4

SQL According to MySQL

The Structured Query Language (SQL) is the language used to read and write to MySQL databases. Using SQL, you can search for data, enter new data, modify data, or delete data. SQL is simply the most fundamental tool you will need for your interactions with MySQL. Even if you are using some application or graphical user interface to access the database, somewhere under the covers that application is generating SQL.

SQL is a sort of "natural" language. In other words, an SQL statement should read—at least on the surface—like a sentence of English text. This approach has both benefits and drawbacks, but the end result is a language very unlike traditional programming languages such as C, Java, or Perl.

SQL Basics

SQL[*] is "structured" in the sense that it follows a very specific set of rules. A computer program can easily parse a formulated SQL query. In fact, the O'Reilly book *lex & yacc* by John Levine, Tony Mason, and Doug Brown implements a SQL grammar to demonstrate the process of writing a program to interpret language! A *query* is a fully-specified command sent to the database server, which then performs the requested action. Below is an example of an SQL query:

```
SELECT name FROM people WHERE name LIKE 'Stac%'
```

As you can see, this statement reads almost like a form of broken English: "Select names from a list of people where the names are like Stac." SQL uses very few of

[*] Pronounced either "sequel" or "ess-que-ell." Certain people get very religious about the pronunciation of SQL. Ignore them. It is important to note, however, that the "SQL" in MySQL is properly pronounced "ess-que-ell."

Copyright © 2001 O'Reilly & Associates, Inc.

the formatting and special characters that are typically associated with computer languages. Consider, for example, “$++;($*++/$|);$&$^,,;$!” in Perl versus “SELECT value FROM table” in SQL.

The SQL Story

IBM invented SQL in the 1970s shortly after Dr. E. F. Codd first invented the concept of a relational database. From the beginning, SQL was an easy to learn, yet powerful language. It resembles a natural language such as English, so that it might be less daunting to a nontechnical person. In the 1970s, even more than today, this advantage was an important one.

There were no casual hackers in the early 1970s. No one grew up learning BASIC or building web pages in HTML. The people programming computers were people who knew everything about how a computer worked. SQL was aimed at the army of nontechnical accountants and business and administrative staff that would benefit from being able to access the power of a relational database.

SQL was so popular with its target audience, in fact, that in the 1980s the Oracle corporation launched the world’s first publicly available commercial SQL system. Oracle SQL was a huge hit and spawned an entire industry built around SQL. Sybase, Informix, Microsoft, and several other companies have since come forward with their implementations of a SQL-based Relational Database Management System (RDBMS).

At the time Oracle and its first competitors hit the scene, SQL was still brand new and there was no standard. It was not until 1989 that the ANSI standards body issued the first public SQL standard. These days it is referred to as SQL89. This new standard, unfortunately, did not go far enough into defining the technical structure of the language. Thus, even though the various commercial SQL languages were drawing closer together, differences in syntax still made it non-trivial to switch among implementations. It was not until 1992 that the ANSI SQL standard came into its own.

The 1992 standard is called both SQL92 and SQL2. The SQL2 standard expanded the language to accommodate as many of the proprietary extensions added by the commercial implementations as was possible. Most cross-DBMS tools have standardized on SQL2 as the way in which they talk to relational databases. Due to the extensive nature of the SQL2 standard, however, relational databases that implement the full standard are very complex and very resource intensive.

Copyright © 2001 O’Reilly & Associates, Inc.

SQL2 is not the last word on the SQL standard. With the growing popularity of object-oriented database management systems (OODBMS) and object-relational database management systems (ORDBMS), there has been increasing pressure to capture support for object-oriented database access in the SQL standard. The recent SQL3 standard is the answer to this problem.

When MySQL came along, it took a new approach to the business of database server development. Instead of manufacturing another giant RDBMS and risk having nothing more to offer than the big guys, Monty created a small, fast implementation of the most commonly used SQL functionality. Over the years, that basic functionality has grown to support just about anything you might want to do with 80% of database applications.

The Design of SQL

As we mentioned earlier, SQL resembles a human language more than a computer language. SQL accomplishes this resemblance by having a simple, defined imperative structure. Much like an English sentence, individual SQL commands, called "queries," can be broken down into language parts. Consider the following examples:

```
CREATE   TABLE             people (name CHAR(10))
verb     object            adjective phrase

INSERT   INTO people       VALUES ('me')
verb     indirect object   direct object

SELECT   name              FROM people             WHERE name LIKE '%e'
verb     direct object     indirect object         adj. phrase
```

Most implementations of SQL, including MySQL, are case-insensitive. Specifically, it does not matter how you type SQL keywords as long as the spelling is correct. The CREATE example from above could just as well appeared:

```
cREatE TAblE people (name cHaR(10))
```

The case-insensitivity only extends to SQL keywords.* In MySQL, names of databases, tables, and columns are case-sensitive. This case-sensitivity is not necessarily true for all database engines. Thus, if you are writing an application that should work against all databases, you should act as if names are case-sensitive.

* For the sake of readability, we capitalize all SQL keywords in this book. We recommend this convention as a solid "best practice" technique.

Copyright © 2001 O'Reilly & Associates, Inc.

This first element of an SQL query is always a verb. The verb expresses the action you wish the database engine to take. While the rest of the statement varies from verb to verb, they all follow the same general format: you name the object upon which you are acting and then describe the data you are using for the action. For example, the query CREATE TABLE people (name CHAR(10)) uses the verb CREATE, followed by the object TABLE. The rest of the query describes the table to be created.

An SQL query originates with a client—the application that provides the façade through which a user interacts with the database. The client constructs a query based on user actions and sends the query to the SQL server. The server then must process the query and perform whatever action was specified. Once the server has done its job, it returns some value or set of values to the client.

Because the primary focus of SQL is to communicate actions to the database server, it does not have the flexibility of a general-purpose language. Most of the functionality of SQL concerns input to and output from the database: adding, changing, deleting, and reading data. SQL provides other functionality, but always with an eye towards how it can be used to manipulate the data within the database.

Sending SQL to MySQL

You can send SQL to MySQL using a variety of mechanisms. The most common way is through some programming API from Part III. For the purposes of this chapter, however, we recommend you use the command line tool *mysql*. When you run this program at the command line, it prompts you for SQL to enter:

```
[09:04pm] carthage$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 3.22.29

Type 'help' for help.

mysql>
```

The *mysql* command above says to connect to the MySQL server on the local machine as the user root with the client prompting you for a password. Another option, the -h option, enables you to connect to MySQL servers on remote machines:

```
[09:04pm] carthage$ mysql -u root -h db.imaginary.com -p
```

There is absolutely no relationship between UNIX or Windows 2000 user names and MySQL user names. Users have to be added to MySQL independently of the host on which they reside. No one therefore has an account on a clean MySQL install except root. As a general rules, you should never connect to MySQL as root

Copyright © 2001 O’Reilly & Associates, Inc.

except when performing database administration tasks. If you have a clean installation of MySQL that you can afford to throw away, then it is useful to connect as root for the purposes of this chapter so that you may create and drop databases. Otherwise, you will have to connect to MySQL as whatever user name has been assigned to you.

You can enter your SQL commands all on a single line, or you can split them across multiple lines. MySQL patiently waits for a semi-colon before executing the SQL you enter:

```
mysql> SELECT book_number
    -> FROM book
    -> ;
+-------------+
| book_number |
+-------------+
|           1 |
|           2 |
|           3 |
+-------------+
3 rows in set (0.00 sec)
```

With the *mysql* command line, you generally get a command history depending on how it was compiled. If it is compiled into your *mysql* client, you can use the up and down arrows on your keyboard to navigate through past SQL commands you have executed. For more information on the *mysql* tool, see Chapter 20.

