

1

Introducing the MySQL Relational Database Management System

In the world of online auction houses, instant mortgages, worldwide reservations, global communication, and overnight deliveries, it's not surprising that even the least technically savvy individuals in our culture are, to some degree, familiar with the concept of a database. As anyone who works with data knows, databases form the backbone for this age of information, and access to those databases can determine one's ability to perform critical tasks effectively and efficiently. To meet the ever-increasing demands for information, programmers are continuously building bigger and better applications that can access and modify data stored in various database systems. Yet in order to create these applications, programmers must have some knowledge of the systems that contain the needed data.

Over the years, as the demands for information have grown, so too have the database systems that have attempted to meet these demands. However, along with this evolution, we have seen an increase in the costs associated with storing data as well as an increase in the demand for products that can run on multiple platforms and can be optimized based on the needs of specific types of organizations. In response to this changing climate, MySQL has emerged as the most popular open-source database management system (DBMS) in the world. Consequently, organizations everywhere are jumping on the MySQL bandwagon, increasing the demand for those who know how to use MySQL to manage data and those who know how to create applications that can access data in MySQL databases.

In learning to use MySQL, whether to work directly in the MySQL environment or create data-driven applications, an individual should have a thorough understanding of how MySQL, as a relational database management system (RDBMS), allows you to manage data and support applications that rely on access to MySQL data. To this end, this chapter introduces you to MySQL and provides you with an overview of databases, RDBMSs, SQL, and data-driven applications. By the end of the chapter, you will understand the following concepts:

- ❑ What a relational database is, how it differs from other types of databases, and how it relates to a database management system.
- ❑ The programming language SQL, how to interpret SQL syntax, how to create SQL statements, and how SQL is implemented in MySQL.
- ❑ How applications can use a host programming language, a MySQL application programming interface (API), and SQL statements to access information in a MySQL database.

Databases and Database Management Systems

Databases and database management systems have become the backbone of most Web-related applications as well as an assortment of other types of applications and systems that rely on data stores to support dynamic information needs. Without the availability of flexible, scalable data sources, many organizations would come to a standstill, their ability to provide services, sell goods, rent movies, process orders, issue forms, lend books, plan events, admit patients, and book reservations undermined by the inability to access the data essential to conducting business. As a result, few lives are unaffected by the use of databases in one form or another, and their ubiquitous application in your everyday existence can only be expected to grow.

What Is a Database?

Over the years, the term *database* has been used to describe an assortment of products and systems that have included anything from a collection of files to a complex structure made up of user interfaces, data storage and access mechanisms, and client/server technologies. For example, a small company might store payroll records in individual files, while a regional electric company uses an integrated system to maintain records on all its customers; generate electric bills to those customers; and create reports that define power usage patterns, profit and loss statements, or changes in customer demographics. In both cases, the organizations might refer to each of their systems as databases.

Despite how a database is used, the amount of data that it stores, or the complexity of the data, a number of common elements define what a database is. At its simplest, a database is a collection of data that is usually related in some fashion. For instance, a database that a bookstore uses might contain information about authors, book titles, and publishers. Yet a database is more than simply a collection of related data. The data must be organized and classified in a structured format that is described by *metadata*, which is data that describes the data being stored. In other words, the metadata defines how the data is stored within the database. Together, the data and the metadata provide an environment that logically organizes the data in a way that can be efficiently maintained and accessed.

One way to better understand what constitutes a database is to use the analogy of a telephone book. A phone book contains the names, addresses, and phone numbers of most of the telephone customers in a particular town or region. If you think of that phone book as a database, you find a set of related data (the names, addresses, and phone numbers of the telephone customers) and you find a structured format (the metadata) that is defined by the way that the pages are bound and by how the information is organized. The phone book provides a system that allows you easy and efficient access to the data contained in its pages. Without the structure of the phone book, it would be next to impossible to locate specific customer data.

In the same way that a phone book provides structure to the customer information, the metadata of a database defines a structure that organizes data logically within that structure. However, not all database structures are the same, and through the years, a number of different data models have emerged. Of these various models, the three most commonly implemented are the hierarchical, network, and relational models.

The Hierarchical Model

In the early days of database design, one of the first data models to emerge was the *hierarchical* model. This model provides a simple structure in which individual records are organized in a parent-child relationship to form an inverted tree. The tree creates a hierarchical structure in which data is decomposed into logical categories and subcategories that use records to represent the logical units of data.

Take a look at an example to help illustrate how to structure a hierarchical database. Suppose you're working with a database that stores parts information for a company that manufactures wind generators. Each model of wind generator is associated with a parent record. The parts that make up that model are then divided into categories that become child records of the model's parent record, as shown in Figure 1-1. In this case, the parent record — Wind Generator Number 101 — is linked to three child records: Tower assemblies, Power assemblies, and Rotor assemblies. The child records are then divided into subcategories that are assigned their own child records. As a result, the original child records now act as parent records as well. For example, the Tower assemblies record is a parent of the Towers record but a child of the Wind Generator Number 101 record.

As you can see in the figure, a parent record can be associated with multiple child records, but a child record can be associated with only one parent record. This structure is similar to what you might see in a directory structure viewed through a graphical user interface (GUI) file management application, such as Windows Explorer. At the top of the directory structure would be Wind Generator Number 101. Beneath this, would be Tower assemblies, Power assemblies, and Rotor assemblies, each with their own set of subdirectories.

After its introduction, the hierarchical data model achieved a great deal of success. One of the most popular implementations of this model is found in IBM's Information Management System (IMS), which was introduced in the 1960s and is still widely used on IBM mainframe computers.

