,
  `c4` binary(40) default NULL
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

If `SHOW CREATE TABLE` does not display a column character set, it is the same as the table default character set. If it does not display a column collation, it is the default collation for the character set.

You also can add the `FULL` keyword to `SHOW COLUMNS` to cause it to display collation information (from which character sets can be determined):

```
mysql> SHOW FULL COLUMNS FROM mytbl;
+-------+------------+-------------------+------+-----+---------+...
| Field | Type       | Collation         | Null | Key | Default |...
+-------+------------+-------------------+------+-----+---------+...
| c1    | char(10)   | utf8_general_ci   | YES  |     | NULL    |...
| c2    | char(40)   | latin2_general_ci | YES  |     | NULL    |...
| c3    | char(10)   | latin1_german1_ci | YES  |     | NULL    |...
| c4    | binary(40) | NULL              | YES  |     | NULL    |...
+-------+------------+-------------------+------+-----+---------+...
```

The preceding discussion mentions column and table character set assignments, but character sets actually can be designated at the column, table, database, or server level. When MySQL processes a character column definition, it determines which character set to use for it by trying the following rules in order:

1. If the column definition includes a character set, use that set. (This includes the case where only a `COLLATE` attribute is present, because that implies which character set to use.)

2. Otherwise, if the table definition includes a character set table option, use that set.
3. Otherwise, use the database character set as the table default character set, which also becomes the column character set. If the database was never assigned a character set explicitly (for example, if it was a database created prior to upgrading to MySQL 4.1), the database character set is taken from the server character set.

In other words, MySQL searches up through the levels at which character sets may be specified until it finds a character set defined, and then uses that for the column's set. The database always has a default character set, so the search process is guaranteed to terminate at the database level even if no character set is specified explicitly at any of the lower levels.

The character set name `binary` is special. If you assign the `binary` character set to a non-binary string column, the result is a type conversion that forces the column to the corresponding binary string type. The following pairs of column definitions each show two equivalent definitions:

```
c1 CHAR(10) CHARACTER SET binary
c1 BINARY(10)

c2 VARCHAR(10) CHARACTER SET binary
c2 VARBINARY(10)

c3 TEXT CHARACTER SET binary
c3 BLOB
```

If you specify `CHARACTER SET binary` for a binary string column, it is ignored because the type already is binary. If you specify `CHARACTER SET binary` for an `ENUM` or `SET`, it is used as is.

If you assign the `binary` character set as a table option, it applies to each string column that does not have any character set information specified in its own definition.

MySQL provides some shortcut attributes for defining character columns:

- The `ASCII` attribute is shorthand for `CHARACTER SET latin1`.
- The `UNICODE` attribute is shorthand for `CHARACTER SET ucs2`.
- If you use the `BINARY` attribute for a non-binary string column, `ENUM`, or `SET`, it is shorthand for specifying the binary collation of the column's character set. For example, assuming a table default character set of `latin1`, these definitions are equivalent:

  ```
  c1 CHAR(10) BINARY
  c2 CHAR(10) CHARACTER SET latin1 BINARY
  c3 CHAR(10) CHARACTER SET latin1 COLLATE latin1_bin
  ```

  If you specify the `BINARY` attribute for a binary string column, it is ignored because the type already is binary.

The general attributes `NULL` or `NOT NULL` can be specified for any of the string types. If you don't specify either of them, `NULL` is the default. However, defining a string column as `NOT NULL` does not prevent you from storing an empty string (that is, `''`) in the column. In MySQL, an empty value is different from a missing value, so don't make the mistake of thinking that you can force a string column to contain non-empty values by defining it `NOT NULL`. If you require string values to be non-empty, that is a constraint you must enforce from within your own applications.

You also can specify a default value using the `DEFAULT` attribute for all string data types except the `BLOB` and `TEXT` types. The rules that MySQL uses for assigning a default value if you specify no `DEFAULT` clause are given in "Specifying Column Default Values."

Choosing String Data Types

When choosing a data type for a string column, consider the following questions:

- Are values represented as text or binary data? For text, non-binary string types are most appropriate. For binary data, use a binary string type.
- Do you want comparisons to be lettercase-aware? If so, use one of the non-binary string types, because those store characters and are associated with a character set and collation.

  You can control case sensitivity of non-binary string values for comparison and sorting purposes by the collation that you assign to them. If you want string values to be regarded equal regardless of lettercase, use a case-insensitive collation. Otherwise, use either a binary or case-sensitive collation. A binary collation compares character units using the numeric character codes. A case-sensitive collation compares character units using a specific collating order, which need not correspond to character code order. In either case, the lowercase and uppercase versions of a given character are considered distinct for comparisons. Suppose that `'mysql'`, `'MySQL'`, and `'MYSQL'` are strings in the `latin1` character set. They are all considered the same if compared using a case-insensitive collation such as `latin1_swedish_ci`, but as three different strings if compared using the binary `latin1_bin` collation or case-sensitive `latin1_general_cs` collation.

  If you want to use a string column both for case-sensitive and not case-sensitive comparisons, use a collation that corresponds to the type of comparison you will perform most often. For comparisons of the other type, apply the `COLLATE` operator to change the collation. For example, if `mycol` is a `CHAR` column that uses the `latin1` character set, you can assign it the `latin1_swedish_ci` collation to perform case-insensitive comparisons by default. The following comparison is not case sensitive:

  ```
  mycol = 'ABC'
  ```

  For those times when you need case-sensitive comparisons, use the `latin1_general_cs` or `latin1_bin` collation. The following comparisons are case sensitive (it doesn't matter whether you apply the `COLLATE` operator to the left hand string or the right hand string):

```
mycol COLLATE latin1_general_cs = 'ABC'
mycol COLLATE latin1_bin = 'ABC'
mycol = 'ABC' COLLATE latin1_general_cs
mycol = 'ABC' COLLATE latin1_bin
```

- Do you want to minimize storage requirements? If so, use a variable-length type, not a fixed-length type.
- Will the allowable set of values for the column always be chosen from a fixed set of legal values? If so, `ENUM` or `SET` might be a good choice.

  `ENUM` also can be useful if you have a limited set of string values that you want to sort in some non-lexical order. Sorting of `ENUM` values occurs according to the order in which you list the enumeration values in the column definition, so you can make the values sort in any order you want.
- Are trailing spaces significant? If values must be retrieved exactly as they are stored without having trailing spaces removed during storage or retrieval, use a `TEXT` column for non-binary strings and a `BLOB` column for binary strings. (As of MySQL 5.0.3, you also can use `VARCHAR` or `VARBINARY`.) This factor is important also if you are storing compressed, hashed, or encrypted values computed in such a way that the encoding method might result in trailing spaces. The following table shows how trailing spaces are handled for storage and retrieval operations for various string data types:

Data Type	**Storage**	**Retrieval**	**Result**
`CHAR`, `BINARY`	Padded	Removed	Retrieved values have no trailing spaces
`VARCHAR`, `VARBINARY` (< MySQL 5.0.3)	Removed	No action	Retrieved values have no trailing spaces
`VARCHAR`, `VARBINARY` (≥ MySQL 5.0.3)	No action	No action	Trailing spaces are not changed
`TEXT`, `BLOB`	No action	No action	Trailing spaces are not changed

Date and Time Data Types

MySQL provides several data types for storing temporal values: `DATE`, `TIME`, `DATETIME`, `TIMESTAMP`, and `YEAR`. Table 3.12 shows these types and the range of legal values for each type. The storage requirements for each type are shown in Table 3.13.

Table 3.12 **Date and Time Data Types**

Type Specification	Range
`DATE`	`'1000-01-01'` to `'9999-12-31'`
`TIME`	`'-838:59:59'` to `'838:59:59'`
`DATETIME`	`'1000-01-01 00:00:00'` to `'9999-12-31 23:59:59'`
`TIMESTAMP`	`'1970-01-01 00:00:00'` to partially through the year 2037
`YEAR[(M)]`	`1901` to `2155` for `YEAR(4)`, and `1970` to `2069` for `YEAR(2)`

Table 3.13 **Date and Time Data Type Storage Requirements**

Type Specification	Storage Required
`DATE`	3 bytes
`TIME`	3 bytes
`DATETIME`	8 bytes
`TIMESTAMP`	4 bytes
`YEAR`	1 byte

Each date and time type has a "zero" value that is stored when you insert a value that is illegal for the type, as shown in Table 3.14. The "zero" value also is the default value for date and time columns that are defined with the `NOT NULL` constraint.

Table 3.14 **Date and Time Type "Zero" Values**

Type Specification	Zero Value
`DATE`	`'0000-00-00'`
`TIME`	`'00:00:00'`
`DATETIME`	`'0000-00-00 00:00:00'`
`TIMESTAMP`	`'0000-00-00 00:00:00'`
`YEAR`	`0000`

MySQL always represents dates with the year first, in accordance with the standard SQL and ISO 8601 specifications. For example, December 3, 2004 is represented as `'2004-12-03'`. However, MySQL does allow some leeway in how you can specify input dates. For example, it will convert two-digit year values to four digits, and you need not supply a leading zero digit for month and day values that are less than 10. However, you must specify the year first and the day last. Formats that you may be more used to, such as `'12/3/99'` or `'3/12/99'`, will not be interpreted as you might intend. The date interpretation rules MySQL uses are discussed further in "Working with Date and Time Values."

For combined date and time values, it is also allowable to specify a '`T`' character rather than a space between the date and time (for example, `'2004-12-31T12:00:00'`).

Time or combined date and time values can include a microseconds part following the time, consisting of a decimal point and up to six digits. (For example, `'12:30:15.5'` or `'2005-06-15 10:30:12.000045'`.) However, current support for microsecond values is only partial. Some temporal functions use them, but you cannot store a temporal value that includes a microseconds part in a table; the microseconds part is discarded.

For retrieval, you can display date and time values in a variety of formats by using the `DATE_FORMAT()` and `TIME_FORMAT()` functions.

The `DATE`, `TIME`, and `DATETIME` Data Types

The `DATE` and `TIME` types hold date and time values. The `DATETIME` type holds combined date and time values. The formats for the three types of values are `'`*`CCYY-MM-DD`*`'`, `'`*`hh:mm:ss`*`'`, and `'`*`CCYY-MM-DD hh:mm:ss`*`'`, where *`CC`*, *`YY`*, *`MM`*, *`DD hh`*, *`mm`*, and *`ss`* represent century, year, month, day, hour, minute, and second, respectively.

For the `DATETIME` type, the date and time parts are both required; if you assign a `DATE` value to a `DATETIME` column, MySQL automatically adds a time part of `'00:00:00'`. Conversions work in the other direction as well. If you assign a `DATETIME` value to a `DATE` or `TIME` column, MySQL discards the part that is irrelevant:

```
mysql> CREATE TABLE t (dt DATETIME, d DATE, t TIME);
mysql> INSERT INTO t (dt,d,t) VALUES(NOW(), NOW(), NOW());
mysql> SELECT * FROM t;
+---------------------+------------+----------+
| dt                  | d          | t        |
+---------------------+------------+----------+
| 2004-07-17 16:30:44 | 2004-07-17 | 16:30:44 |
+---------------------+------------+----------+
```

MySQL treats the time in `DATETIME` and `TIME` values slightly differently. For `DATETIME`, the time part represents a time of day and must be in the range from `'00:00:00'` to `'23:59:59'`. A `TIME` value, on the other hand, represents elapsed time—that's why the range shown in Table 3.12 for `TIME` columns is so great and why negative values are allowed.

One thing to watch out when inserting `TIME` values into a table is that if you use a "short" (not fully qualified) value, it may not be interpreted as you expect. For example, you'll probably find that if you insert `'30'` and `'12:30'`, into a `TIME` column, one value will be interpreted from right to left and the other from left to right, resulting in stored values of `'00:00:30'` and `'12:30:00'`. If you consider `'12:30'` to represent a value of "12 minutes, 30 seconds," you should specify it in fully qualified form as `'00:12:30'`.

The `TIMESTAMP` Data Type

`TIMESTAMP` is a temporal data type that stores combined date and time values. (The word "timestamp" might appear to connote time only, but that is not the case.) The description of the `TIMESTAMP` data type in this section is current as of MySQL 4.1.6. Certain aspects of `TIMESTAMP` properties were in flux during earlier 4.1 releases, so avoid them and use a current release. The end of the section summarizes how the `TIMESTAMP` properties differ in MySQL 4.0 from the current properties.

The `TIMESTAMP` data type has several special properties:

- `TIMESTAMP` columns have a range of values from `'1970-01-01 00:00:00'` to partially through the year 2037. The range is tied to Unix time, where the first day of 1970 is "day zero," also known as "the epoch." Values are stored as a four-byte number of seconds since the epoch. The beginning of 1970 determines the lower end of the `TIMESTAMP` range. The upper end of the range corresponds to the maximum four-byte value for Unix time.
- Values are stored in Universal Coordinated Time (UTC). When you store a `TIMESTAMP` value, the server converts it from the current time zone to UTC. When you retrieve the value later, the server converts it back from UTC to the current time zone, so you see the same value that you stored. However, if another client connects to the server, uses a different time zone, and retrieves the value, it will see the value adjusted to its own time zone. In fact, you can see this effect within a single connection if you change your own time zone:

  ```
  mysql> CREATE TABLE t (ts TIMESTAMP);
  mysql> SET time_zone = '+00:00';   # set time zone to UTC
  mysql> INSERT INTO t VALUES('2000-01-01 00:00:00');
  mysql> SELECT ts FROM t;
  +---------------------+
  | ts                  |
  +---------------------+
  | 2000-01-01 00:00:00 |
  +---------------------+
  mysql> SET time_zone = '+03:00';   # advance time zone 3 hours
  mysql> SELECT ts FROM t;
  +---------------------+
  | ts                  |
  +---------------------+
  | 2000-01-01 03:00:00 |
  +---------------------+
  ```

  These examples specify time zones using values given as a signed offset in hours and minutes relative to UTC. You also can use named time zones such as `'Europe/Zurich'` if the server time zone tables have been set up as described in "Configuring Time Zone Support," in Chapter 11.
- `TIMESTAMP` has automatic initialization and update properties. You can designate any single `TIMESTAMP` column in a table to have either or both of these properties:
  - "Automatic initialization" means that for new records the column is set to the current timestamp if you omit it from the `INSERT` statement or set it to `NULL`.