Database Creation

In order to get started using MySQL, you need to create a database to use. First, let's take a look at the databases that come with a clean MySQL installation using the `SHOW DATABASES` command. On a clean install of MySQL 3.23.40 on Mac OS X, the following tables already exist:

```
mysql> SHOW DATABASES;
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.37 sec)

mysql>
```

The first database, `mysql`, is MySQL's system database. You will learn more about it in Chapter 5. The second table is a play table you can use to learn MySQL and run tests against. You may find other databases on your server if you are not deal-

Copyright © 2001 O'Reilly & Associates, Inc.

ing with a clean install. For right now, however, we want to create a new database to illustrate the use of the MySQL CREATE statement:

```
CREATE DATABASE TEMPDB;
```

And then to work with the new database TEMPDB:

```
USE TEMPDB;
```

Finally, you can delete that database by issuing the DROP DATABASE command:

```
DROP DATABASE TEMPDB;
```

You will find as you explore SQL that you create new things using the CREATE statement and destroy things using the DROP statement just as we used them here.

Table Management

You should now feel comfortable connecting to a database on a MySQL server. For the rest of the chapter, you can use the test database that comes with MySQL or your own play database. Using the SHOW command, you can display a list of tables in the current database in a similar manner to the way you used it to show databases. In a brand new install, the test database has no tables. The following shows the output of the SHOW TABLES command when connected to the mysql system database:

```
mysql> SHOW TABLES;
+-----------------+
| Tables_in_mysql |
+-----------------+
| columns_priv    |
| db              |
| func            |
| host            |
| tables_priv     |
| user            |
+-----------------+
6 rows in set (0.00 sec)
```

To get a look at the what one of these tables looks like, you can use the DESCRIBE command:

```
mysql> DESCRIBE db;
+-----------------+-----------------+------+-----+---------+-------+
| Field           | Type            | Null | Key | Default | Extra |
+-----------------+-----------------+------+-----+---------+-------+
| Host            | char(60) binary |      | PRI |         |       |
| Db              | char(64) binary |      | PRI |         |       |
| User            | char(16) binary |      | PRI |         |       |
| Select_priv     | enum('N','Y')   |      |     | N       |       |
| Insert_priv     | enum('N','Y')   |      |     | N       |       |
| Update_priv     | enum('N','Y')   |      |     | N       |       |
| Delete_priv     | enum('N','Y')   |      |     | N       |       |
| Create_priv     | enum('N','Y')   |      |     | N       |       |
| Drop_priv       | enum('N','Y')   |      |     | N       |       |
| Grant_priv      | enum('N','Y')   |      |     | N       |       |
| References_priv | enum('N','Y')   |      |     | N       |       |
| Index_priv      | enum('N','Y')   |      |     | N       |       |
| Alter_priv      | enum('N','Y')   |      |     | N       |       |
+-----------------+-----------------+------+-----+---------+-------+
13 rows in set (0.36 sec)
```

This output describes each column in the table with its data type, whether or not it can contain null values, what kind of key it is, any default values, and extra information. If all of this means nothing to you, don't worry. We will describe each of these elements as the chapter progresses.

You should now be ready to create your first table. You will, of course, want to connect back to the `test` database since you definitely do not want to be adding tables to the `mysql` database. The *table*, a structured container of data, is the most basic concept of a relational database. Before you can begin adding data to a table, you must define the table's structure. Consider the following layout:

```
+---------------------------------+
|             people              |
+-------------+-------------------+
| name        | char(10) not null |
| address     | text(100)         |
| id          | int               |
+-------------+-------------------+
```

Not only does the table contain the names of the columns, but it also contains the types of each field as well as any additional information the fields may have. A field's data type specified what kind of data the field can hold. SQL data types are similar to data types in other programming languages. The full SQL standard allows for a large range of data types. MySQL implements most of them as well as a few MySQL-specific types.

The general syntax for table creation is:

```
CREATE TABLE table_name (column_name1 type [modifiers]
                      [, column_name2 type [modifiers]]
)
```

What constitutes a valid identifier—a name for a table or column—varies from DBMS to DBMS. MySQL allows up to 64 characters in an identifier, supports the character '$' in identifiers, and lets identifiers start with a valid number. More important, however, MySQL considers any valid letter for your local character set to be a valid letter for identifiers.

Copyright © 2001 O'Reilly & Associates, Inc.

A *column* is the individual unit of data within a table. A table may have any number of columns, but large tables may be inefficient. This is where good database design, discussed in Chapter 8, *Database Design*, becomes an important skill. By creating properly normalized tables, you can "join" tables to perform a single search from data housed in more than one table. We discuss the mechanics of a join later in the chapter.

Consider the following create statement:

```
CREATE TABLE USER (
    USER_ID    BIGINT UNSIGNED NOT NULL PRIMARY KEY,
    USER_NAME  CHAR(10)        NOT NULL,
    LAST_NAME  VARCHAR(30),
    FIRST_NAME VARCHAR(30),
    OFFICE     CHAR(2)         NOT NULL DEFAULT 'NY');
```

This statement creates a table called `USER` with four columns: `USER_ID`, `USER_NAME`, `LAST_NAME`, `FIRST_NAME`, and `OFFICE`. After each column name comes the data type for that column followed by any modifiers. We will discuss data types and the `PRIMARY KEY` modifier later in this chapter.

The NOT NULL modifier indicates that the column may not contain any null values. If you try to assign a null value to that column, your SQL will generate an error. Actually, there are a couple of exceptions to this rule. First, if the column is `AUTO_INCREMENT`, a null value will cause a value to be automatically generated. We cover auto-incrementing later in the chapter. The second exception is for columns that specify default values like the `OFFICE` column. In this case, the `OFFICE` column will be assigned a value of 'NY' when a null value is assigned to the column.

Like most things in life, destruction is much easier than creation. The command to drop a table from the database is:

```
DROP TABLE table_name
```

This command will completely remove all traces of that table from the database. MySQL will remove all data within the destroyed table from existence. If you have no backups of the table, you absolutely cannot recover from this action. The moral of this story is to always keep backups and be very careful about dropping tables. You will thank yourself for it some day.

With MySQL, you can specify more than one table to delete by separating the table names with commas. For example, `DROP TABLE` *`people, animals, plants`* would delete the three named tables. You can also use the `IF EXISTS` modifier to avoid an error should the table not exist when you try to drop it. This modifier is useful for huge scripts designed to create a database and all its tables. Before the create, you do a `DROP TABLE` *`table_name`* `IF EXISTS`.

Copyright © 2001 O'Reilly & Associates, Inc.

MySQL Data Types

In a table, each column has a type. As we mentioned earlier, a SQL data type is similar to a data type in traditional programming languages. While many languages define a bare-minimum set of types necessary for completeness, SQL goes out of its way to provide types such as `MONEY` and `DATE` that will be useful to every day users. You could store a `MONEY` type in a more basic numeric type, but having a type specifically dedicated to the nuances of money processing helps add to SQL's ease of use—one of SQL's primary goals.

Chapter 17, *MySQL Data Types*, provides a full reference of SQL types supported by MySQL. Table 4-1 is an abbreviated listing of the most common types.

Table 4-1. Common MySQL Data Types (see Chapter 17 for a full list)

Data Type	Description
INT	An integer value. MySQL allows an INT to be either signed or unsigned.
REAL	A floating point value. This type offers a greater range and precision than the INT type, but it does not have the exactness of an INT.
CHAR(*length*)	A fixed-length character value. No CHAR fields can hold strings greater in length than the specified value. Fields of lesser length are padded with spaces. This type is likely the most commonly used type in any SQL implementation.
TEXT(*length*)	A variable length character value.
DATE	A standard date value. The DATE type stores arbitrary dates for the past, present, and future. MySQL is Y2K compliant in its date storage.
TIME	A standard time value. This type stores the time of day independent of a particular date. When used together with a date, a specific date and time can be stored. MySQL additionally supplies a DATETIME type that will store date and time together in one field.

MySQL supports the UNSIGNED attribute for all numeric types. This modifier forces the column to accept only positive (unsigned) numbers. Unsigned fields have an upper limit that is double that of their signed counterparts. An unsigned TINYINT—MySQL's single byte numeric type—has a range of 0 to 255 instead of the -127 to 127 range of its signed counterpart.

MySQL provides more types than those mentioned above. In day-to-day programming, however, you will find yourself using mostly these types. The size of the data you wish to store plays a large role the design of your MySQL tables.

Copyright © 2001 O'Reilly & Associates, Inc.