- "Automatic update" means that for existing records the column is updated to the current timestamp when you change any other column. Setting a column to its current value does not count as a change. You must set it to a different value for automatic update to occur.

In addition, if you set any `TIMESTAMP` column to `NULL`, its value is set to the current timestamp. You can defeat this by defining the column with the `NULL` attribute to allow `NULL` values to be stored in the column.

Only one `TIMESTAMP` column in a table can be designated to have automatic properties. You cannot have automatic initialization for one `TIMESTAMP` column and automatic update for another. Nor can you have automatic initialization for multiple columns, or automatic update for multiple columns.

The syntax for specifying a `TIMESTAMP` column is as follows, assuming a column name of `ts`:

```
ts TIMESTAMP [DEFAULT value] [ON UPDATE CURRENT_TIMESTAMP]
```

The `DEFAULT` and `ON UPDATE` attributes can be given in any order, if both are given. The default value can be `CURRENT_TIMESTAMP` or a constant value such as `0` or a value in `'CCYY-MM-DD hh:mm:ss'` format. Synonyms for `CURRENT_TIMESTAMP` are `CURRENT_TIMESTAMP()` and `NOW()`; they're all interchangeable in a `TIMESTAMP` definition.

To have one or both of the automatic properties for the first `TIMESTAMP` column in a table, you can define it using various combinations of the `DEFAULT` and `ON UPDATE` attributes:

- With `DEFAULT CURRENT_TIMESTAMP`, the column has automatic initialization. It also has automatic update if `ON UPDATE CURRENT_TIMESTAMP` is given.
- With neither attribute, MySQL defines the column with both `DEFAULT CURRENT_TIMESTAMP` and `ON UPDATE CURRENT_TIMESTAMP`. (This preserves compatibility with table definitions from MySQL 4.0.)
- With a `DEFAULT` *`constant_value`* attribute that specifies a constant value, the column does not have automatic initialization. It does have automatic update if `ON UPDATE CURRENT_TIMESTAMP` is given.
- Without `DEFAULT` but with `ON UPDATE CURRENT_TIMESTAMP`, the default value is 0 and the column has automatic update.

To use automatic initialization or update for a different `TIMESTAMP` column than the first one, you must explicitly define the first one with a `DEFAULT` *`constant_value`* attribute and no `ON UPDATE CURRENT_TIMESTAMP` attribute. Then you can use `DEFAULT CURRENT_TIMESTAMP` or `ON UPDATE CURRENT_TIMESTAMP` (or both) with any other single `TIMESTAMP` column.

If you want to defeat automatic initialization or update for a `TIMESTAMP` column, set it explicitly to the desired value for insert or update operations. For example, you can prevent an update from changing the column by setting the column to its current value.

TIMESTAMP column definitions also can include NULL or NOT NULL. The default is NOT NULL. Its effect is that when you explicitly set the column to NULL, MySQL sets it to the current timestamp. (This is true both for inserts and updates.) If you specify NULL, setting the column to NULL stores NULL rather than the current timestamp.

If you want a table to contain a column that is set to the current timestamp for new records and that remains unchanged thereafter, you can achieve that two ways:

- Use a TIMESTAMP column declared as follows without an ON UPDATE attribute:

  ```
  ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  ```

  When you create a new record, initialize the column to the current timestamp by setting it to NULL or by omitting it from the INSERT statement. The column will retain its value for subsequent updates unless you change it explicitly.
- Use a DATETIME column. When you create a record, initialize the column to NOW(). Whenever you update the record thereafter, leave the column alone.

If you want a table to contain columns for both a time-created value and a last-modified value, use two TIMESTAMP columns:

```
CREATE TABLE t
(
    t_created  TIMESTAMP DEFAULT 0,
    t_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                         ON UPDATE CURRENT_TIMESTAMP
    ... other columns ...
);
```

When inserting a new record, set both TIMESTAMP columns to NULL to set them to the insertion timestamp. When updating an existing record, leave both columns alone; t_modified will be updated automatically to the modification timestamp.

During table creation, TIMESTAMP columns are subject to the setting of the SQL mode. If the MAXDB mode is enabled, any TIMESTAMP column is created as a DATETIME column instead. This is for compatibility with the MaxDB DBMS.

If you have been using MySQL 4.0, you'll notice that its handling of the TIMESTAMP data type differs in several ways from 4.1. Here's a brief characterization of the important differences for MySQL 4.0:

- TIMESTAMP display format is not the same as for DATETIME. Instead, MySQL displays TIMESTAMP values in 14-digit *CCYYMMDDhhmmss* format by default. A display width can be specified to display only part of the 14-digit value. For example, the display format for TIMESTAMP(10) values is *YYMMDDhhmm*.
- Only the first TIMESTAMP column is subject to automatic initialization and update, and you cannot disable these properties. You can defeat them only by assigning a non-NULL value to the column for specific statements.
- You cannot store NULL in a TIMESTAMP column. Setting the column to NULL always sets it to the current timestamp.
- There is only one time zone.

The `YEAR` Data Type

`YEAR` is a one-byte data type intended for efficient representation of year values. A `YEAR` column definition may include a specification for a display width *`M`*, which should be either 4 or 2. If you omit *`M`* from a `YEAR` definition, the default is 4. `YEAR(4)` has a range of `1901` to `2155`. `YEAR(2)` has a range of `1970` to `2069`, but only the last two digits are displayed. You can use the `YEAR` type when you want to store date information but only need the year part of the date, such as year of birth, year of election to office, and so forth. When you do not require a full date value, `YEAR` is much more space-efficient than other date types.

`TINYINT` has the same storage size as `YEAR` (one byte), but not the same range. To cover the same range of years as `YEAR` by using an integer type, you would need a `SMALLINT`, which takes twice as much space. If the range of years you need to represent coincides with the range of the `YEAR` type, `YEAR` is more space-efficient than `SMALLINT`. Another advantage of `YEAR` over an integer column is that MySQL converts two-digit values into four-digit values for you using MySQL's usual year-guessing rules. For example, `97` and `14` become `1997` and `2014`. However, be aware that inserting the numeric value `00` into a four-digit `YEAR` column results in the value `0000` being stored, not `2000`. If you want a value of `00` to convert to `2000`, you should specify it in string form as `'00'`.

Date and Time Data Type Attributes

The following remarks apply to all temporal types except `TIMESTAMP`:

- The general attributes `NULL` or `NOT NULL` may be specified. If you don't specify either of them, `NULL` is the default.
- You also can specify a default value using the `DEFAULT` attribute. The rules that MySQL uses for assigning a default value if you specify no `DEFAULT` clause are given in "Specifying Column Default Values."

  Note that because default values must be constants, you cannot use a function such as `NOW()` to supply a value of "the current date and time" as the default for a `DATETIME` column. To achieve that result, set the column value explicitly to `NOW()` whenever you create a new record, or else consider using a `TIMESTAMP` column instead.

`TIMESTAMP` columns are special; the default for the first such column in a table is the current date and time, and the "zero" value for any others. However, the full set of rules governing default values is more complex. See "The `TIMESTAMP` Data Type" for details.

Working with Date and Time Values

MySQL tries to interpret input values for date and time columns in a variety of formats, including both string and numeric forms. Table 3.16 shows the allowable formats for each of the date and time types.

Table 3.16 **Date and Time Type Input Formats**

Type	Allowable Formats
`DATETIME`, `TIMESTAMP`	`'`*`CCYY-MM-DD hh`*`:`*`mm`*`:`*`ss`*`'`
	`'`*`YY-MM-DD hh`*`:`*`mm`*`:`*`ss`*`'`
	`'`*`CCYYMMDDhhmmss`*`'`
	`'`*`YYMMDDhhmmss`*`'`
	`CCYYMMDDhhmmss`
	`YYMMDDhhmmss`
`DATE`	`'`*`CCYY-MM-DD`*`'`
	`'`*`YY-MM-DD`*`'`
	`'`*`CCYYMMDD`*`'`
	`'`*`YYMMDD`*`'`
	`CCYYMMDD`
	`YYMMDD`
`TIME`	`'`*`hh`*`:`*`mm`*`:`*`ss`*`'`
	`'`*`hhmmss`*`'`
	`hhmmss`
`YEAR`	`'`*`CCYY`*`'`
	`'`*`YY`*`'`
	`CCYY`
	`YY`

MySQL interprets formats that have no century part (*`CC`*) using the rules described in "Interpretation of Ambiguous Year Values." For string formats that include delimiter characters, you don't have to use '`-`' for dates and '`:`' for times. Any punctuation character may be used as the delimiter. Interpretation of values depends on context, not on the delimiter. For example, although times are typically specified using a delimiter of '`:`', MySQL won't interpret a value containing '`:`' as a time in a context where a date is expected. In addition, for the string formats that include delimiters, you need not specify two digits for month, day, hour, minute, or second values that are less than 10. The following are all equivalent:

```
'2012-02-03 05:04:09'
'2012-2-03 05:04:09'
'2012-2-3 05:04:09'
'2012-2-3 5:04:09'
'2012-2-3 5:4:09'
'2012-2-3 5:4:9'
```

Note that MySQL may interpret values with leading zeros in different ways depending on whether they are specified as strings or numbers. The string `'001231'` will be seen as a six-digit value and interpreted as `'2000-12-31'` for a `DATE`, and as `'2000-12-31 00:00:00'` for a `DATETIME`. On the other hand, the number `001231` will be seen as `1231` after the parser gets done with it and then the interpretation becomes problematic. This is a case where it's best to supply a string value `'001231'`, or else use a fully qualified value if you are using numbers (that is, `20001231` for `DATE` and `200012310000` for `DATETIME`).

In general, you may freely assign values between the `DATE`, `DATETIME`, and `TIMESTAMP` types, although there are certain restrictions to keep in mind:

- If you assign a `DATETIME` or `TIMESTAMP` value to a `DATE`, the time part is discarded.
- If you assign a `DATE` value to a `DATETIME` or `TIMESTAMP`, the time part of the resulting value is set to zero (`'00:00:00'`).
- The types have different ranges. In particular, `TIMESTAMP` has a more limited range (1970 to 2037); so, for example, you cannot assign a pre-1970 `DATETIME` value to a `TIMESTAMP` and expect reasonable results. Nor can you assign values to a `TIMESTAMP` that are far in the future.

MySQL provides many functions for working with date and time values. See Appendix C for more information.

Interpretation of Ambiguous Year Values

For all date and time types that include a year part (`DATE`, `DATETIME` `TIMESTAMP`, `YEAR`), MySQL handles values that contain two-digit years by converting them to four-digit years:

- Year values from 00 to 69 become 2000 to 2069
- Year values from 70 to 99 become 1970 to 1999

You can see the effect of these rules most easily by storing different two-digit values into a `YEAR` column and then retrieving the results. This demonstrates something you should take note of:

```
mysql> CREATE TABLE y_table (y YEAR);
mysql> INSERT INTO y_table VALUES(68),(69),(99),(00),('00');
mysql> SELECT * FROM y_table;
+------+
| y    |
+------+
| 2068 |
| 2069 |
| 1999 |
| 0000 |
| 2000 |
+------+
```

Notice that `00` is converted to `0000`, not to `2000`. That's because, as a number, `00` is the same as `0`, and is a perfectly legal value for the `YEAR` type. If you insert a numeric zero, that's what you get. To get `2000` using a value that does not contain the century, insert the string `'0'` or `'00'`. You can make sure that MySQL sees a string and not a number by inserting `YEAR` values using `CAST(`*`value`*` AS CHAR)` to produce a string result uniformly regardless of whether *`value`* is a string or a number.

In any case, keep in mind that the rules for converting two-digit to four-digit year values provide only a reasonable guess. There is no way for MySQL to be certain about the meaning of a two-digit year when the century is unspecified. If MySQL's conversion rules don't produce the values that you want, the solution is to provide unambiguous data with four-digit years.

Is MySQL Year-2000 Safe?

MySQL itself is year-2000 safe because it stores dates internally with four-digit years, but it's your responsibility to provide data that result in the proper values being stored in the first place. The real problem with two-digit year interpretation comes not from MySQL, but from the human desire to take a shortcut and enter ambiguous data. If you're willing to take the risk, go ahead. It's your risk to take, and MySQL's guessing rules are adequate for many situations. Just be aware that there are times when you really do need to enter four digits. For example, to enter birth and death dates into the president table, which lists U.S. presidents back into the 1700s, four-digit year values are in order. Values in these columns span several centuries, so letting MySQL guess the century from a two-digit year is definitely the wrong thing to do.

Spatial Data Types

MySQL 4.1 and up supports spatial values. This capability allows representation of values such as points, lines, and polygons. These data types are implemented per the OpenGIS specification, which is available at the Open Geospatial Consortium Web site:

`http://www.opengeospatial.org/`

The spatial data types allowed in MySQL are listed in Table 3.17.

Table 3.17 **Spatial Data Types**

Type Name	Meaning
`GEOMETRY`	A spatial value of any type
`POINT`	A point (a pair of X,Y coordinates)
`LINESTRING`	A curve (one or more `POINT` values)
`POLYGON`	A polygon
`GEOMETRYCOLLECTION`	A collection of `GEOMETRY` values
`MULTILINESTRING`	A collection of `LINESTRING` values
`MULTIPOINT`	A collection of `POINT` values
`MULTIPOLYGON`	A collection of `POLYGON` values

Currently, MySQL supports spatial types only for MyISAM tables, and indexed spatial columns do not allow `NULL` values. These restrictions may or may not matter to you. Probably the most significant implications to consider are these:

- MyISAM is a non-transactional storage engine, so you cannot use GIS types within transactional operations that require commit and rollback.
- You cannot use `NULL` to represent missing values within indexed columns. Depending on your application, it might be acceptable to use an empty (zero-dimensional) value instead.

MySQL works with spatial values in three formats. Well-Known Text (WKT) and Well-Known Binary (WKB) formats represent spatial values as text strings or in a standard binary format. The syntax for text strings and the binary representation are defined in the OpenGIS specification. For example, the WKT format for a `POINT` value with coordinates of *x* and *y* is written as a string:

```
'POINT(x y)'
```

Note the absence of a comma between the coordinate values. More complex values have a more complex string representation. The following strings represent a `LINESTRING` consisting of several points and a `POLYGON` that has a rectangular outer boundary and a triangular inner boundary:

```
'LINESTRING(10 20, 0 0, 10 20, 0 0)'
'POLYGON((0 0, 100 0, 100 100, 0 100, 0 0),(30 30, 30 60, 45 60, 30 30))'
```

The third format is the internal format that MySQL uses for storing spatial values in tables.

Because spatial values can be complex, most operations on them are done by invoking functions. The set of spatial functions is extensive and includes functions for converting from one format to another. For the complete list, see Appendix C.

The following example shows how to use several aspects of spatial support:

- It creates a table that includes a spatial column.
- It populates the table with some `POINT` values, using the `POINTFROMTEXT()` function that produces an internal-format value from a WKT representation.
- It creates a stored function that computes the distance between two points, using the `X()` and `Y()` functions to extract point coordinates.
- It computes the distance of each point in the table from a given reference point.

```
mysql> CREATE TABLE pt_tbl (p POINT);
mysql> INSERT INTO pt_tbl (p) VALUES
    ->    (POINTFROMTEXT('POINT(0 0)')),
    ->    (POINTFROMTEXT('POINT(0 50)')),
    ->    (POINTFROMTEXT('POINT(100 100)'));
mysql> CREATE FUNCTION dist (p1 POINT, p2 POINT)
    ->    RETURNS FLOAT
    ->    RETURN SQRT(POW(X(p2)-X(p1),2) + POW(Y(p2)-Y(p1),2));
mysql> SET @ref_pt = POINTFROMTEXT('POINT(0 0)');
mysql> SELECT ASTEXT(p), dist (p, @ref_pt) AS dist FROM pt_tbl;
```

```
+----------------+-----------------+
| ASTEXT(p)      | dist            |
+----------------+-----------------+
| POINT(0 0)     |               0 |
| POINT(0 50)    |              50 |
| POINT(100 100) | 141.42135620117 |
+----------------+-----------------+
```

How MySQL Handles Invalid Data Values

In the past, the dominant principle for data handling in MySQL has been, "Garbage in, garbage out." In other words, if you don't verify data values first before storing them, you may not like what you get back out. However, as of MySQL 5.0.2, several SQL modes were introduced that enable you to reject bad values and cause an error to occur instead. The following discussion first discusses how MySQL handles improper data by default, and then covers the changes that occur when you enable the various SQL modes that affect data handling.

MySQL's default handling of out-of-range or otherwise improper values is as follows:

- For numeric or `TIME` columns, values that are outside the legal range are clipped to the nearest endpoint of the range and the resulting value is stored.
- For string columns other than `ENUM` or `SET`, strings that are too long are truncated to fit the maximum length of the column. Assignments to an `ENUM` or `SET` column depend on the values that are listed as legal in the column definition. If you assign to an `ENUM` column a value that is not listed as an enumeration member, the error member is assigned instead (that is, the empty string that corresponds to the zero-valued member). If you assign to a `SET` column a value containing substrings that are not listed as set members, those strings drop out and the column is assigned a value consisting of the remaining members.
- For date or time columns, illegal values are converted to the appropriate "zero" value for the type (see Table 3.14).

These conversions are reported as warnings for `ALTER TABLE`, `LOAD DATA`, `UPDATE`, `INSERT INTO ... SELECT`, and multiple-row `INSERT` statements. In the `mysql` client, this information is displayed in the status line that is reported for a query. In a programming language, you may be able to get this information by some other means. If you're using the MySQL C or PHP APIs, you can invoke the `mysql_info()` function. With the Perl DBI API, you can use the `mysql_info` attribute of your database handle. The information provided is a count of the number of warnings.

To turn on stricter checking of data values for `INSERT` and `UPDATE`, enable one of the following SQL modes:

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES';
mysql> SET sql_mode = 'STRICT_TRANS_TABLES';
```

For transactional tables, both modes are identical. If an invalid or missing value is found, an error occurs, the statement aborts and rolls back, and has no effect. For non-transactional tables, the modes have the following effects:

- For both modes, if an invalid or missing value is found in the first row of a statement that inserts or updates rows, an error occurs. The statement aborts and has no effect, which is similar to what happens for transactional tables.
- If an error occurs after the first row in a statement that inserts or updates multiple rows, some rows already will have been modified. The two strict modes control whether the statement aborts at that point or continues to execute:
    - With `STRICT_ALL_TABLES`, an error occurs and the statement aborts. Rows affected earlier by the statement will already have been modified, so the result is a partial update.
    - With `STRICT_TRANS_TABLES`, MySQL aborts the statement for non-transactional tables only if doing so would have the same effect as for a transactional table. That is true only if the error occurs in the first row; an error in a later row leaves the earlier rows already changed. Those changes cannot be undone for a non-transactional table, so MySQL continues to execute the statement to avoid a partial update. It converts each invalid value to the closest legal value (as defined earlier in this section). For a missing value, MySQL sets the column to the implicit default for its data type. Implicit defaults are described in "Specifying Column Default Values."

Strict mode actually does not enable the strictest checking that MySQL can perform. You can use any or all of the following modes to impose additional constraints on input data:

- `ERROR_FOR_DIVIDE_BY_ZERO` causes errors for divide-by-zero operations.
- `NO_ZERO_DATE` prevents entry of the "zero" date value.
- `NO_ZERO_IN_DATE` prevents entry of incomplete date values that have a month or day part of zero.

For example, to enable strict mode for all table types and also check for divide-by-zero errors, set the SQL mode like this:

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES,ERROR_FOR_DIVIDE_BY_ZERO';
```

To turn on strict mode and all of the additional restrictions, you can simply enable `TRADITIONAL` mode:

```
mysql> SET sql_mode = 'TRADITIONAL';
```

`TRADITIONAL` is shorthand for "both strict modes, plus a bunch of other restrictions." This is more like the way that other "traditional" SQL DBMSs act with regard to data checking.

It is also possible to selectively weaken strict mode in some respects. If you enable the `ALLOW_INVALID_DATE` SQL mode, MySQL doesn't perform full checking of date parts.

Instead, it requires only that months be in the range from 1 to 12 and days be in the range from 1 to 31. Another way to suppress errors is to use the `IGNORE` keyword with `INSERT` or `UPDATE` statements. With `IGNORE`, statements that would result in an error due to invalid values result only in a warning.

The various options available give you the flexibility to choose the level of validity checking that is appropriate for your applications.

Working with Sequences

Many applications need to generate unique numbers for identification purposes. The requirement for unique values occurs in a number of contexts: membership numbers, sample or lot numbering, customer IDs, bug report or trouble ticket tags, and so forth.

MySQL's mechanism for providing unique numbers is through the `AUTO_INCREMENT` column attribute, which enables you to generate sequential numbers automatically. However, `AUTO_INCREMENT` columns are handled somewhat differently by the various storage engines that MySQL supports, so it's important to understand not only the general concepts underlying the `AUTO_INCREMENT` mechanism, but also the differences between storage engines. This section describes how `AUTO_INCREMENT` columns work in general and for specific storage engines so that you can use them effectively without running into the traps that sometimes surprise people. It also describes how you can generate a sequence without using an `AUTO_INCREMENT` column.

General `AUTO_INCREMENT` Concepts

`AUTO_INCREMENT` columns must be defined according to the following conditions:

- There can be only one column per table with the `AUTO_INCREMENT` attribute and it should have an integer data type.
- The column must be indexed. It is most common to use a `PRIMARY KEY` or `UNIQUE` index, but it is allowable to use a non-unique index.
- The column must have a `NOT NULL` constraint. MySQL makes the column `NOT NULL` even if you don't explicitly declare it that way.

Once created, an `AUTO_INCREMENT` column behaves like this:

- Inserting `NULL` into an `AUTO_INCREMENT` column causes MySQL to generate the next sequence number automatically and insert that value into the column. `AUTO_INCREMENT` sequences normally begin at 1 and increase monotonically, so successive records inserted into a table get sequence values of 1, 2, 3, and so forth. Under some circumstances and depending on the storage engine, it may be possible to set or reset the next sequence number explicitly or to reuse values deleted from the top end of the sequence.
- The value of the most recently generated sequence number can be obtained by calling the `LAST_INSERT_ID()` function, as long as you call it during the same connection that was used to generate the number. This enables you to reference

the AUTO_INCREMENT value in subsequent statements even without knowing what the value is. LAST_INSERT_ID() returns 0 if no AUTO_INCREMENT value has been generated during the current connection.

LAST_INSERT_ID() is tied only to AUTO_INCREMENT values generated during the current connection to the server. In particular, it is not affected by AUTO_INCREMENT activity associated with other clients. You can generate a sequence number, and then call LAST_INSERT_ID() to retrieve it later, even if other clients have generated their own sequence values in the meantime.

For a multiple-row INSERT that generates several AUTO_INCREMENT values, LAST_INSERT_ID() returns the first one.

- Inserting a row without specifying an explicit value for the AUTO_INCREMENT column is the same as inserting NULL into the column. If ai_col is an AUTO_INCREMENT column, these statements are equivalent:

```
INSERT INTO t (ai_col,name) VALUES(NULL,'abc');
INSERT INTO t (name) VALUES('abc');
```

- By default, inserting 0 into an AUTO_INCREMENT column has the same effect as inserting NULL. If you enable the NO_AUTO_VALUE_ON_ZERO SQL mode, inserting a 0 results in a 0 being stored, not the next sequence value.
- If you insert a record and specify a non-NULL, non-zero value for an AUTO_INCREMENT column that has a unique index, one of two things will happen. If a record already exists with that value, a duplicate-key error occurs. If a record does not exist with that value, the record is inserted with the AUTO_INCREMENT column set to the given value. If the value is larger than the current next sequence number, the sequence is reset to continue with the next value after that for subsequent rows. In other words, you can "bump up" the counter by inserting a record with a sequence value greater than the current counter value.

Bumping up the counter can result in gaps in the sequence, but you also can exploit this behavior to generate a sequence that begins at a value higher than 1. Suppose that you create a table with an AUTO_INCREMENT column, but you want the sequence to begin at 1000 rather than at 1. To achieve this, insert a "fake" record with a value of 999 in the AUTO_INCREMENT column. Records inserted subsequently are assigned sequence numbers beginning with 1000, after which you can delete the fake record.

Why might you want to begin a sequence with a value higher than 1? One reason is to make sequence numbers all have the same number of digits. If you're generating customer ID numbers, and you expect never to have more than a million customers, you could begin the series at 1,000,000. You'll be able to add well over a million customer records before the digit count for customer ID values changes. Other reasons for not beginning a sequence at 1 might have nothing to do with technical considerations. For example, if you were assigning membership numbers, you might want to begin a sequence at a number higher than 1 to forestall political squabbling over who gets to be member number 1, by making sure there isn't any such number. Hey, it happens. Sad, but true.

- For some storage engines, values deleted from the top of a sequence are reused. In this case, if you delete the record containing the largest value in an AUTO_INCREMENT column, that value is reused the next time you generate a new value. An implication of this property is that if you delete all the records in the table, all values are reused and the sequence starts over beginning at 1.
- If you use UPDATE to set an AUTO_INCREMENT column to a value that already exists in another row, a duplicate-key error occurs if the column has a unique index. If you update the column to a value larger than any existing column value, the sequence continues with the next number after that for subsequent records. If you update the column to 0, it is set to 0 (this is true regardless of whether NO_AUTO_VALUE_ON_ZERO is enabled).
- If you use REPLACE to update a record based on the value of the AUTO_INCREMENT column, the AUTO_INCREMENT value does not change. If you use REPLACE to update a record based on the value of some other PRIMARY KEY or UNIQUE index, the AUTO_INCREMENT column is updated with a new sequence number if you set it to NULL, or if you set it to 0 and NO_AUTO_VALUE_ON_ZERO is not enabled.

AUTO_INCREMENT **Handling Per Storage Engine**

The general AUTO_INCREMENT characteristics just described form the basis for understanding sequence behavior specific to other storage engines. Most engines implement behavior that for the most part is similar to that just described, so keep the preceding discussion in mind as you read on.

AUTO_INCREMENT **for MyISAM Tables**

MyISAM tables offer the most flexibility for sequence handling. The MyISAM storage engine has the following AUTO_INCREMENT characteristics:

- MyISAM sequences normally are monotonic. The values in an automatically generated series are strictly increasing and are not reused if you delete records. If the maximum value is 143 and you delete the record containing that value, MySQL still generates the next value as 144. There are two exceptions to this behavior. First, if you empty a table with TRUNCATE TABLE, the counter may be reset to begin at 1. Second, values deleted from the top of a sequence are reused if you use a composite index to generate multiple sequences within a table. (This technique is discussed shortly.)
- MyISAM sequences begin at 1 by default, but it is possible to start the sequence at a higher value. With MyISAM tables, you can specify the initial value explicitly by using an AUTO_INCREMENT = *n* option in the CREATE TABLE statement. The following example creates a MyISAM table with an AUTO_INCREMENT column named seq that begins at 1,000,000:

```
CREATE TABLE mytbl
(
    seq INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (seq)
) ENGINE = MYISAM AUTO_INCREMENT = 1000000;
```

A table can have only one `AUTO_INCREMENT` column, so there is never any ambiguity about the column to which the terminating `AUTO_INCREMENT =` *n* option applies, even if the table has multiple columns.

- You can change the current sequence counter for an existing MyISAM table with `ALTER TABLE`. If the sequence currently stands at 1000, the following statement causes the next number generated to be 2000:

  ```
  ALTER TABLE mytbl AUTO_INCREMENT = 2000;
  ```

  If you want to reuse values that have been deleted from the top of the sequence, you can do that, too. The following statement will set the counter down as far as possible, causing the next number to be one larger than the current maximum sequence value:

  ```
  ALTER TABLE mytbl AUTO_INCREMENT = 1;
  ```

  You cannot use the `AUTO_INCREMENT` option to set the current counter lower than the current maximum value in the table. If an `AUTO_INCREMENT` column contains the values 1 and 10, using `AUTO_INCREMENT = 5` sets the counter so that the next automatic value is 11.
- If you use `INSERT DELAYED`, the `AUTO_INCREMENT` value is not generated until the record actually is inserted. In this case, `LAST_INSERT_ID()` cannot be relied on to return the sequence value.

The MyISAM storage engine supports the use of composite (multiple-column) indexes for creating multiple independent sequences within the same table. To use this feature, create a multiple-column `PRIMARY KEY` or `UNIQUE` index that includes an `AUTO_INCREMENT` column as its final column. For each distinct key in the leftmost column or columns of the index, the `AUTO_INCREMENT` column will generate a separate sequence of values. For example, you might use a table named `bugs` for tracking bug reports of several software projects, where the table is defined as follows:

```
CREATE TABLE bugs
(
    proj_name   VARCHAR(20) NOT NULL,
    bug_id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
    description VARCHAR(100),
    PRIMARY KEY (proj_name, bug_id)
) ENGINE = MYISAM;
```

Here, the `proj_name` column identifies the project name and the `description` column contains the bug description. The `bug_id` column is an `AUTO_INCREMENT` column; by creating an index that ties it to the `proj_name` column, you can generate an independent series of sequence numbers for each project. Suppose that you enter the following records into the table to register three bugs for SuperBrowser and two for SpamSquisher:

```
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SuperBrowser','crashes when displaying complex tables');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SuperBrowser','image scaling does not work');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SpamSquisher','fails to block known blacklisted domains');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SpamSquisher','fails to respect whitelist addresses');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SuperBrowser','background patterns not displayed');
```

The resulting table contents are as follows:

```
mysql> SELECT * FROM bugs ORDER BY proj_name, bug_id;
+--------------+--------+------------------------------------------+
| proj_name    | bug_id | description                              |
+--------------+--------+------------------------------------------+
| SpamSquisher |      1 | fails to block known blacklisted domains |
| SpamSquisher |      2 | fails to respect whitelist addresses     |
| SuperBrowser |      1 | crashes when displaying complex tables   |
| SuperBrowser |      2 | image scaling does not work              |
| SuperBrowser |      3 | background patterns not displayed        |
+--------------+--------+------------------------------------------+
```

The table numbers the `bug_id` values for each project separately, regardless of the order in which records are entered for projects. You need not enter all records for one project before you enter records for another.

If you use a composite index to create multiple sequences, values deleted from the top of each individual sequence *are* reused. This contrasts with the usual MyISAM behavior of not reusing values.

`AUTO_INCREMENT` for MEMORY Tables

The MEMORY storage engine has the following `AUTO_INCREMENT` characteristics:

- The initial sequence value can be set with an `AUTO_INCREMENT = n` table option in the `CREATE TABLE` statement, and can be modified after table creation time using that option with `ALTER TABLE`.
- Values that are deleted from the top of the sequence are not reused. Exception: If you empty the table with `TRUNCATE TABLE`, the sequence may be reset to begin at 1.
- Composite indexes cannot be used to generate multiple independent sequences within a table.

- The MEMORY storage engine does not support `AUTO_INCREMENT` prior to MySQL 4.1.
- If you use `INSERT DELAYED`, the `AUTO_INCREMENT` value is not generated until the record actually is inserted. In this case, `LAST_INSERT_ID()` cannot be relied on to return the sequence value.

`AUTO_INCREMENT` for InnoDB Tables

The InnoDB storage engine has the following `AUTO_INCREMENT` characteristics:

- Before MySQL 5.0.3, the initial sequence value cannot be set with an `AUTO_INCREMENT = n` table option in the `CREATE TABLE` statement, nor can it be modified using that option with `ALTER TABLE`.
- Values that are deleted from the top of the sequence normally are not reused. Exception: As of MySQL 5.0.3, if you empty the table with `TRUNCATE TABLE`, the sequence may be reset to begin at 1. Reuse can occur under the following conditions as well. For an InnoDB table, the first time that you generate a sequence value for an `AUTO_INCREMENT` column, InnoDB uses one greater than the current maximum value in the column (or 1 if the table is empty). InnoDB maintains this counter in memory for use in generating subsequent values; it is not stored in the table itself. This means that if you delete values from the top of the sequence and then restart the server, the deleted values are reused. Restarting the server also cancels the effect of using an `AUTO_INCREMENT` table option in a `CREATE TABLE` or `ALTER TABLE` statement.
- Gaps in a sequence can occur if transactions that generate `AUTO_INCREMENT` values are rolled back.
- Composite indexes cannot be used to generate multiple independent sequences within a table.

`AUTO_INCREMENT` for BDB Tables

The BDB storage engine has the following `AUTO_INCREMENT` characteristics:

- The initial sequence value cannot be set with an `AUTO_INCREMENT = n` table option in the `CREATE TABLE` statement, nor can it be modified using that option with `ALTER TABLE`.
- Values that are deleted from the top of the sequence are reused.
- Composite indexes can be used to generate multiple independent sequences within a table. The `AUTO_INCREMENT` column must be the final column named in the index. Values deleted from the top of each individual sequence are reused, just as for single-column sequences.

Issues to Consider with AUTO_INCREMENT Columns

You should keep the following points in mind to avoid being surprised when you use AUTO_INCREMENT columns:

- Although it is common to use the term "AUTO_INCREMENT column," AUTO_INCREMENT is not a data type; it's a data type attribute. Furthermore, AUTO_INCREMENT is an attribute intended for use only with integer types. Older versions of MySQL are lax in enforcing this constraint and will let you define a data type such as CHAR with the AUTO_INCREMENT attribute. However, only the integer types work *correctly* as AUTO_INCREMENT columns.
- The primary purpose of the AUTO_INCREMENT mechanism is to allow you to generate a sequence of positive integers. The use of non-positive numbers in an AUTO_INCREMENT column is unsupported. Consequently, you may as well define AUTO_INCREMENT columns to be UNSIGNED. Using UNSIGNED also has the advantage of giving you twice as many sequence numbers before you hit the upper end of the data type's range.
- Don't be fooled into thinking that adding AUTO_INCREMENT to a column definition is a magic way of getting an unlimited sequence of numbers. It's not; AUTO_INCREMENT sequences are always bound by the range of the underlying data type. For example, if you use a TINYINT column, the maximum sequence number is 127. When you reach that limit, your application begins to fail with duplicate-key errors. If you use TINYINT UNSIGNED instead, you reach the limit at 255.
- Clearing a table's contents entirely with TRUNCATE TABLE may reset a sequence to begin again at 1, even for storage engines that normally to not reuse AUTO_INCREMENT values. The sequence reset occurs due to the way that MySQL attempts to optimize a complete table erasure operation: When possible, it tosses the data rows and indexes and re-creates the table from scratch rather than deleting rows one at a time. This causes sequence number information to be lost. If you want to delete all records but preserve the sequence information, you can suppress this optimization by using DELETE with a WHERE clause that is always true, to force MySQL to evaluate the condition for each row and thus to delete every row individually:

  ```
  DELETE FROM tbl_name WHERE 1;
  ```

Tips for Working with AUTO_INCREMENT Columns

This section describes some techniques that are useful when working with AUTO_INCREMENT columns.

Adding a Sequence Number Column to a Table

Suppose that you create a table and put some information into it:

```
mysql> CREATE TABLE t (c CHAR(10));
mysql> INSERT INTO t VALUES('a'),('b'),('c');
mysql> SELECT * FROM t;
```

```
+------+
| c    |
+------+
| a    |
| b    |
| c    |
+------+
```

Then you decide that you want to include a sequence number column in the table. To do this, issue an `ALTER TABLE` statement to add an `AUTO_INCREMENT` column, using the same kind of type definition that you'd use with `CREATE TABLE`:

```
mysql> ALTER TABLE t ADD i INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY;
mysql> SELECT * FROM t;
+------+---+
| c    | i |
+------+---+
| a    | 1 |
| b    | 2 |
| c    | 3 |
+------+---+
```

Note how MySQL assigns sequence values to the `AUTO_INCREMENT` column automatically. You need not do so yourself.

Resequencing an Existing Column

If a table already has an `AUTO_INCREMENT` column, but you want to renumber it to eliminate gaps in the sequence that may have resulted from row deletions, the easiest way to do it is to drop the column and then add it again. When MySQL adds the column, it assigns new sequence numbers automatically.

Suppose that a table `t` looks like this, where `i` is the `AUTO_INCREMENT` column:

```
mysql> CREATE TABLE t (c CHAR(10), i INT UNSIGNED AUTO_INCREMENT
    -> NOT NULL PRIMARY KEY);
mysql> INSERT INTO t (c)
    -> VALUES('a'),('b'),('c'),('d'),('e'),('f'),('g'),('h'),('i'),('j'),('k');
mysql> DELETE FROM t WHERE c IN('a','d','f','g','j');
mysql> SELECT * FROM t;
+------+----+
| c    | i  |
+------+----+
| b    |  2 |
| c    |  3 |
| e    |  5 |
| h    |  8 |
| i    |  9 |
| k    | 11 |
+------+----+
```

The following `ALTER TABLE` statement drops the column and then adds it again, renumbering the column in the process:

```
mysql> ALTER TABLE t
    -> DROP i,
    -> ADD i INT UNSIGNED NOT NULL AUTO_INCREMENT,
    -> AUTO_INCREMENT = 1;
mysql> SELECT * FROM t;
+------+---+
| c    | i |
+------+---+
| b    | 1 |
| c    | 2 |
| e    | 3 |
| h    | 4 |
| i    | 5 |
| k    | 6 |
+------+---+
```

The `AUTO_INCREMENT = 1` clause resets the sequence to begin again at 1. For a MyISAM or MEMORY table (or InnoDB table, as of MySQL 5.0.3), you can use a value other than 1 to begin the sequence at a different value. For other storage engines, just omit the `AUTO_INCREMENT` clause, because they do not allow the initial value to be specified this way. The sequence will begin at 1.

Note that although it's easy to resequence a column, and the question, "How do you do it?" is a common one, there is usually very little need to do so. MySQL doesn't care whether a sequence has holes in it, nor do you gain any performance efficiencies by resequencing. In addition, if you have records in another table that refer to the values in the `AUTO_INCREMENT` column, resequencing the column destroys the correspondence between tables.

Generating Sequences Without `AUTO_INCREMENT`

MySQL supports a method for generating sequence numbers that doesn't use an `AUTO_INCREMENT` column at all. Instead, it uses an alternative form of the `LAST_INSERT_ID()` function that takes an argument. If you insert or update a column using `LAST_INSERT_ID(`*`expr`*`)`, the next call to `LAST_INSERT_ID()` with no argument returns the value of *`expr`*. In other words, MySQL treats *`expr`* as though it had been generated as an `AUTO_INCREMENT` value. This allows you to create a sequence number and then retrieve it later in your session, confident that the value will not have been affected by the activity of other clients.

One way to use this strategy is to create a single-row table containing a value that you update each time you want the next value in the sequence. For example, you can create and initialize the table like this:

```
CREATE TABLE seq_table (seq INT UNSIGNED NOT NULL);
INSERT INTO seq_table VALUES(0);
```

Those statements set up `seq_table` with a single row containing a `seq` value of 0. To use the table, generate the next sequence number and retrieve it like this:

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+1);
SELECT LAST_INSERT_ID();
```

The `UPDATE` statement retrieves the current value of the `seq` column and increments it by 1 to produce the next value in the sequence. Generating the new value using `LAST_INSERT_ID(seq+1)` causes it to be treated like an `AUTO_INCREMENT` value, which allows it to be retrieved by calling `LAST_INSERT_ID()` without an argument. `LAST_INSERT_ID()` is client-specific, so you get the correct value even if other clients have generated other sequence numbers in the interval between the `UPDATE` and the `SELECT`.