Numeric Types

Before you create a table, you should have a good idea of what kind of data you wish to store in the table. Beyond obvious decisions about whether your data is character-based or numeric, you should know the approximate size of the data to be stored. If it is a numeric field, what is its maximum possible value? What is its minimum possible value? Could that change in the future? If the minimum is always positive, you should consider an unsigned type. You should always choose the smallest numeric type that can support your largest conceivable value. If, for example, we had a field that represented the population of a state, we would use an unsigned `INT` field. No state can have a negative population. Furthermore, in order for an unsigned `INT` field not to be able to hold a number representing a state's population, that state's population would have to be roughly the population of the entire Earth.

Character Types

Managing character types is a little more complicated. Not only do you have to worry about the minimum and maximum string lengths, but you also have to worry about the average size, the amount of variation likely, and the need for indexing. For our current purposes, an *index* is a field or combination of fields on which you plan to search—basically, the fields in your `WHERE` clause. Indexing is, however, much more complicated than this simplistic description, and we will cover indexing later in the chapter. The important fact to note here is that indexing on character fields works best when the field is fixed length. If there is little—or, preferably, no—variation in the length of your character-based fields, then a CHAR type is likely the right answer. An example of a good candidate for a CHAR field is a country code. The ISO provides a comprehensive list of standard two-character representations of country codes (US for the U.S.A., FR for France, etc.).* Since these codes are always exactly two characters, a CHAR(2) is always the right answer for this field.

A value does not need to be invariant in its length to be a candidate for a CHAR field. It should, however, have very little variance. Phone numbers, for example, can be stored safely in a CHAR(13) field even though phone number length varies from nation to nation. The variance simply is not that great, so there is no value to making a phone number field variable in length. The important thing to keep in mind with a CHAR field is that no matter how big the actual string being stored is,

* Don't be lulled into believing states/provinces work this way. If you want to write an application that works in an international environment and stores state/province codes, make sure to make it a CHAR(3) since Australia uses three-character state codes. Also note that there is a 3-character ISO country-code standard.

Copyright © 2001 O'Reilly & Associates, Inc.

the field always takes up exactly the number of characters specified as the field's size—no more, no less. Any difference between the length of the text being stored and the length of the field is made up by padding the value with spaces. While the few potential extra characters being wasted on a subset of the phone number data is not anything to worry about, you do not want to be wasting much more. Variable-length text fields meet this need.

A good, common example of a field that demands a variable-length data type is a web URL. Most web addresses can fit into a relatively small amount of space—*http://www.ora.com*, *http://www.imaginary.com*, *http://www.mysql.com*—and consequentially do not represent a problem. Occasionally, however, you will run into web addresses like:

> *http://www.winespectator.com/Wine/Spectator/ _notes|5527293926834323221480431354?Xv11=&Xr5=&Xv1=&type-region-search-code=&Xa14=flora+springs&Xv4=.*

If you construct a `CHAR` field large enough to hold that URL, you will be wasting a significant amount of space for most every other URL being stored. Variable-length fields let you define a field length that can store the odd, long-length value while not wasting all that space for the common, short-length values.

Variable-length text fields in MySQL use precisely the minimum storage space required to store an individual field. A `VARCHAR(255)` column that holds the string "hello world," for example, only takes up twelve bytes (one byte for each character plus an extra byte to store the length).

> In opposition to the ANSI standard, `VARCHAR` in MySQL fields are not padded. Any extra spaces are removed from a value before it is stored.

You cannot store strings whose lengths are greater than the field length you have specified. With a `VARCHAR(4)` field, you can store at most a string with 4 characters. If you attempt to store the string "happy birthday," MySQL will truncate the string to "happ." The downside is that there is no way to store the odd string that exceeds your designated field size. Table 4-2 shows the storage space required to

Copyright © 2001 O'Reilly & Associates, Inc.

store the 144 character Wine Spectator URL shown above along with an average-sized 30 character URL.

Table 4-2. The Storage Space Required by the Different MySQL Character Types

Data Type	Storage for a 144 Character String	Storage for a 30 Character String	Maximum String Size
CHAR(150)	150	150	255
VARCHAR(150)	145	31	255
TINYTEXT(150)	145	31	255
TEXT(150)	146	32	65535
MEDIUMTEXT(150)	147	33	16777215
LONGTEXT(150)	148	34	4294967295

In this table, you will note that storage requirements grow one byte at a time for variable-length types of `MEDIUM_TEXT` and `LONGTEXT`. This is because `TEXT` uses an extra byte to store the potentially greater length of the text it contains. Similarly, `MEDIUM_TEXT` uses an extra two bytes over `VARCHAR` and `LONGTEXT` an extra three bytes.

If, after years of uptime with your database, you find that the world has changed and a field that once comfortably existed as a `VARCHAR(25)` now must be able to hold strings as long as 30 characters, you are not out of luck. MySQL provides a command called `ALTER TABLE` that enables you to redefine a field type without losing any data.

```
ALTER TABLE mytable MODIFY mycolumn LONGTEXT
```

Binary Data Types

MySQL provides a set of binary data types that closely mirror their character counterparts. The MySQL binary types are `CHAR BINARY`, `VARCHAR BINARY`, `TINYBLOB`, `BLOB`, `MEDIUMBLOB`, and `LONGBLOB`. The practical distinction between character types and their binary counterparts is the concept of encoding. *Binary data* is basically just a chunk of data that MySQL makes no effort to interpret. *Character data*, on the other hand, is assumed to represent textual data from human alphabets. It thus is encoded and sorted based on rules appropriate to the character set in question. In the case of installations on an ASCII system, MySQL sorts binary in a case-insensitive, ASCII order.

Enumerations and Sets

MySQL provides two other special kinds of types. The `ENUM` type allows you specify at table creation a list of possible values that can be inserted into that field. For

Copyright © 2001 O'Reilly & Associates, Inc.

example, if you had a column named fruit into which you wanted to allow only “apple,” “orange,” “kiwi,” or “banana,” you would assign this column the type ENUM:

```
CREATE TABLE meal(meal_id INT NOT NULL PRIMARY KEY,
                  fruit ENUM('apple', 'orange', 'kiwi',
                             'banana'))
```

When you insert a value into that column, it must be one of the specified fruits. Because MySQL knows ahead of time what valid values are for the column, it can abstract them to some underlying numeric type. In other words, instead of storing “apple” in the column as a string, it stores it as a single byte number. You just use “apple” when you call the table or when you view results from the table.

The MySQL SET type works in the same way, except it lets you store multiple values in a field at the same time.

Other Kinds of Data

Every piece of data you will ever encounter can be stored using numeric or character types. Technically, you could even store numbers as character types. Just because you can do so, however, does not mean that you should do so. Consider, for example, storing money in the database. You could store that as an INT or a REAL. While a REAL might seem more intuitive—money requires decimal places, after all—an INT fields actually makes more sense. With floating point values like REAL fields, it is often impossible to capture a number with a specific decimal value. If, for example, you insert the number 0.43 to represent $0.43, MySQL may store that as 0.42999998. This small difference can be problematic when applied to a large number of mathematical operations. By storing the number as an INT and inserting the decimal into the right place, you can be certain that the value represents exactly what you intend it to represent.

Isn’t all of that a major pain? Wouldn’t it be nice if MySQL provided some sort of data type specifically suited to money values? MySQL provides special data types to handle special kinds of data. MONEY is an example of one of these kinds of data. DATE is another.

Indexing

While MySQL has better performance than any of the larger database servers, some problems still call for careful database design. For instance, if we had a table with millions of rows of data, a search for a specific row would take a long time. Most database engines enable you to help it in these searches through a tool called an index.

Copyright © 2001 O’Reilly & Associates, Inc.

Indices help the database store data in a way that makes for quicker searches. Unfortunately, you sacrifice disk space and modification speed for the benefit of quicker searches. The most efficient use of indices is to create an index for columns on which you tend to search the most. MySQL supports the following syntax for index creation:

```
CREATE INDEX index_name ON tablename (column1,
                                     column2,
                                     ...,
                                     columnN)
```

MySQL also lets you create an index at the same time you create a table using the following syntax:

```
CREATE TABLE materials (id          INT      NOT NULL,
                        name        CHAR(50) NOT NULL,
                        resistance INT,
                        melting_pt REAL,
                        INDEX index1 (id, name),
                        UNIQUE INDEX index2 (name))
```

The previous example creates two indices for the table. The first index—named `index1`—consists of both the `id` and `name` fields. The second index includes only the `name` field and specifies that values for the name field must always be unique. If you try to insert a field with a `name` held by a row already in the database, the insert will fail. All fields declared in a unique index must be declared as being `NOT NULL`.