Other uses for this method are to generate sequence values that increment by a value other than 1, or that are negative. For example, this statement can be executed repeatedly to generate a sequence of numbers that increase by 100 each time:

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+100);
```

Repeating the following statement generates a sequence of decreasing numbers:

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq-1);
```

You also can use this technique to generate a sequence that begins at an arbitrary value, by setting the `seq` column to an appropriate initial value.

The preceding discussion describes how to set up a counter using a table with a single row. That's okay for a single counter. If you want several counters, add another column to the table to server as a counter identifier, and insert a row into the table for each counter. Suppose that you have a Web site and you want to put some "this page has been accessed *n* times" counters in several pages. Create a table with two columns. One column holds a name that uniquely identifies each counter. The other holds the current counter value. You can still use the `LAST_INSERT_ID()` function, but you determine which row it applies to by using the counter name. For example, you can create such a table with the following statement:

```
CREATE TABLE counter
(
    name  VARCHAR(255) CHARACTER SET latin1 COLLATE latin1_bin NOT NULL,
    value INT UNSIGNED,
    PRIMARY KEY (name)
);
```

The `name` column is a string so that you can name a counter whatever you want, and it's defined as a `PRIMARY KEY` to prevent duplicate names. This assumes that applications using the table agree on the names they'll be using. For Web counters, uniqueness of counter names is ensured simply by using the pathname of each page within the document tree as its counter name. The `name` column has a binary collation to cause pathname values to be treated as case sensitive. (If your system has pathnames that are not case sensitive, use a collation that is not case sensitive.)

To use the `counter` table, insert a row corresponding to each page for which you need a counter. For example, to set up a new counter for the site's home page, do this:

```
INSERT INTO counter (name,value) VALUES('index.html',0);
```

That initializes a counter named `'index.html'` with a value of zero. To generate the next sequence value for the page, use its pathname to look up the correct counter value and increment it with `LAST_INSERT_ID(`*`expr`*`)`, and then retrieve the value with `LAST_INSERT_ID()`:

```
UPDATE counter SET value = LAST_INSERT_ID(value+1) WHERE name = 'index.html';
SELECT LAST_INSERT_ID();
```

An alternative approach is to increment the counter without using `LAST_INSERT_ID()`, like this:

```
UPDATE counter SET value = value+1 WHERE name = 'index.html';
SELECT value FROM counter WHERE name = 'index.html';
```

However, that doesn't work correctly if another client increments the counter after you issue the `UPDATE` and before you issue the `SELECT`. You could solve that problem by putting `LOCK TABLES` and `UNLOCK TABLES` around the two statements. Or you could create the table as an InnoDB or BDB table and update the table within a transaction. Either method blocks other clients while you're using the counter, but the `LAST_INSERT_ID()` method accomplishes the same thing more easily. Because its value is client-specific, you always get the value you inserted, not the one from some other client, and you don't have to complicate the code with transactions or locks to keep other clients out.

Choosing Data Types

The section "MySQL Data Types" describes the various data types from which you can choose and the general properties of those types, such as the kind of values they may contain, how much storage space they take, and so on. But how do you actually decide which types to use when you create a table? This section discusses issues to consider that will help you choose.

The most "generic" data types are the string types. You can store anything in them because numbers and dates can be represented in string form. So should you just define all your columns as strings and be done with it? No. Let's consider a simple example. Suppose that you have values that look like numbers. You could represent these as strings, but should you? What happens if you do?

For one thing, you'll probably use more space, because numbers can be stored more efficiently using numeric columns than string columns. You'll also notice some differences in query results due to the different ways that numbers and strings are handled. For example, the sort order for numbers is not the same as for strings. The number 2 is less than the number 11, but the string `'2'` is lexically greater than the string `'11'`. You can work around this by using the column in a numeric context like this:

```
SELECT col_name + 0 as num ... ORDER BY num;
```

Adding zero to the column forces a numeric sort, but is that a reasonable thing to do? It's a useful technique sometimes, but you don't want to have to use it every time you want a numeric sort. Causing MySQL to treat a string column as a number has a couple of significant implications. It forces a string-to-number conversion for each column value, which is inefficient. Also, using the column in a calculation prevents MySQL from using any index on the column, which slows down the query further. Neither of these performance degradations occur if you store the values as numbers in the first place.

The preceding example illustrates that several issues come into play when you choose data types. The simple choice of using one representation rather than another has implications for storage requirements, query handling, and processing performance. The following list gives a quick rundown of factors to think about when picking a type for a column.

- **What kind of values will the column hold?** Numbers? Strings? Dates? Spatial values? This is an obvious question, but you must ask it. You can represent any type of value as a string, but as we've just seen, it's likely that you'll get better performance if you use other more appropriate types for numeric values. (This is also true for temporal and spatial values.) However, assessing the kind of values you're working with isn't necessarily trivial, particularly for other people's data. It's especially important to ask what kind of values the column will hold if you're setting up a table for someone else, and you must be sure to ask enough questions to get sufficient information for making a good decision.
- **Do your values lie within some particular range?** If they are integers, will they always be non-negative? If so, you can use `UNSIGNED`. If they are strings, will they always be chosen from among a fixed, limited set of values? If so, you may find `ENUM` or `SET` a useful type.

  There is a tradeoff between the range of a type and the amount of storage it uses. How "big" a type do you need? For numbers, you can choose small types with a limited range of values, or large types with a much larger range. For strings, you can make them short or long, so you wouldn't choose `CHAR(255)` if all the values you want to store contain fewer than 10 characters.
- **What are the performance and efficiency issues?** Some types can be processed more efficiently than others. Numeric operations generally can be performed more quickly than string operations. Short strings can be compared more quickly than long strings, and also involve less disk overhead. For MyISAM tables, performance is worse for variable-length rows than for fixed-length rows, but fixed-length rows are used only if all columns have fixed-length data types.

Now let's consider each of these issues in more detail. But before we do, allow me to point something out: You want to make the best data type choices you can when you create a table, but it's not the end of the world if you make a choice that turns out to be non-optimal. You can use `ALTER TABLE` to change the type to a better one. This might be as simple as changing a `SMALLINT` to `MEDIUMINT` after finding out your dataset contains values larger than you originally thought. Or it can be more complex, such as changing a `CHAR` to an `ENUM` with a specific set of allowed values. You can use `PROCEDURE ANALYSE()` to obtain information about your table's columns, such as the minimum and maximum values as well as a suggested optimal type to cover the range of values in a column:

```
SELECT * FROM tbl_name PROCEDURE ANALYSE();
```

The output from this query may help you determine that a smaller type can be used, which can improve the performance of queries that involve the table and reduce the amount of space required for table storage.

What Kind of Values Will the Column Hold?

The first thing you think of when you're trying to decide on a data type is the kind of values the column will be used for because this has the most evident implications for the type you choose. In general, you do the obvious thing: You store numbers in numeric columns, strings in string columns, and dates and times in temporal columns. If your numbers have a fractional part, you use a floating-point type rather than an integer type. But sometimes there are exceptions. The principle here is that you need to understand the nature of your data to be able to choose the type in an informed manner. If you're going to store your own data, you probably have a good idea of how to characterize it. On the other hand, if others ask you to set up a table for them, it's sometimes a different story. It may not be so easy to know just what you're working with. Be sure to ask enough questions to find out what kind of values the table really should contain.

Suppose that you're told that a table needs a column to record "amount of precipitation." Is that a number? Or is it "mostly" numeric—that is, typically but not always coded as a number? For example, when you watch the news on television, the weather report generally includes a measure of precipitation. Sometimes this is a number (as in "0.25 inches of rain"), but sometimes it's a "trace" of precipitation, meaning "not much at all." That's fine for the weather report, but what does it mean for storage in a database? You either need to quantify "trace" as a number so that you can use a numeric data type to record precipitation amounts, or you need to use a string so that you can record the word "trace." Or you could come up with some more complicated arrangement, using a number column and a string column where you fill in one column and leave the other one `NULL`. It should be obvious that you want to avoid that option, if possible; it makes the table harder to understand and it makes query-writing much more difficult.

I would probably try to store all rows in numeric form, and then convert them as necessary for display purposes. For example, if any non-zero amount of precipitation less than .01 inches is considered a trace amount, you could display values from the column like this:

```
SELECT IF(precip>0 AND precip<.01,'trace',precip) FROM ... ;
```

Some values are obviously numeric but you must determine whether to use a floating-point or integer type. You should ask what your units are and what accuracy you require. Is whole-unit accuracy sufficient or do you need to represent fractional units? This may help you distinguish between integer and floating-point column types. For example, if you're representing weights, you can use an integer column if you record values to the nearest pound. You'd use a floating-point column if you want to record fractional units. In some cases, you might even use multiple columns—for example, if you want to record weight in terms of pounds and ounces.

Height is a numeric type of information for which there are several representational possibilities:

- Use a string such as `'6-2'` for a value like "6 feet, 2 inches." This has the advantage of having a form that's easy to look at and understand (certainly more so than "74 inches"), but it's difficult to use this kind of value for mathematical operations such as summation or averaging.
- Use one numeric column for feet and another for inches. This would be a little easier to work with for numerical operations, but two columns are more difficult to use than one.
- Use one numeric column representing inches. This is easiest for a database to work with, and least meaningful for humans. But remember that you don't have to present values in the same format that you use to work with them. You can reformat values for meaningful display using MySQL's many functions. That means this might be the best way to represent height.

Another type of numeric information is money. For monetary calculations, you're working with values that have dollars and cents parts. These look like floating-point values, but `FLOAT` and `DOUBLE` are subject to rounding error and may not be suitable except for records in which you need only approximate accuracy. Because people tend to be touchy about their money, it's more likely you need a type that affords perfect accuracy. You have a couple of choices:

- You can represent money as a `DECIMAL(`*`M`*`,2)` type, choosing *`M`* as the maximum width appropriate for the range of values you need. This gives you floating point values with two decimal places of accuracy. The advantage of `DECIMAL` is that values are represented as strings and are not subject to roundoff error. The disadvantage is that string operations are less efficient than operations on values represented internally as numbers. Also, although storage and retrieval will be exact, calculations on `DECIMAL` values might still be done using floating-point operations; roundoff error can occur as a result.
- You can represent all monetary values internally as cents using an integer type. The advantage is that calculations are done internally using integers, which is very fast. The disadvantage is that you will need to convert values on input or output by multiplying or dividing by 100.

Some kinds of "numbers" aren't. Telephone numbers, credit card numbers, and Social Security numbers all contain non-digit characters and cannot be stored directly in a numeric column unless you strip the non-digits.

If you need to store date information, do the values include a time? That is, will they *ever* need to include a time? MySQL doesn't provide a date type that has an optional time part: DATE never has a time, and DATETIME must have a time. If the time really is optional, use a DATE column to record the date, and a separate TIME column to record the time. Then allow the TIME column to be NULL and interpret that as "no time":

```
CREATE TABLE mytbl
(
    date DATE NOT NULL,    # date is required
    time TIME NULL         # time is optional (may be NULL)
);
```

One type of situation in which it's especially important to determine whether you need a time value occurs when you're joining two tables with a master-detail relationship that are "linked" based on date information. Suppose that you're conducting research involving test subjects. Following a standard initial battery of tests, you might run several additional tests, with the choice of tests varying according to the results of the initial tests. You can represent this information using a master-detail relationship, in which the subject identification information and the standard initial tests are stored in a master record and any additional tests are stored as rows in a secondary detail table. Then you link together the two tables based on subject ID and the date on which the tests are given.

The question you must answer in this situation is whether you can use just the date or whether you need both date and time. This depends on whether a subject might go through the testing procedure more than once during the same day. If so, record the time (for example, the time that the procedure begins), using either a DATETIME column or separate DATE and TIME columns that both must be filled in. Without the time value, you will not be able to associate a subject's detail records with the proper master records if the subject is tested twice in a day.

I've heard people claim "I don't need a time; I will never test a subject twice on the same day." Sometimes they're correct, but I have also seen some of these same people turn up later wondering how to prevent detail records from being mixed up with the wrong master record after entering data for subjects who were tested multiple times in a day. Sorry, by then it's too late!

Sometimes you can deal with this problem by retrofitting a TIME column into the tables. Unfortunately, it's difficult to fix existing records unless you have some independent data source, such as the original paper records. Otherwise, you have no way to disambiguate detail records to associate them to the proper master record. Even if you have an independent source of information, this is very messy and likely to cause problems for applications that you've already written to use the tables. It's best to explain the issues to the table owners and make sure that you've gotten a good characterization of the data values before creating their tables.

Sometimes you have incomplete data, and this will influence your choice of data types. You may be collecting birth and death dates for genealogical research, and sometimes all you can find out is the year or year and month someone was born or died, but not the exact date. If you use a DATE column, you can't enter a date unless you have the full date. If you want to be able to record whatever information you have, even if it's incomplete, you may have to keep separate year, month, and day columns. Then you can enter such parts of the date as you have and leave the rest NULL. Another possibility is to use DATE values in which the day or month and day parts are set to 0. Such "fuzzy" dates can be used to represent incomplete date values.

Do Your Values Lie Within Some Particular Range?

If you've decided on the general category from which to pick a data type for a column, thinking about the range of values you want to represent will help you narrow down your choices to a particular type within that category. Suppose that you want to store integer values. The range of your values determines the types you can use. If you need values in the range from 0 to 1000, you can use anything from a SMALLINT up to a BIGINT. If your values range up to 2 million, you can't use SMALLINT, so your choices range from MEDIUMINT to BIGINT.

You could, of course, simply use the largest type for the kind of value you want to store (BIGINT for the examples in the previous paragraph). Generally, however, you should use the smallest type that is large enough for your purposes. By doing so, you'll minimize the amount of storage used by your tables, and they will give you better performance because smaller columns usually can be processed more quickly than larger ones. (Reading smaller values requires less disk activity, and more key values fit into the key cache, allowing indexed searches to be performed faster.)

If you don't know the range of values you'll need to be able to represent, you either must guess or use BIGINT to accommodate the worst possible case. If you guess and the type you choose turns out later to be too small, all is not lost. Use ALTER TABLE later to make the column bigger.

Sometimes you even find out that you can make a column smaller. In Chapter 1, we created a score table for the grade-keeping project that had a score column for recording quiz and test scores. The column was created using INT in order to keep the discussion simpler, but you can see now that if scores are in the range from 0 to 100, a better choice would be TINYINT UNSIGNED, because that would use less storage.

The range of values in your data also affects the attributes you can use with your data type. If values never are negative, you can use UNSIGNED; otherwise, you can't.

String types don't have a "range" in the same way numeric columns do, but they have a length, and the maximum length you need affects the column types you can use. If you're storing character strings that are shorter than 256 characters, you can use CHAR, VARCHAR, or TINYTEXT. If you want longer strings, you can use a longer TEXT type, or VARCHAR as of MySQL 5.0.3.

For a string column used to represent a fixed set of values, you might consider using an ENUM or SET data type. These can be good choices because they are represented internally as numbers. Operations on them are performed numerically, which makes them more efficient than other string types. They also can be more compact than other string types, which saves space. In addition, for MySQL 5.0.2 and up, you can prevent entry of values not present in the list of legal values by enabling strict SQL mode. See "How MySQL Handles Invalid Data Values."

When characterizing the range of values you have to deal with, the best terms are "always" and "never" (as in "always less than 1000" or "never negative"), because they allow you to constrain your data type choices more tightly. But be wary of using these terms when they're not really justified. Be especially wary if you're consulting with other people about their data and they start throwing around those two terms. When people say "always" or "never," be sure they really mean it. Sometimes people say their data always have a particular characteristic when they really mean "almost always."

Suppose that you're designing a table for a group of investigators who tell you, "Our test scores are always 0 to 100." Based on that statement, you choose `TINYINT` and you make it `UNSIGNED` because the values are always non-negative. Then you find out that the people who code the data for entry into the database sometimes use –1 to mean "student was absent due to illness." Oops. They didn't tell you that. It might be acceptable to use `NULL` to represent such values, but if not, you'll have to record a –1, and then you can't use an `UNSIGNED` column. (This is an instance where `ALTER TABLE` comes to your rescue.)

Sometimes decisions about these cases can be made more easily by asking a simple question: Are there ever exceptions? If an exceptional case ever occurs, even just once, you must allow for it. You will find that people who talk to you about designing a database invariably think that if exceptions don't occur very often, they don't matter. When you're creating a table, you can't think that way. The question you need to ask isn't "how often do exceptions occur?" It's "do exceptions *ever* occur?" If they do, you must take them into account.

What Are the Performance and Efficiency Issues?

Your choice of data type can influence query performance in several ways. If you keep the general guidelines discussed in the following sections in mind, you'll be able to choose types that will help MySQL process your tables more efficiently.

Numeric Versus String Operations

Numeric operations are generally faster than string operations. Consider comparison operations. Numbers can be compared in a single operation. String comparisons may involve several byte-by-byte or character-by-character comparisons, more so as the strings become longer.

If a string column has a limited number of values, you can use an `ENUM` or `SET` type to get the advantages of numeric operations. These types are represented internally as numbers and can be processed more efficiently.

Consider alternative representations for strings. Sometimes you can improve performance by representing string values as numbers. For example, to represent IP numbers in dotted-quad notation, such as 192.168.0.4, you might use a string. As an alternative, you could convert the IP numbers to integer form by storing each part of the dotted-quad form in one byte of a four-byte `INT UNSIGNED` type. Storing integers would both save space and speed lookups. On the other hand, representing IP numbers as `INT` values might make it difficult to perform pattern matches such as you might do if you wanted to look for numbers in a given subnet. Perhaps you can do the same thing by using bitmask operations. These kinds of issues illustrate that you cannot consider only space issues; you must decide which representation is most appropriate based on what you want to do with the values. (Whatever choice you make, the `INET_ATON()` and `INET_NTOA()` functions can help convert between the two representations.)

Smaller Types Versus Larger Types

Smaller types can be processed more quickly than larger types. A general principle is that they take less space and involve less overhead for disk activity. For strings in particular, processing time is in direct relationship to string length.

For columns that use fixed-size data types, choose the smallest type that will hold the required range of values. For example, don't use `BIGINT` if `MEDIUMINT` will do. Don't use `DOUBLE` if you need only `FLOAT` precision. For variable-size types, you may still be able to save space. A `BLOB` uses 2 bytes to record the length of the value, a `LONGBLOB` uses 4 bytes. If you're storing values that are never as long as 64KB, using `BLOB` saves you 2 bytes per value. (Similar considerations apply for `TEXT` types.)

Fixed-Length Versus Variable-Length Types

Fixed-length and variable-length types have different performance implications, although the particular effects of each varies per storage engine. For example, MyISAM tables that have fixed-length rows generally can be processed more quickly than tables with variable-length rows. For further discussion of this issue, see "Data Type Choices and Query Efficiency," in Chapter 4.

Prohibiting or Allowing `NULL` Values

If you define a column to be `NOT NULL`, it can be handled more quickly because MySQL doesn't have to check the column's values during query processing to see whether they are `NULL`. It also saves one bit per row in the table. Avoiding `NULL` in columns may make your queries simpler because you don't have to think about `NULL` as a special case, and simpler queries generally can be processed more quickly.

Inter-Relatedness of Data Type Choice Issues

You can't always consider the issues involved in choosing data types as though they are independent of one another. For example, range is related to storage size for numeric types: As you increase the range, you require more storage, which affects performance. Or consider the implications of using `AUTO_INCREMENT` to create a column for holding unique sequence numbers. That single choice has several consequences involving the data type, indexing, and the use of `NULL`:

- `AUTO_INCREMENT` is a column attribute that should be used only with integer types. That immediately limits your choices to `TINYINT` through `BIGINT`.
- An `AUTO_INCREMENT` column is intended only for generating sequences of positive values, so you should define it as `UNSIGNED`.

- `AUTO_INCREMENT` columns must be indexed. Furthermore, to prevent duplicates in the column, the index must be unique. This means you must define the column as a `PRIMARY KEY` or as a `UNIQUE` index.
- `AUTO_INCREMENT` columns must be `NOT NULL`.

All of this means you do not just define an `AUTO_INCREMENT` column like this:

```
mycol arbitrary_type AUTO_INCREMENT
```

You define it like this:

```
mycol integer_type UNSIGNED NOT NULL AUTO_INCREMENT,
PRIMARY KEY (mycol)
```

Or like this:

```
mycol integer_type UNSIGNED NOT NULL AUTO_INCREMENT,
UNIQUE (mycol)
```

Expression Evaluation and Type Conversion

Expressions contain terms and operators and are evaluated to produce values. Expressions can include constants, function calls, and references to table columns. These values may be combined using different kinds of operators, such as arithmetic or comparison operators, and terms of an expression may be grouped with parentheses. Expressions occur most commonly in the output column list and `WHERE` clause of `SELECT` statements. For example, here is a query that is similar to one used for age calculations in Chapter 1:

```
SELECT
    CONCAT(last_name, ', ', first_name),
    (YEAR(death) - YEAR(birth)) - IF(RIGHT(death,5) < RIGHT(birth,5),1,0)
FROM president
WHERE
    birth > '1900-1-1' AND DEATH IS NOT NULL;
```

Each selected value represents an expression, as does the content of the `WHERE` clause. Expressions also occur in the `WHERE` clause of `DELETE` and `UPDATE` statements, the `VALUES()` clause of `INSERT` statements, and so forth.

When MySQL encounters an expression, it evaluates the expression to produce a result. For example, `(4*3)/(4-2)` evaluates to the value `6`. Expression evaluation may involve type conversion, such as when MySQL converts the number `960821` into a date `'1996-08-21'` if the number is used in a context requiring a `DATE` value.

This section discusses how you can write expressions in MySQL and the rules that govern the various kinds of type conversions that MySQL performs during the process of expression evaluation. Each of MySQL's operators is listed here, but MySQL has so many functions that only a few are touched on. For more information, see Appendix C.

Writing Expressions

An expression can be as simple as a single constant:

`0`	Numeric constant
`'abc'`	String constant
`'2007-11-19'`	Date constant

Expressions can use function calls. Some functions take arguments (values inside the parentheses), and some do not. Multiple arguments should be separated by commas. When you invoke a function, there can be spaces around arguments, but there must be no space between the function name and the opening parenthesis:

`NOW()`	Function with no arguments
`STRCMP('abc','def')`	Function with two arguments
`STRCMP( 'abc', 'def' )`	Spaces around arguments are legal
`STRCMP ('abc','def')`	Space after function name is illegal

If there is a space after the function name, the MySQL parser may interpret the function name as a column name. The usual result is a syntax error. This happens because function names are not reserved words and you can use them for column names if you want. You can tell MySQL to allow spaces after function names by enabling the `IGNORE_SPACE` SQL mode. However, that also causes function names to be treated as reserved words.

Expressions can include references to table columns. In the simplest case, when the table to which a column belongs is clear from context, a column reference may be given simply as the column name. Only one table is named in each of the following `SELECT` statements, so the column references are unambiguous, even though the same column names are used in each statement:

```
SELECT last_name, first_name FROM president;
SELECT last_name, first_name FROM member;
```

If it's not clear which table should be used, a column name can be qualified by preceding it with the proper table name. If it's not even clear which database should be used, the table name can be preceded by the database name. You also can use these more-specific qualified forms in unambiguous contexts if you simply want to be more explicit:

```
SELECT
    president.last_name, president.first_name,
    member.last_name, member.first_name
FROM president, member
WHERE president.last_name = member.last_name;

SELECT sampdb.student.name FROM sampdb.student;
```

Finally, you can combine all these kinds of values (constants, function calls, and column references) to form more complex expressions.

Operator Types

Terms of expressions can be combined using several kinds of operators. Arithmetic operators, listed in Table 3.18, include the usual addition, subtraction, multiplication, and division operators, as well as the modulo operator. Arithmetic is performed using `BIGINT` (64-bit) integer values for `+`, `-`, and `*` when both operands are integers, as well as for `/`, `DIV`, and `%` when the operation is performed in a context where the result is expected to be an integer. Otherwise, `DOUBLE` is used. Be aware that if an integer operation involves large values such that the result exceeds 64-bit range, you will get unpredictable results. (Actually, you should try to avoid exceeding 63-bit values; one bit is needed to represent the sign.)

Table 3.18 **Arithmetic Operators**

Operator	**Syntax**	**Meaning**
`+`	`a + b`	Addition; sum of operands
`-`	`a - b`	Subtraction; difference of operands
`-`	`-a`	Unary minus; negation of operand
`*`	`a * b`	Multiplication; product of operands
`/`	`a / b`	Division; quotient of operands
`DIV`	`a DIV b`	Division; integer quotient of operands
`%`	`a % b`	Modulo; remainder after division of operands

Logical operators, shown in Table 3.19, evaluate expressions to determine whether they are true (non-zero) or false (zero). It is also possible for a logical expression to evaluate to `NULL` if its value cannot be ascertained. For example, `1 AND NULL` is of indeterminate value.

Table 3.19 **Logical Operators**

Operator	**Syntax**	**Meaning**
`AND`, `&&`	`a AND b`, `a && b`	Logical intersection; true if both operands are true
`OR`, `||`	`a OR b`, `a || b`	Logical union; true if either operand is true
`XOR`	`a XOR b`	Logical exclusive-OR; true if exactly one operand is true
`NOT`, `!`	`NOT a`, `!a`	Logical negation; true if operand is false

As alternative forms of `AND`, `OR`, and `NOT`, MySQL allows the `&&`, `||`, and `!` operators, respectively, as used in the C programming language. Note in particular the `||` operator. Standard SQL specifies `||` as the string concatenation operator, but in MySQL it signifies a logical `OR` operation. If you use the following expression, expecting it to perform string concatenation, you may be surprised to discover that it returns the number 0:

```
'abc' || 'def'                                    → 0
```

This happens because `'abc'` and `'def'` are converted to integers for the operation, and both turn into 0. In MySQL, you must use `CONCAT('abc','def')` or proximity to perform string concatenation:

```
CONCAT('abc','def')                                → 'abcdef'
'abc' 'def'                                        → 'abcdef'
```

If you want the standard SQL behavior for `||`, enable the `PIPES_AS_CONCAT` SQL mode.