Even though we created an index for `name` by itself, we did not create an index for just `id`. If we did want such an index, we would not need to create it—it is already there. When an index contains more than one column (for example: `name`, `rank`, and `serial_number`), MySQL reads the columns in order from left to right. Because of the structure of the index MySQL uses, any subset of the columns from left to right are automatically created as indices within the "main" index. For example, `name` by itself and `name` and `rank` together are both "free" indices created when you create the index `name`, `rank`, `serial_number`. An index of `rank` by itself or `name` and `serial_number` together, however, is not created unless you explicitly create it yourself.

MySQL also supports the ANSI SQL semantics of a special index called a primary key. In MySQL, a primary key is a unique key with the name `PRIMARY`. By calling a column a primary key at creation, you are naming it as a unique index that will support table joins. The following example creates a cities table with a primary key of `id`.

```
CREATE TABLE cities (id      INT  NOT NULL PRIMARY KEY,
                     name    VARCHAR(100),
                     pop     MEDIUMINT,
                     founded DATE)
```

Before you create a table, you should determine which fields, if any, should be keys. As we mentioned above, any fields which will be supporting joins are good candidates for primary keys. See Chapter 8 for a detailed discussion on how to design your tables with good primary keys.

Though MySQL supports the ANSI syntax for foreign keys, it does not actually use them to perform integrity checking in the database. This is an sue where the introduction of a feature would cause a slowdown in performance with little real benefit. Applications themselves should generally worry about foreign key integrity.

Managing Data

The first thing you do with a newly created table is add data to it. With the data in place, you may want to make changes and eventually remove it.

Inserts

Adding data to a table is one of the more straightforward concepts in SQL. You have already seen several examples of it in this book. MySQL supports the standard SQL INSERT syntax:

```
INSERT INTO table_name (column1, column2, ..., columnN)
VALUES (value1, value2, ..., valueN)
```

When inserting data into numeric fields, you can insert the value as is; for all other fields, you must wrap them in single quotes. For example, to insert a row of data into a table of addresses, you might issue the following command:

```
INSERT INTO addresses (name, address, city, state, phone, age)
VALUES('Irving Forbush', '123 Mockingbird Lane', 'Corbin', 'KY',
       '(800) 555-1234', 26)
```

In addition, the escape character—‘\’ by default—enables you to escape single quotes and other literal instances of the escape character:

```
# Insert info for the directory Stacie’s Directory which
# is in c:\Personal\Stacie
INSERT INTO files (description, location)
VALUES ('Stacie\'s Directory', 'C:\\Personal\\Stacie')
```

MySQL allows you to leave out the column names as long as you specify a value for every single column in the table in the exact same order they were specified in the table’s `CREATE` call. If you want to use the default values for a column, however, you must specify the names of the columns for which you intend to insert non-default data. If you do not have a default value set up for a column and that

Copyright © 2001 O’Reilly & Associates, Inc.

column is NOT NULL, you must include that column in the INSERT statement with a non-NULL value. If the earlier files table had contained a column called size, then the default value would be used. MySQL allows you to specify a custom default value in the table's CREATE.

Newer versions of MySQL support a nonstandard INSERT call for inserting multiple rows at once:

```
INSERT INTO foods VALUES (NULL, 'Oranges', 133, 0, 2, 39),
                         (NULL, 'Bananas', 122, 0, 4, 29),
                         (NULL, 'Liver', 232, 3, 15, 10)
```

While these nonstandard syntaxes supported by MySQL are useful for quick system administration tasks, you should not use them when writing database applications unless you really need the speed benefit they offer. As a general rule, you should stick as close to the ANSI SQL2 standard as MySQL will let you. By doing so, you are making certain that your application can run against any other database in the future. Being flexible is especially critical for people with mid-range database needs because such users generally hope one day to become people with high-end database needs.

Another non-standard syntax supported by MySQL is where you specify the column name and value together:

```
INSERT INTO book SET title='The Vampire Lestat', author='Anne Rice';
```

Another approach to inserting data is by using the data from some other table (or group of tables) to populate your new table. For example:

```
INSERT INTO foods (name, fat)
SELECT food_name, fat_grams FROM recipes
```

You should note that the number of columns in the INSERT matches the number of columns in the SELECT. In addition, the data types for the INSERT columns must match the data types for the corresponding SELECT columns. Finally, the SELECT clause in an INSERT statement cannot contain an ORDER BY modifier and cannot be selected from the same table where the INSERT is occurring.

Sequence Generation

The best kind of primary key is one that has absolutely no meaning in the database except to act as a primary key. The best way to achieve this is to make a numeric primary key that increments every time you insert a new row. Looking at the cities table shown earlier, the first city you insert would have an id of 1, the second 2, the third 3, and so on. In order to successfully manage this sequencing of a primary key, you need some way to guarantee that a number can be read and incremented by one and only one client at a time.

When you create a table in MySQL, you can specify at most one column as being AUTO_INCREMENT. When you do this, you can automatically have this column insert the highest current value for that column + 1 when you insert a row and specify NULL or 0 for that row's value. The AUTO_INCREMENT row must be indexed. The following command creates the cities table with the id field being AUTO_INCREMENT:

```
CREATE TABLE cities (id      INT  NOT NULL PRIMARY KEY AUTO_INCREMENT,
                     name    VARCHAR(100),
                     pop     MEDIUMINT,
                     founded DATE)
```

The first time you insert a row, the id field for your first row will be 1 so long as you use NULL or 0 for that field in the INSERT statement. For example, this command takes advantage of the AUTO_INCREMENT feature:

```
INSERT INTO cities (id, name, pop)
VALUES (NULL, 'Houston', 3000000)
```

If no other values are in that table when you issue this command, MySQL will set this field to 1, not NULL (remember, it cannot be NULL). If other values are present in the table, the value inserted will be one greater than the largest current value for id.

Another way to implement sequences is by referring to the value returned by the LAST_INSERT_ID() function:

```
UPDATE table SET id=LAST_INSERT_ID (id+1);
```

Updates

The insertion of new rows into a database is just the start of data management. Unless your database is read-only, you will probably also need to make periodic changes to the data. The standard SQL modification statement looks like this:

```
UPDATE table_name
SET column1=value1, column2=value2, ..., columnN=valueN
[WHERE clause]
```

In addition to assigning literal values to a column, you can also assign calculate the values. You can even calculate the value based on a value in another column:

```
UPDATE years
SET end_year = begin_year+5
```

This command sets the value in the end_year column equal to the value in the begin_year column plus 5 for each row in that table.

Copyright © 2001 O'Reilly & Associates, Inc.

The WHERE Clause

You probably noted something earlier called the WHERE clause. In SQL, a WHERE clause enables you to pick out specific rows in a table by specifying a value that must be matched by the column in question. For example:

```
UPDATE bands
SET lead_singer = 'Ian Anderson'
WHERE band_name = 'Jethro Tull'
```

This UPDATE specifies that you should only change the lead_singer column for the row where band_name is identical to "Jethro Tull." If the column in question is not a unique index, that WHERE clause may match multiple rows. Many SQL commands employ WHERE clauses to help pick out the rows on which you wish to operate. Because the columns in the WHERE clause are columns on which you are searching, you should generally have indices created around whatever combinations you commonly use. We discuss the kinds of comparisons you can perform in the WHERE clause later in the chapter.

Deletes

Deleting data is a very straightforward operation. You simply specify the table from which you want to delete followed by a WHERE clause that identifies the rows you want to delete:

```
DELETE FROM table_name [WHERE clause]
```

As with other commands that accept a WHERE clause, the WHERE clause is optional. In the event you leave out the WHERE clause, you will delete all of the records in the table! Of all destructive commands in SQL, this is the easiest one to issue mistakenly.