Bit operators, shown in Table 3.20, perform bitwise intersection, union, and exclusive-OR, where each bit of the result is evaluated as the logical AND, OR, or exclusive-OR of the corresponding bits of the operands. You also can perform bit shifts left or right. Bit operations are performed using `BIGINT` (64-bit) integer values.

Table 3.20 **Bit Operators**

Operator	**Syntax**	**Meaning**
`&`	`a & b`	Bitwise AND (intersection); each bit of result is set if corresponding bits of both operands are set
`|`	`a | b`	Bitwise OR (union); each bit of result is set if corresponding bit of either operand is set
`^`	`a ^ b`	Bitwise exclusive-OR; each bit of result is set only if exactly one corresponding bit of the operands is set
`<<`	`a << b`	Left shift of `a` by `b` bit positions
`>>`	`a >> b`	Right shift of `a` by `b` bit positions

Comparison operators, shown in Table 3.21, include operators for testing relative magnitude or lexical ordering of numbers and strings, as well as operators for performing pattern matching and for testing `NULL` values. The `<=>` operator is MySQL-specific.

Table 3.21 **Comparison Operators**

Operator	**Syntax**	**Meaning**
`=`	`a = b`	True if operands are equal
`<=>`	`a <=> b`	True if operands are equal (even if `NULL`)
`<>, !=`	`a <> b, a != b`	True if operands are not equal
`<`	`a < b`	True if `a` is less than `b`
`<=`	`a <= b`	True if `a` is less than or equal to `b`
`>=`	`a >= b`	True if `a` is greater than or equal to `b`
`>`	`a > b`	True if `a` is greater than `b`
`IN`	`a IN (b1, b2, ...)`	True if `a` is equal to any of `b1`, `b2`, ...
`BETWEEN`	`a BETWEEN b AND C`	True if `a` is between the values of `b` and `c`, inclusive
`NOT BETWEEN`	`a NOT BETWEEN b AND C`	True if `a` is not between the values of `b` and `c`, inclusive
`LIKE`	`a LIKE b`	SQL pattern match; true if `a` matches `b`

Table 3.21 **Continued**

Operator	Syntax	Meaning
`NOT LIKE`	`a NOT LIKE b`	SQL pattern match; true if `a` does not match `b`
`REGEXP`	`a REGEXP b`	Regular expression match; true if `a` matches `b`
`NOT REGEXP`	`a NOT REGEXP b`	Regular expression match; true if `a` does not match `b`
`IS NULL`	`a IS NULL`	True if operand is `NULL`
`IS NOT NULL`	`a IS NOT NULL`	True if operand is not `NULL`

For a discussion of the comparison properties of string values, see "String Values."

Pattern matching allows you to look for values without having to specify an exact literal value. MySQL provides SQL pattern matching using the `LIKE` operator and the wildcard characters '`%`' (match any sequence of characters) and '`_`' (match any single character). MySQL also provides pattern matching based on the `REGEXP` operator and regular expressions that are similar to those used in Unix programs such as `grep`, `sed`, and `vi`. You must use one of these pattern-matching operators to perform a pattern match; you cannot use the `=` operator. To reverse the sense of a pattern match, use `NOT LIKE` or `NOT REGEXP`.

The two types of pattern matching differ in important respects besides the use of different operators and pattern characters:

- `LIKE` is multi-byte safe. `REGEXP` works correctly only for single-byte character sets.
- `LIKE` SQL patterns match only if the entire string is matched. `REGEXP` regular expressions match if the pattern is found anywhere in the string.

Patterns used with the `LIKE` operator may include the '`%`' and '`_`' wildcard characters. For example, the pattern `'Frank%'` matches any string that begins with `'Frank'`:

```
'Franklin' LIKE 'Frank%'                          → 1
'Frankfurter' LIKE 'Frank%'                       → 1
```

The wildcard character '`%`' matches any sequence of characters, including the empty sequence, so `'Frank%'` matches `'Frank'`:

```
'Frank' LIKE 'Frank%'                             → 1
```

This also means the pattern `'%'` matches any string, including the empty string. However, `'%'` will not match `NULL`. In fact, any pattern match with a `NULL` operand fails:

```
'Frank' LIKE NULL                                 → NULL
NULL LIKE '%'                                     → NULL
```

MySQL's `LIKE` operator is not case sensitive unless one of its operands is a binary string or a non-binary string with a case-sensitive or binary collation:

```
'Frankly' LIKE 'Frank%'                              → 1
'frankly' LIKE 'Frank%'                              → 1
BINARY 'Frankly' LIKE 'Frank%'                       → 1
BINARY 'frankly' LIKE 'Frank%'                       → 0
'Frankly' COLLATE latin1_general_cs LIKE 'Frank%'    → 1
'frankly' COLLATE latin1_general_cs LIKE 'Frank%'    → 0
'Frankly' COLLATE latin1_bin LIKE 'Frank%'           → 1
'frankly' COLLATE latin1_bin LIKE 'Frank%'           → 0
```

This behavior differs from that of the standard SQL `LIKE` operator, which is case sensitive.

The other wildcard character allowed with `LIKE` is '`_`', which matches any single character. The pattern `'___'` matches any string of exactly three characters. `'c_t'` matches `'cat'`, `'cot'`, `'cut'`, and even `'c_t'` (because '`_`' matches itself).

Wildcard characters may be specified anywhere in a pattern. `'%bert'` matches `'Englebert'`, `'Bert'`, and `'Albert'`. `'%bert%'` matches all of those strings, and also strings like `'Berthold'`, `'Bertram'`, and `'Alberta'`. `'b%t'` matches `'Bert'`, `'bent'`, and `'burnt'`.

To match literal instances of the '`%`' or '`_`' characters, turn off their special meaning by preceding them with a backslash ('`\%`' or '`\_`'):

```
'abc' LIKE 'a%c'                                     → 1
'abc' LIKE 'a\%c'                                    → 0
'a%c' LIKE 'a\%c'                                    → 1
'abc' LIKE 'a_c'                                     → 1
'abc' LIKE 'a\_c'                                    → 0
'a_c' LIKE 'a\_c'                                    → 1
```

MySQL's other form of pattern matching uses regular expressions. The operator is `REGEXP` rather than `LIKE`. The most common regular expression pattern characters are as follows:

The '`.`' character is a wildcard that matches any single character:

```
'abc' REGEXP 'a.c'                                   → 1
```

The `[...]` construction matches any character listed between the square brackets.

```
'e' REGEXP '[aeiou]'                                 → 1
'f' REGEXP '[aeiou]'                                 → 0
```

You can specify a range of characters by listing the endpoints of the range separated by a dash ('`-`'), or negate the sense of the class (to match any character not listed) by specifying '`^`' as the first character of the class:

```
'abc' REGEXP '[a-z]'                                 → 1
'abc' REGEXP '[^a-z]'                                → 0
```

'`*`' means "match any number of the previous thing," so that, for example, the pattern `'x*'` matches any number of '`x`' characters:

```
'abcdef' REGEXP 'a.*f'                          → 1
'abc' REGEXP '[0-9]*abc'                        → 1
'abc' REGEXP '[0-9][0-9]*'                      → 0
```

"Any number" includes zero instances, which is why the second expression succeeds. To match one or more instances of the preceding thing rather than zero or more, use '+' instead of '*':

```
'abc' REGEXP 'cd*'                              → 1
'abc' REGEXP 'cd+'                              → 0
'abcd' REGEXP 'cd+'                             → 1
```

`'^pattern'` and `'pattern$'` anchor a pattern match so that the pattern *pattern* matches only when it occurs at the beginning or end of a string, and `'^pattern$'` matches only if *pattern* matches the entire string:

```
'abc' REGEXP 'b'                                → 1
'abc' REGEXP '^b'                               → 0
'abc' REGEXP 'b$'                               → 0
'abc' REGEXP '^abc$'                            → 1
'abcd' REGEXP '^abc$'                           → 0
```

MySQL's regular expression matching has other special pattern elements as well. See Appendix C for more information.

A LIKE or REGEXP pattern can be taken from a table column, although this will be slower than a constant pattern if the column contains several different values. The pattern must be examined and converted to internal form each time the column value changes.

Operator Precedence

When MySQL evaluates an expression, it looks at the operators to determine the order in which it should group the terms of the expression. Some operators have higher precedence; that is, they are "stronger" than others in the sense that they are evaluated earlier than others. For example, multiplication and division have higher precedence than addition and subtraction. The following two expressions are equivalent because * and / are evaluated before + and -:

```
1 + 2 * 3 - 4 / 5                               → 6.2
1 + 6 - .8                                      → 6.2
```

Operator precedence is shown in the following list, from highest precedence to lowest. Operators listed on the same line have the same precedence. Operators at a higher precedence level are evaluated before operators at a lower precedence level. Operators at the same precedence level are evaluated left to right.

```
BINARY  COLLATE
!
- (unary minus)  ~ (unary bit negation)
^
```

```
*  /  DIV  %  MOD
+  -
<<  >>
&
|
<  <=  =  <=>  <>  !=  >=  >  IN  IS  LIKE  REGEXP  RLIKE
BETWEEN  CASE  WHEN  THEN  ELSE
NOT
AND  &&
OR  ||  XOR
:=
```

Some operators have a different precedence depending on the SQL mode or MySQL version. See Appendix C for details.

If you need to override the precedence of operators and change the order in which expression terms are evaluated, use parentheses to group terms:

```
1 + 2 * 3 - 4 / 5                                  → 6.2
(1 + 2) * (3 - 4) / 5                              → -0.6
```

NULL Values in Expressions

Take care when using NULL values in expressions, because the result may not always be what you expect. The following guidelines will help you avoid surprises.

If you supply NULL as an operand to any arithmetic or bit operator, the result is NULL:

```
1 + NULL                                           → NULL
1 | NULL                                           → NULL
```

With logical operators, the result is NULL unless the result can be determined with certainty:

```
1 AND NULL                                         → NULL
1 OR NULL                                          → 1
0 AND NULL                                         → 0
0 OR NULL                                          → NULL
```

NULL as an operand to any comparison or pattern-matching operator produces a NULL result, except for the <=>, IS NULL, and IS NOT NULL operators, which are intended specifically for dealing with NULL values:

```
1 = NULL                                           → NULL
NULL = NULL                                        → NULL
1 <=> NULL                                         → 0
NULL LIKE '%'                                      → NULL
NULL REGEXP '.*'                                   → NULL
NULL <=> NULL                                      → 1
1 IS NULL                                          → 0
NULL IS NULL                                       → 1
```

Functions generally return NULL if given NULL arguments, except for those functions designed to deal with NULL arguments. For example, IFNULL() is able to handle NULL arguments and returns true or false appropriately. On the other hand, STRCMP() expects non-NULL arguments; if you pass it a NULL argument, it returns NULL rather than true or false.

In sorting operations, NULL values group together. They appear first in ascending sorts and last in descending sorts.

Type Conversion

Whenever a value of one type is used in a context that requires a value of another type, MySQL performs extensive type conversion automatically according to the kind of operation you're performing. Type conversion may occur for any of the following reasons:

- Conversion of operands to a type appropriate for evaluation of an operator
- Conversion of a function argument to a type expected by the function
- Conversion of a value for assignment into a table column that has a different type

You also can perform explicit type conversion using a cast operator or function.

The following expression involves implicit type conversion. It consists of the addition operator + and two operands, 1 and '2':

```
1 + '2'
```

The operands are of different types (number and string), so MySQL converts one of them to make them the same type. But which one should it change? In this case, + is a numeric operator, so MySQL wants the operands to be numbers thus and converts the string '2' to the number 2. Then it evaluates the expression to produce the result 3.

```
1 + '2'                                          → 3
```

Here's another example. The CONCAT() function concatenates strings to produce a longer string as a result. To do this, it interprets its arguments as strings, no matter what type they are. If you pass it a bunch of numbers, CONCAT() converts them to strings, and then returns their concatenation:

```
CONCAT(1,23,456)                                 → '123456'
```

If the call to CONCAT() is part of a larger expression, further type conversion may take place. Consider the following expression and its result:

```
REPEAT('X',CONCAT(1,2,3)/10)                     → 'XXXXXXXXXXXX'
```

CONCAT(1,2,3) produces the string '123'. The expression '123'/10 is converted to 123/10 because division is an arithmetic operator. The result of this expression would be 12.3 in floating-point context, but REPEAT() expects an integer repeat count, so an integer division is performed to produce 12. Then REPEAT('X',12) produces a string result of 12 'x' characters.

A general principle to keep in mind is that, by default, MySQL attempts to convert values to the type required by an expression rather than generating an error. Depending on the context, it converts values of each of the three general categories (numbers, strings, or dates and times) to values in any of the other categories. However, values can't always be converted from one type to another. If a value to be converted to a given type doesn't look like a legal value for that type, the conversion fails. Conversion to numbers of things like `'abc'` that don't look like numbers results in a value of `0`. Conversion to date or time types of things that don't look like a date or time result in the "zero" value for the type. For example, converting the string `'abc'` to a date results in the "zero" date `'0000-00-00'`. On the other hand, any value can be treated as a string, so generally it's not a problem to convert a value to a string.