Queries

The last common SQL command used is the one that enables you to view the data in the database: SELECT. This action is by far the most common action performed in SQL. While data entry and modifications do happen on occasion, most databases spend the vast majority of their lives serving up data for reading. The general form of the SELECT statement is as follows:

```
SELECT column1, column2, ..., columnN
FROM table1, table2, ..., tableN
[WHERE clause]
```

This syntax is certainly the most common way in which you will retrieve data from any SQL database. Of course, there are variations for performing complex and powerful queries. We cover the full range of the SELECT syntax in Chapter 16. The simplest form is this:

Copyright © 2001 O'Reilly & Associates, Inc.

```
SELECT 1;
```

This simple, though completely useless query returns a result set with a single row containing a single column with the value of 1. A more useful version of this query might be something like:

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| test       |
+------------+
1 row in set (0.01 sec)
```

The expression `DATABASE()` is a MySQL function that returns the value of the current database in use. We will cover functions in more detail later in the chapter. Nevertheless, you can see how simple SQL can provide a quick and dirty way of finding out important information.

Most of time, however, you will want to use slightly more complex queries that help you pull data from a table in the database. The first part of a `SELECT` statement enumerates the columns you wish to retrieve. You may specify a "*" to say that you want to select all columns. The `FROM` clause specifies which tables those columns come from. The `WHERE` clause identifies the specific rows to be used and enables you to specify how to join two tables.

Joins

Joins put the "relational" in relational databases. Specifically, a join enables you to match a row from one table up with a row in another table. The basic form of a join is what you may hear sometimes described as an *inner join*. Joining tables is a matter of specifying equality in columns from two tables:

```
SELECT book.title, author.name
FROM author, book
WHERE book.author = author.id
```

Consider a database where the `book` table looks like Table 4-3.

Table 4-3. A book Table

ID	Title	Author	Pages
1	The Green Mile	4	894
2	Guards, Guards!	2	302
3	Imzadi	3	354
4	Gold	1	405
5	Howling Mad	3	294

And the author table looks like Table 4-4.

Table 4-4. An author Table

ID	Name	Citizen
1	Isaac Asimov	US
2	Terry Pratchet	UK
3	Peter David	US
4	Stephen King	US
5	Neil Gaiman	UK

An inner join creates a virtual table by combining the fields of both tables for rows that satisfy the query in both tables. In our example, the query specifies that the `author` field of the `book` table must be identical to the `id` field of the `author` table. The query's result would thus look like Table 4-5.

Table 4-5. Query Results Based on an Inner Join

Book Title	Author Name
The Green Mile	Stephen King
Guards, Guards!	Terry Pratchet
Imzadi	Peter David
Gold	Isaac Asimov
Howling Mad	Peter David

Neil Gaiman is nowhere to be found in these results. He is left out because there is no value for his `author.id` value found in the `author` column of the `book` table. An inner join only contains those rows that exactly match the query. We will discuss the concept of an outer join later in the chapter for situations where we would be interested in the fact that we have an author in the database who does not have a book in the database.

Aliasing

When you use column names that are fully qualified with their table and column name, the names can grow to be quite unwieldy. In addition, when referencing SQL functions, which will be discussed later in the chapter, you will likely find it cumbersome to refer to the same function more than once within a statement. The aliased name, usually shorter and more descriptive, can be used anywhere in the same SQL statement in place of the longer name. For example:

```
# A column alias
SELECT long_field_names_are_annoying AS myfield
FROM table_name
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
WHERE myfield = 'Joe'
# A table alias under MySQL
SELECT people.names, tests.score
FROM tests, really_long_people_table_name AS people
# A table alias under mSQL
SELECT people.names, tests.score
FROM tests, really_long_people_table_name=people
```

Ordering and Grouping

The results you get back from a select are, by default, indeterminate in the order they will appear. Fortunately, SQL provides some tools for imposing order on this seemingly random list: ordering and grouping.

Basic Ordering

You can tell a database that it should order any results you see by a certain column. For example, if you specify that a query should order the results by `last_name`, then the results will appear alphabetized according to the `last_name` value. Ordering comes in the form of the `ORDER BY` clause:

```
SELECT last_name, first_name, age
FROM people
ORDER BY last_name, first_name
```

In this situation, we are ordering by two columns. You can order by any number of columns, but the columns must be named in the `SELECT` clause. If we had failed to select the `last_name` above, we could not have ordered by the `last_name` field.

If you want to see things in reverse order, add the `DESC` keyword:

```
ORDER BY last_name DESC
```

The `DESC` keyword applies only to the field that comes right before it. If you are sorting on multiple fields, only the field right before `DESC` is reversed; the others occur in ascending order.

Localized Sorting

Sorting is actually a very complex problem for applications that need to be able to run on computers all over the world. The rules for sorting strings vary from alphabet to alphabet, even when two alphabets use mostly the same symbols. MySQL handles the problem of sorting by making it dependent on the character set of the MySQL engine. Out of the box, the default character set is ISO-8859-1 (Latin-1). MySQL uses the sorting rules for Swedish and Finnish with ISO-8859-1.

To change the sorting rules, you change the character set. First, you need to make sure the character set is compiled into the server when you compile MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.

Chapter 3 contains detailed instructions on installing MySQL with specific character sets, including those you define yourself. With the proper character set compiled into the server, you can change the default character set by launching the server with the argument *--default-character-set*=CHARSET.

Because of the simplicity of the English alphabet, the use of a single set of sorting rules MySQL associates with ISO-8859-1 does not affect English sorting. Unfortunately, different languages can have different sorting rules even though they share the same character sets. Swedish and German, for example, both use the ISO-8859-1 character set. Swedish sorts 'ä' after 'z', while German sorts 'ä' before 'a'. The default rules therefore fail German users.

MySQL lets you address this problem by creating custom character sets. When you compile the driver, you can compile in support for whatever character sets you desire as long as you have a configuration file for that character set. This file contains the characters that make up the character set and the rules for sorting them. You can write your own as well as use the ones that come with MySQL.

The real problem here is that MySQL incorrectly associates sorting rules with character sets. A character set is nothing more than a grouping of characters with a related purpose. Nothing about the ISO-8859-1 character set implies sorting for Swedes, Italians, Germans, or anyone else. when working with MySQL, however, you just need to remember that sorting rules are directly tied to the character set.

Grouping

Grouping lets you group rows with a similar value into a single row in order to operate on them together. You usually do this to perform aggregate functions on the results. We will go into functions a little later in the chapter.

Consider the following:

```
mysql> SELECT name, rank, salary FROM people;
+--------------+----------+--------+
| name         | rank     | salary |
+--------------+----------+--------+
| Jack Smith   | Private  |  23000 |
| Jane Walker  | General  | 125000 |
| June Sanders | Private  |  22000 |
| John Barker  | Sargeant |  45000 |
| Jim Castle   | Sargeant |  38000 |
+--------------+----------+--------+
5 rows in set (0.01 sec)
```

If you group the results by rank, the output changes:

```
mysql> SELECT rank FROM people GROUP BY rank;
+----------+
| rank     |
+----------+
| General  |
| Private  |
| Sargeant |
+----------+
3 rows in set (0.01 sec)
```

Now that you have the output grouped, you can finally find out the average salary for each rank. Again, we will discuss more on the functions you see in this example later in the chapter.

```
mysql> SELECT rank, AVG(salary) FROM people GROUP BY rank;
+----------+-------------+
| rank     | AVG(salary) |
+----------+-------------+
| General  | 125000.0000 |
| Private  |  22500.0000 |
| Sargeant |  41500.0000 |
+----------+-------------+
3 rows in set (0.04 sec)
```

The power of ordering and grouping combined with the utility of SQL functions enables you to do a great deal of data manipulation even before you retrieve the data from the server. You should take great care not to rely too heavily on this power. While it may seem like an efficiency gain to place as much processing load as possible onto the database server, it is not really the case. Your client application is dedicated to the needs of a particular client, while the server is being shared by many clients. Because of the greater amount of work a server already has to do, it is almost always more efficient to place as little load as possible on the database server. MySQL may be the fastest database around, but you do not want to waste that speed on processing that a client application is better equipped to manage.

If you know that a lot of clients will be asking for the same summary information often (for instance, data on a particular rank in our previous example), just create a new table containing that information and keep it up to date as the original tables change. This is similar to caching and is a common database programming technique.

Limiting Results

Sometimes an application is looking for only the first few rows that match a query. Limiting queries can help avoid problems bogging down the network with unwanted results. MySQL enables an application to limit the number of results through a `LIMIT` clause in a query:

Copyright © 2001 O’Reilly & Associates, Inc.

```
SELECT * FROM people ORDER BY name LIMIT 10;
```

Of course, to get the last 10 people from the table, you can use the `DESC` keyword. If you want people from the middle, however, you have to get a bit trickier. To accomplish this task, you need to specify the number of the first record you want to see (record 0 is the first record, 1 the second) and the number of rows you want to see:

```
SELECT * FROM people ORDER BY name LIMIT 19, 30;
```

This sample displays records 20 through 49. The 19 in the `LIMIT` clause tells MySQL to start with the twentieth record. The thirty then tells MySQL to return the next 30 records.

SQL Operators

So far, we have basically used the = operator for the obvious task of verifying that two values in a WHERE clause equal one another. Other fairly basic operations include <>, >, <, <=, and >=. One special thing to note is that MySQL allows you to use either <> or `!=` for "not equal". Table 4-6 contains a full set of simple SQL operators.

Table 4-6. . The Simple SQL Operators Supported by MySQL

Operator	Context	Description
+	Arithmetic	Addition
-	Arithmetic	Subtraction
*	Arithmetic	Multiplication
/	Arithmetic	Division
=	Comparison	Equal
<> or !=	Comparison	Not equal
<	Comparison	Less than
>	Comparison	Greater than
<=	Comparison	Less than or equal to
>=	Comparison	Greater than or equal to
AND	Logical	And
OR	Logical	Or
NOT	Logical	Negation

MySQL operators have the following rules of precedence:

1. BINARY
2. NOT

Copyright © 2001 O’Reilly & Associates, Inc.

3. - (unary minus)
4. * / %
5. + -
6. << >>
7. &
8. |
9. < <= > >= = <=> <> IN IS LIKE REGEXP
10. BETWEEN
11. AND
12. OR

Logical Operators

SQL's logical operators—`AND`, `OR`, and NOT—let you build more dynamic `WHERE` clauses. The `AND` and `OR` operators specifically let you add multiple criteria to a query:

```
SELECT USER_NAME
FROM USER
WHERE AGE > 18 AND STATUS = 'RESIDENT';
```

This sample query provides a list of all users who are residents and are old enough to vote.

You can build increasingly complex queries through the use of parentheses. The parentheses tell MySQL which comparisons to evaluate first:

```
SELECT USER_NAME
FROM USER
WHERE (AGE > 18 AND STATUS = 'RESIDENT')
OR (AGE > 18 AND STATUS = 'APPLICANT');
```

In this more complex query, we are looking for anyone currently eligible to vote as well as people who might be eligible in the near future. Finally, you can use the `NOT` operator to negate an entire expression:

```
SELECT USER_NAME
FROM USER
WHERE NOT (AGE > 18 AND STATUS = 'RESIDENT');
```

Null's Idiosyncrasies

Null is a tricky concept for most people new to databases to understand. As in other programming languages, null is not a value, but an absence of a value. This concept is useful, for example, if you have a customer profiling database that grad-

Copyright © 2001 O'Reilly & Associates, Inc.

ually gathers information about your customers as they offer it. When you first create the record, for example, you may not know how many pets they have. You want that column to hold NULL instead of 0 so you can tell the difference between customers with no pets and customers whose pet ownership is unknown to you.

The concept of null gets a little funny when you use it in SQL calculations. Many programming languages use null as simply another kind of value. In Java, the following syntax evaluates to true when the variable is null and false when it is not:

```
str == null
```

The similar expression in SQL, COL = NULL, is neither true nor false—it is always NULL, no matter what the value of the COL column. The following query will therefor not act as you would expect:

```
SELECT TITLE FROM BOOK WHERE AUTHOR = NULL;
```

This query will always provide an empty result set, even when you have AUTHOR columns with NULL values. To test for "nullness", you should use the IS NULL and IS NOT NULL operators:

```
SELECT TITLE FROM BOOK WHERE AUTHOR IS NULL;
```

MySQL provides a special operator to use when you are not sure if you might be dealing with null values called the null-safe operator <=>. It will return true if both sides are null or if both sides are not null:

```
mysql> SELECT 1 <=> NULL, NULL <=> NULL, 1 <=> 1;
+------------+---------------+---------+
| 1 <=> NULL | NULL <=> NULL | 1 <=> 1 |
+------------+---------------+---------+
|          0 |             1 |       1 |
+------------+---------------+---------+
1 row in set (0.00 sec)
```

Membership Tests

Sometimes applications need to be able to check if a value is a member of a set of values or within a particular range. The IN operator helps with the former:

```
SELECT TITLE FROM BOOK WHERE AUTHOR IN ('Stephen King', 'Richard Bachman');
```

This query will return the titles of all books where Stephen King is the author.* Similarly, you can check for all books not written by him by using the NOT IN operator.

To determine if a value is in a particular range, an application should use the BETWEEN operator:

* Richard Bachman is a pseudonym used by Stephen King for some of his books.

Copyright © 2001 O'Reilly & Associates, Inc.

```
SELECT TITLE FROM BOOK WHERE BOOK_ID BETWEEN 1  AND 100;
```

Both of these simple examples could, of course, be replicated with the more basic operators. The Stephen King check, for example, could have been done by using the = operator and an OR. The check on book ID's also could have been done with an OR clause using the >= and <= or > and <. As your queries get more complex, however, these operators can help you build both readable and better performing queries than those you might create with the basic operators.

Pattern Matching

We provided a peek at ANSI SQL pattern matching earlier in the chapter with the query:

```
SELECT name FROM people WHERE name LIKE ‘Stac%’
```

Using the LIKE operator, we compared a column value (name) to an incomplete literal ('Stac%'). MySQL supports the ability to place special characters into string literals that match like wild cards. The '%' character, for example, matches any arbitrary number of characters, including no character at all. The above SELECT statement would therefore match 'Stacey', 'Stacie', 'Stacy', and even 'Stac'. The character '_' matches any single character. 'Stac_y' would match only 'Stacey'. 'Stac__' would match 'Stacie' and 'Stacey', but not 'Stacy' or 'Stac'.

Pattern matching expressions should never be used with the basic comparison operators. Instead, they should be used only with the LIKE and NOT LIKE operators. It is also important to remember that these comparisons are case-insensitive.

MySQL supports a non-ANSI kind of pattern matching that is actually much more powerful using the same kind of expressions that Perl programmers and *grep* users are accustomed to. MySQL refers to these as extended regular expressions. Instead of LIKE and NOT LIKE, these operators must be used with the REGEXP and NOT REGEXP operators.* Table 4-7 contains a list of the supported extended regular expression patterns.

Table 4-7. . MySQL Extended Regular Expressions

Pattern	Description	Examples
.	Matches any single character.	*Stac..* matches any value containing the characters "Stac" followed by two characters of any value.

* MySQL provides synonyms for these operators: RLIKE and NOT RLIKE.

Copyright © 2001 O'Reilly & Associates, Inc.

Table 4-7. . MySQL Extended Regular Expressions

Pattern	Description	Examples
[]	Matches any character in the brackets. You can also match a range of characters.	*[Ss]tacey* matches values containing both "Stacey" and "stacey". *[a-zA-Z]* matches values containing one instance of any character in the English (unaccented) portion of the Roman alphabet.
*	Matches zero or more instances of the character that precedes it.	*Ap*le* matches values containing "Aple", "Apple", "Appple", etc. *Los .*es* matches values containing the string "Los " and "es" with anything in between. *[0-9]** matches values containing any arbitrary number.
^	What follows must come at the beginning of the value.	*^Stacey* matches values that start with "Stacey".
$	What precedes it must end the value.	*cheese$* matches any value ending in the string "cheese"

You should note a couple of important facts about extended regular expressions. First, unlike basic pattern matching, MySQL extended regular expressions are case sensitive. They also do not require a match for the entire string. The pattern simply needs to occur somewhere within the value. Consider the following example:

```
mysql> SELECT * FROM BOOK;
+---------+-----------------------------------------+---------------+
| BOOK_ID | TITLE                                   | AUTHOR        |
+---------+-----------------------------------------+---------------+
|       1 | Database Programming with JDBC and Java | George Reese  |
|       2 | JavaServer Pages                        | Hans Bergsten |
|       3 | Java Distributed Computing              | Jim Farley    |
+---------+-----------------------------------------+---------------+
3 rows in set (0.01 sec)
```

In this table, we have three books from O'Reilly's Java series. The interesting thing about the Java series is that all books begin with or end with the word "Java". The first sample query checks for any titles `LIKE 'Java'`:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE LIKE 'Java';
Empty set (0.01 sec)
```

Because `LIKE` looks for an exact match of the pattern specified, no rows match—none of the titles are exactly 'Java'. To find out which books start with the word 'Java' using simple patterns, we need to add a '%' sign:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE LIKE 'Java%';
+----------------------------+
| TITLE                      |
+----------------------------+
| JavaServer Pages           |
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
| Java Distributed Computing |
+----------------------------+
2 rows in set (0.00 sec)
```

This query had two matches because only two of the books had titles that matched 'Java%' exactly. The extended regular expression matches, however, are not exact matches. They simply look for the expression anywhere within the compared value:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP 'Java';
+-----------------------------------------+
| TITLE                                   |
+-----------------------------------------+
| Database Programming with JDBC and Java |
| JavaServer Pages                        |
| Java Distributed Computing              |
+-----------------------------------------+
3 rows in set (0.06 sec)
```

By simple changing the operator from `LIKE` to `REGEXP`, we changed how it matches things. 'Java' appears somewhere in each of the titles, so the query returns all of the titles. In order to find only the titles that start with the word 'Java' using extended regular expressions, we need to specify that we are interested in the start:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP '^Java';
+----------------------------+
| TITLE                      |
+----------------------------+
| JavaServer Pages           |
| Java Distributed Computing |
+----------------------------+
2 rows in set (0.01 sec)
```

The same thing applies to finding 'Java' at the end:

```
mysql> SELECT TITLE FROM BOOK WHERE TITLE REGEXP 'Java$';
+-----------------------------------------+
| TITLE                                   |
+-----------------------------------------+
| Database Programming with JDBC and Java |
+-----------------------------------------+
1 row in set (0.00 sec)
```

The extended regular expression syntax is definitely much more complex than the simple pattern matching of ANSI SQL. In addition to the burden of extra complexity, you also should consider the fact that MySQL extended regular expressions do not work in most other databases. When you need complex pattern matching, however, they provide you with power that is simply unsupportable by simple pattern matching.

Copyright © 2001 O'Reilly & Associates, Inc.

Advanced Features

Using the SQL presented thus far in this chapter should handle 90% of your database programming needs. On occasion, however, you will need some extra power not available in the basic SQL functionality. We close out the chapter with a discussion of a few of these features.

Transactions

MySQL recently introduced transactions and thus SQL for executing statements in a transactional context. By default, MySQL is in a state called autocommit. Autocommit mode means that any SQL you send to MySQL is executed immediately. In some cases, however, you may want to execute two or more SQL statements together as a single unit of work.

A transfer between to bank accounts is the perfect example of such a transaction. The bank system needs to make sure that the debit of the first account and the credit to the second account occur as a single unit of work. If they were treated separately, the server could in theory crash between the debit and the credit. The result would be that you would lose that money!

By making sure the two statements occur as a single unit of work, transactions ensure that the first statement can be "rolled back" in the event the second statement fails. To use transactions in MySQL, you first need to create a table using a transactional table type such as BDB or InnoDB. If your MySQL install was not compiled with support for these table types, you cannot use transactions. The SQL to create a transactional table is:

```
CREATE TABLE ACCOUNT (
    ACCOUNT_ID BIGINT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    BALANCE    DOUBLE)
TYPE = BDB;
```

For a transaction against a transactional table to work, you need to turn off autocommit. You can do this through the command:

```
SET AUTOCOMMIT=0;
```

Now you are ready to begin using MySQL transactions. Transactions start with the `BEGIN` command:

```
BEGIN;
```

Your mysql client is now in a transactional context with respect to the server. Any change you make to a transactional table will not be made permanent until you commit it. Changes to non-transactional tables, however, will still take place immediately. In the case of the account transfer, we issue the following statements:

Copyright © 2001 O’Reilly & Associates, Inc.

```
UPDATE ACCOUNT SET BALANCE = 50.25 WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = 100.25 WHERE ACCOUNT_ID = 2;
```

Once done with any changes, you complete the transaction using the `COMMIT` command:

```
COMMIT;
```

The true advantage of transactions, of course, comes into play should an error occur in executing the second statement. To abort the entire transaction before a commit, issue the `ROLLBACK` command:

```
ROLLBACK;
```

Of course, it would be useful if MySQL performed the actual math. It can do just that so long as you store the values you want with a SELECT call:

```
SELECT @FIRST := BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 1;
SELECT @SECOND := BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 2;
UPDATE ACCOUNT SET BALANCE = @FIRST - 25.00 WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = @SECOND + 25.00 WHERE ACCOUNT_ID = 2;
```

In addition to issuing the `COMMIT` command, a handful of other commands will automatically end any current transaction as if a `COMMIT` had been issued. These commands are:

- `ALTER TABLE`
- `BEGIN`
- `CREATE INDEX`
- `DROP DATABASE`
- `DROP TABLE`
- `RENAME TABLE`
- `TRUNCATE`

Chapter 9 covers some of the more intricate details of using transactions in data-base applications.

Table Locking

Table locking is the poor man's transaction. In short, MySQL lets you lock down any table so that only a single client can use it. Unlike transactions, you are not limited by the type of the table. You cannot, however, rollback any actions taken against a locked table.

Locking has two basic functions:

Copyright © 2001 O'Reilly & Associates, Inc.

1. To enable multiple statements to execute against a single table as one unit of work.
2. To enable multiple updates to occur faster since under some circumstances.

MySQL supports three kinds of locks, read, read local, and write. Both kinds of read locks lock the table for reading by a client and all other clients. As long as the lock is in place, no one can write to the locked tables. Read and read local locks differ in that read local allows a client to execute non-conflicting INSERT statements so long as no changes to the MySQL files from outside of MySQL will occur while the lock as held. If changes might occur by agents outside of MySQL, then a read lock is required.

A write lock locks the specified tables against all access, read or write, by any other client. To lock a table, use the following command:

```
LOCK TABLES ACCOUNT WRITE;
```

Now that the `ACCOUNT` table is locked, you can read from it and then modify the data behind it and be certain that no one else will change the data you read between your read and write operations.

```
SELECT @BAL:=BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = 1;
UPDATE ACCOUNT SET BALANCE = @BAL * 0.03 WHERE ACCOUNT_ID = 1;
```

Finally, you need to release the locks:

```
UNLOCK TABLES;
```

Functions

Functions in SQL are similar to functions in other programming languages like C and Perl. The function takes zero or more arguments and returns some value. For example, the function SQRT(16) returns 4. Within a MySQL `SELECT` statement, functions may be used in either of two places:

As a value to be retrieved

This form involves a function in the place of a column in the list of columns to be retrieved. The return value of the function, evaluated for each selected row, is part of the returned result set as if it were a column in the database.* For example:

```
# Select the name of each event as well as the date of the event
# formatted in a human-readable form for all events more
# recent than the given time. The FROM_UnixTIME() function
# transforms a standard Unix time value into a human
# readable form.
SELECT name, FROM_UnixTIME(date)
```

* You can use the aliasing you used earlier in the chapter to give the resulting columns "friendly" names.

Copyright © 2001 O’Reilly & Associates, Inc.