If you want to prevent conversion of illegal values to the closest legal values during data input operations, you can enable strict mode to cause an error to occur instead. See "How MySQL Handles Invalid Data Values."

MySQL also performs more minor type conversions. If you use a floating-point value in an integer context, the value is converted (with rounding). Conversion in the other direction works as well; an integer can be used without problem as a floating-point number.

Hexadecimal constants are treated as binary strings unless the context clearly indicates a number. In string contexts, each pair of hexadecimal digits is converted to a character and the result is used as a string. The following examples illustrate how this works:

```
0x61                                      → 'a'
0x61 + 0                                  → 97
X'61'                                     → 'a'
X'61' + 0                                 → 97
CONCAT(0x61)                              → 'a'
CONCAT(0x61 + 0)                          → '97'
CONCAT(X'61')                             → 'a'
CONCAT(X'61' + 0)                         → '97'
```

For comparisons, context determines whether to treat a hexadecimal constant as a binary string or a number:

- This expression treats the operands as binary strings and performs a byte-by-byte comparison.

  ```
  0x0d0a = '\r\n'                         → 1
  ```

- This expression compares a hexadecimal constant to a number, so it is converted to a number for the comparison.

  ```
  0x0a = 10                               → 1
  ```

- This expression performs a binary string comparison. The first byte of the left operand has a lesser byte value than the first byte of the right operand, so the result is false.

  ```
  0xee00 > 0xff                           → 0
  ```

- In this expression, the right operand hex constant is converted to a number because of the arithmetic operator. Then for the comparison, the left operand is converted to a number. The result is false because `0xee00` (60928) is not numerically less than `0xff` (255).

  ```
  0xee00 > 0xff+0                            → 1
  ```

It's possible to force a hexadecimal constant to be treated as a non-binary string by using a character set introducer or `CONVERT()`:

```
0x61                                    → 'a'
0x61 = 'A'                              → 0
_latin1 0x61 = 'A'                      → 1
CONVERT(0x61 USING latin1) = 'A'        → 1
```

Some operators force conversion of the operands to the type expected by the operator, no matter what the type of the operands is. Arithmetic operators are an example of this. They expect numbers, and the operands are converted accordingly:

```
3 + 4                                   → 7
'3' + 4                                 → 7
'3' + '4'                               → 7
```

In a string-to-number conversion, it's not enough for a string simply to contain a number somewhere. MySQL doesn't look through the entire string hoping to find a number, it looks only at the beginning; if the string has no leading numeric part, the conversion result is 0.

```
'1973-2-4' + 0                          → 1973
'12:14:01' + 0                          → 12
'23-skidoo' + 0                         → 23
'-23-skidoo' + 0                        → -23
'carbon-14' + 0                         → 0
```

MySQL's string-to-number conversion rule converts numeric-looking strings to floating-point values:

```
'-428.9' + 0                            → -428.9
'3E-4' + 0                              → 0.0003
```

This does not work for hexadecimal constants, though:

```
'0xff'                                  → '0xff'
```

The logical and bit operators are even stricter than the arithmetic operators. They want the operators to be not just numeric, but integers, and type conversion is performed accordingly. This means that a floating-point number such as `0.3` is not considered true, even though it's non-zero; that's because when it's converted to an integer, the result is `0`. In the following expressions, the operands are not considered true until they have a value of at least 1.

```
0.3 OR .04                                       → 0
1.3 OR .04                                       → 1
0.3 AND .04                                      → 0
1.3 AND .04                                      → 0
1.3 AND 1.04                                     → 1
```

This type of conversion also occurs with the `IF()` function, which expects the first argument to be an integer. This means that values that round to zero will be considered false:

```
IF(1.3,'non-zero','zero')                        → 'non-zero'
IF(0.3,'non-zero','zero')                        → 'zero'
IF(-0.3,'non-zero','zero')                       → 'zero'
IF(-1.3,'non-zero','zero')                       → 'non-zero'
```

To test floating-point values properly, it's best to use an explicit comparison:

```
IF(0.3>0,'non-zero','zero')                      → 'non-zero'
```

Pattern matching operators expect to operate on strings. This means you can use MySQL's pattern matching operators on numbers because it will convert them to strings in the attempt to find a match!

```
12345 LIKE '1%'                                  → 1
12345 REGEXP '1.*5'                              → 1
```

The magnitude comparison operators (`<`, `<=`, `=`, and so on) are context sensitive; that is, they are evaluated according to the types of their operands. The following expression compares the operands numerically because they both are numbers:

```
2 < 11                                           → 1
```

This expression involves string operands and thus results in a lexical comparison:

```
'2' < '11'                                       → 0
```

In the following comparisons, the types are mixed, so MySQL compares them as numbers. As a result, both expressions are true:

```
'2' < 11                                         → 1
2 < '11'                                         → 1
```

When evaluating comparisons, MySQL converts operands as necessary according to the following rules:

- Other than for the `<=>` operator, comparisons involving `NULL` values evaluate as `NULL`. (`<=>` is like `=`, except that `NULL <=> NULL` is true.)
- If both operands are strings, they are compared lexically as strings. Binary strings are compared on a byte-by-byte basis using the numeric value of each byte. Comparisons for non-binary strings are performed character-by-character using the collating sequence of the character set in which the strings are expressed. If the strings have different character sets, the comparison may result in an error or fail to yield meaningful results. A comparison between a binary and a non-binary string is treated as a comparison of binary strings.

- If both operands are integers, they are compared numerically as integers.
- As of MySQL 4.1.1, hexadecimal constants that are not compared to a number are compared as binary strings. (This differs from MySQL 4.0, which compares hexadecimal constants as numbers by default.)
- If either operand is a `TIMESTAMP` or `DATETIME` value and the other is a constant, the operands are compared as `TIMESTAMP` values. This is done to make comparisons work better for ODBC applications.
- Otherwise, the operands are compared numerically as floating-point values. Note that this includes the case of comparing a string and a number. The string is converted to a number, which results in a value of `0` if the string doesn't look like a number. For example, `'14.3'` converts to `14.3`, but `'L4.3'` converts to `0`.

Date and Time Interpretation Rules

MySQL freely converts strings and numbers to date and time values as demanded by context in an expression, and vice versa. Date and time values are converted to numbers in numeric context; numbers are converted to dates or times in date or time contexts. This conversion to a date or time value happens when you assign a value to a date or time column or when a function requires a date or time value. In comparisons, the general rule is that date and time values are compared as strings.

If the table `mytbl` contains a `DATE` column `date_col`, the following statements are equivalent:

```
INSERT INTO mytbl SET date_col = '2004-04-13';
INSERT INTO mytbl SET date_col = '20040413';
INSERT INTO mytbl SET date_col = 20040413;
```

In the following examples, the argument to the `TO_DAYS()` function is interpreted as the same value for all three expressions:

```
TO_DAYS('2004-04-10')                         → 732046
TO_DAYS('20040410')                           → 732046
TO_DAYS(20040410)                             → 732046
```

Testing and Forcing Type Conversion

To see how type conversion will be handled in an expression, issue a `SELECT` query that evaluates the expression so that you can examine the result:

```
mysql> SELECT 0x41, 0x41 + 0;
+------+----------+
| 0x41 | 0x41 + 0 |
+------+----------+
| A    |       65 |
+------+----------+
```

As you might imagine, I did quite a lot of that sort of testing with different versions of MySQL while writing this chapter.

Testing expression evaluation is especially important for statements such as `DELETE` or `UPDATE` that modify records, because you want to be sure you're affecting only the intended rows. One way to check an expression is to run a preliminary `SELECT` statement with the same `WHERE` clause that you're going to use with the `DELETE` or `UPDATE` statement to verify that the clause selects the proper rows. Suppose that the table `mytbl` has a `CHAR` column `char_col` containing these values:

```
'abc'
'def'
'00'
'ghi'
'jkl'
'00'
'mno'
```

Given these values, what is the effect of the following statement?

```
DELETE FROM mytbl WHERE char_col = 00;
```

The intended effect is probably to delete the two rows containing the value `'00'`. The actual effect would be to delete all the rows—an unpleasant surprise. This happens as a consequence of MySQL's comparison rules. `char_col` is a string column, but `00` in the statement is not quoted, so it is treated as a number. By MySQL's comparison rules, a comparison involving a string and a number evaluates as a comparison of two numbers. As MySQL executes the `DELETE` statement, it converts each value of `char_col` to a number and compares it to `0`. Unfortunately, although `'00'` converts to `0`, so do all the strings that don't look like numbers. As a result, the `WHERE` clause is true for every row, and the `DELETE` statement empties the table. This is a case where it would have been prudent to test the `WHERE` clause with a `SELECT` statement prior to executing the `DELETE`, because that would have shown you that too many rows are selected by the expression:

```
mysql> SELECT char_col FROM mytbl WHERE char_col = 00;
+----------+
| char_col |
+----------+
| abc      |
| def      |
| 00       |
| ghi      |
| jkl      |
| 00       |
| mno      |
+----------+
```

When you're uncertain about the way a value will be used, you may want to exploit MySQL's type conversion to force an expression to a value of a particular type, or to call a function that performs the desired conversion. The following list demonstrates several useful conversion techniques:

- Add `+0` or `+0.0` to a term to force conversion to a numeric value:

```
0x65                                  → 'e'
0x65 + 0                              → 101
0x65 + 0.0                            → 101.0
```

- Use `FLOOR()` to convert a floating-point number to an integer, or add `+0.0` to convert an integer to a floating-point number:

```
FLOOR(13.3)                           → 13
13 + 0.0                              → 13.0
```

  If you want rounding instead, use `ROUND()` rather than `FLOOR()`.

- Use `CAST()` or `CONCAT()` to turn a value into a string:

```
14                                    → 14
CAST(14 AS CHAR)                      → '14'
CONCAT(14)                            → '14'
```

  Or, use `HEX()` to convert a number to a hexadecimal string:

```
HEX(255)                              → 'FF'
HEX(65535)                            → 'FFFF'
```

  You also can use `HEX()` with a string value to convert it to a string of hex digit pairs representing successive bytes in the string:

```
HEX('abcd');                          → '61626364'
```

- Use `ASCII()` to convert a character to its ASCII value:

```
'A'                                   → 'A'
ASCII('A')                            → 65
```

  To go in the other direction from ASCII code to character, use `CHAR()`:

```
CHAR(65)                              → 'A'
```

- Use `DATE_ADD()` or `INTERVAL` arithmetic to force a string or number to be treated as a date:

```
20050101                              → 20050101
DATE_ADD(20050101, INTERVAL 0 DAY)    → '2005-01-01'
20050101 + INTERVAL 0 DAY             → '2005-01-01'
'20050101'                            → '20050101'
DATE_ADD('20050101', INTERVAL 0 DAY)  → '2005-01-01'
'20050101' + INTERVAL 0 DAY           → '2005-01-01'
```

- Generally, you can convert a temporal value to numeric form by adding zero:

```
CURDATE()                             → 2004-09-06
CURDATE()+0                           → 20040906
```

```
CURTIME()                                     → 16:43:21
CURTIME()+0                                   → 164321
```

- To convert a string from one character set to another, use `CONVERT()`. To check whether the result has the desired character set, use the `CHARSET()` function:

  ```
  'abcd'                                        → 'abcd'
  CONVERT('abcd' USING ucs2)                    → '\0a\0b\0c\0d'
  CHARSET('abcd')                               → 'latin1'
  CHARSET(CONVERT('abcd' USING ucs2))           → 'ucs2'
  ```

  Preceding a string with a character set introducer does not cause conversion of the string, but MySQL interprets it as though it has the character set indicated by the introducer:

  ```
  CHARSET(_ucs2 'abcd')                         → 'ucs2'
  ```

- To determine the hexadecimal value of the UTF-8 character that corresponds to a given hexadecimal UCS-2 character, combine `CONVERT()` with `HEX()`. The following expression determines the UTF-8 value of the trademark symbol:

  ```
  HEX(CONVERT(_ucs2 0x2122 USING utf8))         → 'E284A2'
  ```

- To change the collation of a string, use the `COLLATE` operator. To check whether the result has the desired collation, use the `COLLATION()` function:

  ```
  COLLATION('abcd')                             → 'latin1_swedish_ci'
  COLLATION('abcd' COLLATE latin1_bin)          → 'latin1_bin'
  ```

  The collation must be compatible with the string's character set. If it is not, use a combination of `CONVERT()` to convert the character set first and `COLLATE` to change the collation:

  ```
  CONVERT('abcd' USING latin2) COLLATE latin2_bin
  ```

- To convert a binary string to a non-binary string with a given character set, use `CONVERT()`:

  ```
  0x61626364                                    → 'abcd'
  0x61626364 = 'ABCD'                           → 0
  CONVERT(0x61626364 USING latin1) = 'ABCD'     → 1
  ```

  For binary quoted strings or hexadecimal values, an alternative is to use an introducer to change the interpretation of the binary string:

  ```
  _latin1 0x61626364 = 'ABCD'                   → 1
  ```

- To cast a non-binary string to a binary string, use the `BINARY` keyword:

  ```
  'abcd' = 'ABCD'                               → 1
  BINARY 'abcd' = 'ABCD'                        → 0
  'abcd' = BINARY 'ABCD'                        → 0
  ```