```
FROM events
WHERE time > 90534323

# Select the title of a paper, the full text of the paper,
# and the length (in bytes) of the full text for all
# papers authored by Stacie Sheldon.
# The LENGTH() function returns the character length of
# a given string.
SELECT title, text, LENGTH(text)
FROM papers
WHERE author = 'Stacie Sheldon'
```

As part of a WHERE clause

This form involves a function in the place of a constant when evaluating a WHERE clause. The value of the function is used for comparison for each row of the table. For example:

```
# Randomly select the name of an entry from a pool of 35
# entries. The RAND() function generates a random number
# between 0 and 1 (multiplied by 34 to make it between 0
# and 34 and incremented by 1 to make it between 1 and
# 35). The ROUND() function returns the given number
# rounded to the nearest integer, resulting in a whole
# number between 1 and 35, which should match one of
# the ID numbers in the table.
SELECT name
FROM entries
WHERE id = ROUND( (RAND()*34) + 1 )

# You may use functions in both the value list and the
# WHERE clause. This example selects the name and date
# of each event less than a day old. The UNIX_TIMESTAMP()
# function, with no arguments, returns the current time
# in Unix format.
SELECT name, FROM_UnixTIME(date)
FROM events
WHERE time > (Unix_TIMESTAMP() - (60 * 60 * 24) )

# You may also use the value of a table field within
# a function. This example returns the name of anyone
# who used their name as their password. The ENCRYPT()
# function returns a Unix password-style encryption
# of the given string using the supplied 2-character salt.
# The LEFT() function returns the left-most n characters
# of the given string.
SELECT name
FROM people
WHERE password = ENCRYPT(name, LEFT(name, 2))
```

Date Functions

The most common functions you will use are likely to be the MySQL functions that enable you to manipulate dates. You already saw some of these functions above

Copyright © 2001 O'Reilly & Associates, Inc.

for translating a UNIX-style date into a human-readable form of the date. MySQL, of course, provides more powerful functions for doing things like calculating the time between two dates:

```
SELECT TO_DAYS(NOW()) - TO_DAYS('2000-12-31');
```

This example provides the number of days that have passed in this millennium. The `NOW()` function, of course, returns the `DATETIME` representing the moment in time when the command is executed. Less obviously, the `TO_DAYS()` function returns the number of days since the year 1 B.C.* represented by the specified `DATE` or `DATETIME`.

Not everyone likes to see dates formatted the way MySQL provides them by default. Fortunately, MySQL lets you format dates to your own liking using the DATE_FORMAT function. It takes a DATE or DATETIME and a format string indicating how you want the date formatted:

```
mysql> SELECT DATE_FORMAT('1969-02-17', '%W, %M %D, %Y');
+-------------------------------------------+
| DATE_FORMAT('1969-02-17', '%W, %M %D, %Y') |
+-------------------------------------------+
| Monday, February 17th, 1969               |
+-------------------------------------------+
1 row in set (0.39 sec)
```

Chapter 16 contains a full list of valid tokens for the `DATE_FORMAT()` function.

String Functions

In addition to date functions, you are likely to make use of string functions. We saw one such function above: the `LENGTH()` function. This function naturally provides the number of characters in the string. The most common string function you are likely to use, however, is the `TRIM()` function that helps remove spaces from columns that may be padded with spaces.

One interesting function is the SOUNDEX() function. It translates a word into its soundex representation. The soundex representation is a way of representing the sound of a string so that you can compare two strings to see if they sound alike:

```
mysql> SELECT SOUNDEX('too');
+----------------+
| SOUNDEX('too') |
+----------------+
| T000           |
+----------------+
1 row in set (0.42 sec)
```

* MySQL is actually incapable of representing this date. Valid date ranges in MySQL are from the January 1, 1000 to December 31, 9999. There is no support in MySQL for alternative calendaring systems.

Copyright © 2001 O'Reilly & Associates, Inc.

```
mysql> SELECT SOUNDEX('two');
+----------------+
| soundex('two') |
+----------------+
| T000           |
+----------------+
1 row in set (0.00 sec)
```

Outer Joins

MySQL supports a more powerful joining than the simple inner joins we have used so far. Specifically, MySQL supports something called a *left outer join* (also known as simply *outer join*). This type of join is similar to an inner join, except that it includes data in the first column named that does not match any in the second column. If you remember our author and book tables from earlier in the chapter, you will remember that our join would not list any authors who did not have a book in our database. It is common that you may want to show entries from one table that have no corresponding data in the table to which you are joining. That is where an outer join comes into play:

```
SELECT book.title, author.name
FROM author
LEFT JOIN book ON book.author = author.id
```

Note that a outer join uses the keyword ON instead of WHERE. The results of our query would look like this:

```
+----------------+----------------+
| book.title     | author.name    |
+----------------+----------------+
| The Green Mile | Stephen King   |
| Guards, Guards!| Terry Pratchett|
| Imzadi         | Peter David    |
| Gold           | Isaac Asimov   |
| Howling Mad    | Peter David    |
| NULL           | Neil Gaiman    |
+----------------+----------------+
```

MySQL takes this concept one step further through the use of a natural outer join. A natural outer join will combine the rows from two tables where the two tables have identical column names with identical types and the values in those columns are identical:

```
SELECT my_prod.name
FROM my_prod
NATURAL LEFT JOIN their_prod
```

Copyright © 2001 O’Reilly & Associates, Inc.

Batch Processing

Batch loading is the act of loading a lot of data into or pulling a lot of data out of MySQL all at once. MySQL supports two manners of batch loading.

Command Line Loads

The simplest kind of batch load is where you stick all of your SQL commands in a file and then send the contents of that file to MySQL:

```
mysql -h somehost -u uid -p < filename
```

In other words, you are using the command line to pipe the SQL commands into a *mysql* command line. The examples that come with this book contain several SQL command files that you can load into MySQL in this manner before you run the examples.

The LOAD Command

The LOAD command enables you to load data from a file containing only data (no SQL commands). For example, if you had a file containing the names of all the books in your collection with one book on each line and the title and author separated by a tab, you could use the following command to load that data into your book table:

```
LOAD DATA LOCAL INFILE 'books.dat' INTO TABLE BOOK;
```

This command assumes that the file *books.dat* has one line for each database record to be inserted. It further assumes that there is a value for every column in the table or `\N` for null values. So, if the `BOOK` table has 3 columns, then each line of books.dat should have three tab-separated values.

The `LOCAL`* keyword tells the mysql command line to look for the file on the same machine as the client. Without it, it will try to look for the file on the server. Of course, if you are trying to load something on the server, you need to be granted the special file privilege. Finally, keep in mind that non-local loads refer to files relative to the installation directory of MySQL.

If you have a comma-separated value file like an Excel file, you can change the delimiter of the LOAD command:

```
LOAD DATA LOCAL INFILE 'books.dat'
INTO TABLE BOOK
FIELDS TERMINATED BY ',';
```

If a file contains values that would cause duplicate records in the database, you can use the `REPLACE` and `IGNORE` keywords to dictate the correct behavior.

* Reading from files local to the client is available only to users of MySQL 3.22.15 and later.

Copyright © 2001 O'Reilly & Associates, Inc.

`REPLACE` will cause the values from the file to replace the ones in the database, where the `IGNORE` keyword will cause the duplicate values to be ignored. The default behavior is to ignore duplicates.

Pulling Data from MySQL

Finally, MySQL provides a tool for pulling the results of a `SELECT` from the database and sticking it into a file:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS TERMINATED BY ','
FROM BOOK;
```

This query puts all rows in the `BOOK` table into the file *books.dat*. You could then use this file to load into an Excel spreadsheet or another database. Because this file is created on the server, it is created relative to the base directory for the database in use. On an Mac OS X basic installation, for example, this file is created as */usr/local/var/test/test.dat*.

A more complex version of this command enables you to put quotes (or any other character) around fields:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS ENCLOSED BY '"' TERMINATED BY ','
FROM BOOK;
```

Of course, you probably want only the string fields (`CHAR`, `VARCHAR`, etc.) enclosed in quotes. You can accomplish this by adding the `OPTIONALLY` keyword:

```
SELECT * INTO OUTFILE 'books.dat'
FIELDS OPTIONALLY ENCLOSED BY '"' TERMINATED BY ','
FROM BOOK;
```

Chapter 16 contains a full range of options for loading and extracting data from MySQL.

Copyright © 2001 O'Reilly & Associates, Inc.