However, despite the popularity of the hierarchical model, it is unsuitable for many of today's applications. Inherent in the simplicity of the parent-child organization is a rigid structure that results in a cumbersome navigation process that requires application developers to programmatically navigate through the connected records to find the necessary information. Records must be accessed one at a time by moving up and down through the hierarchical levels, which often made modifications to the database and application a complicated, time-consuming process. In addition, the hierarchical structure cannot support complex relationships between records. For example, if you return to the wind generator database example, you discover that Figure 1-1 doesn't show a record for the belts used to connect the generators to the shafts. If you were to create a child record for belts, should it be added under the Generators record or the Shaft assemblies record? The hierarchical design makes it difficult to fully represent the relationship that exists between the belts and the generators and shafts. Indeed, a child record can be associated with only one parent record.

Chapter 1

Even with the limitations of the hierarchical model, a large amount of data is still being stored in hierarchical databases, and many management systems have found ways to work around some of these limitations. In addition, this is the type of system used primarily for file management systems associated with operating systems because it allows users to go directly where they need to go to find a file, rather than having to iterate through a lot of nodes. As a result, the hierarchical database probably isn't going anywhere anytime soon.

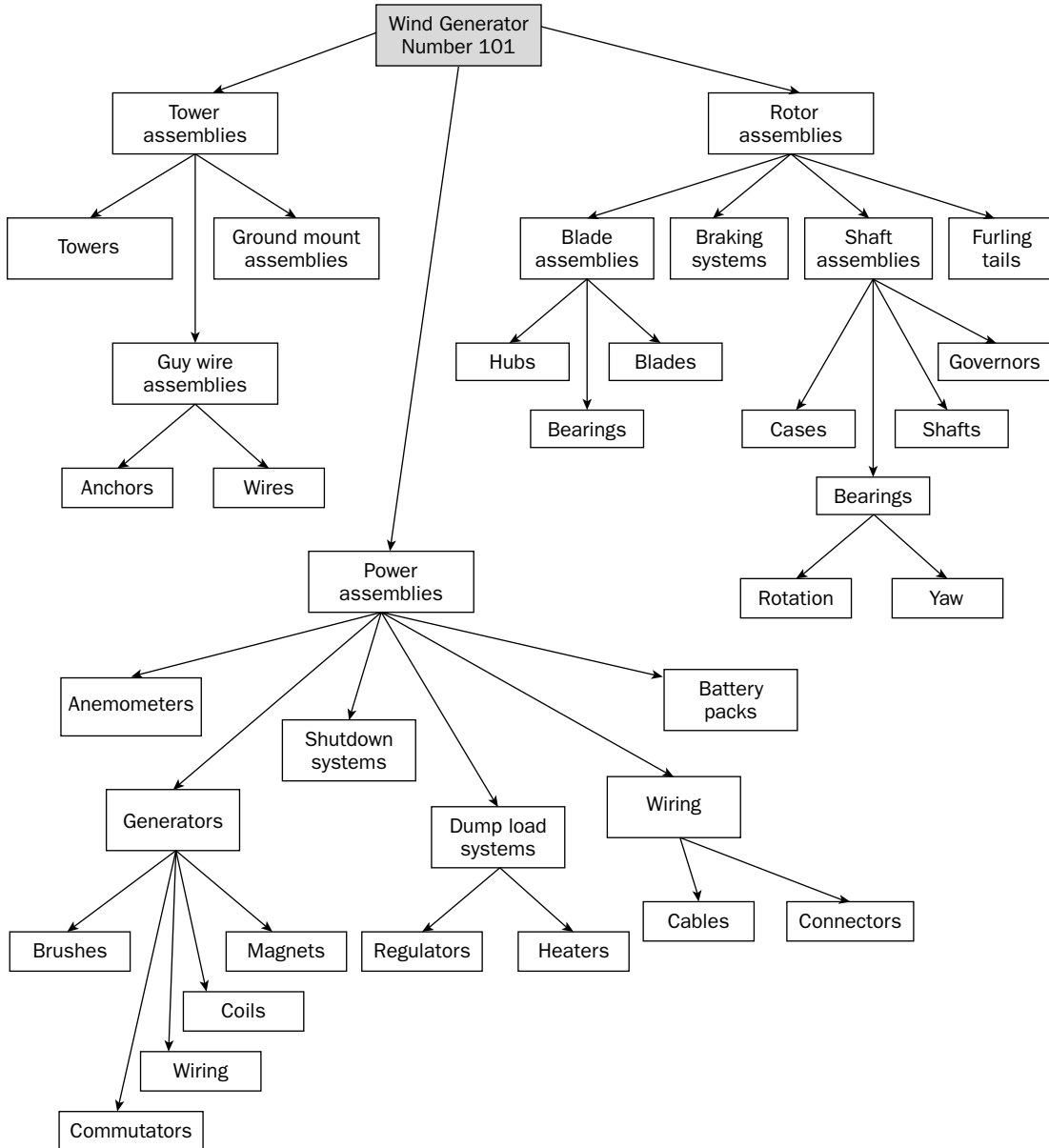


Figure 1-1

The Network Model

To work around the limitations of hierarchical databases, a new model of database design, built upon the hierarchical model, emerged in the 1970s. The *network* model enhanced the hierarchical model by allowing records to participate in multiple parent-child relationships. For example, suppose the wind generator database must also store data about employees, customers, and orders. A hierarchical database might contain four parent records: Orders, Employees, Customers, and Wind generators. Each of these records would then contain the necessary child records to support this structure, as shown in Figure 1-2.

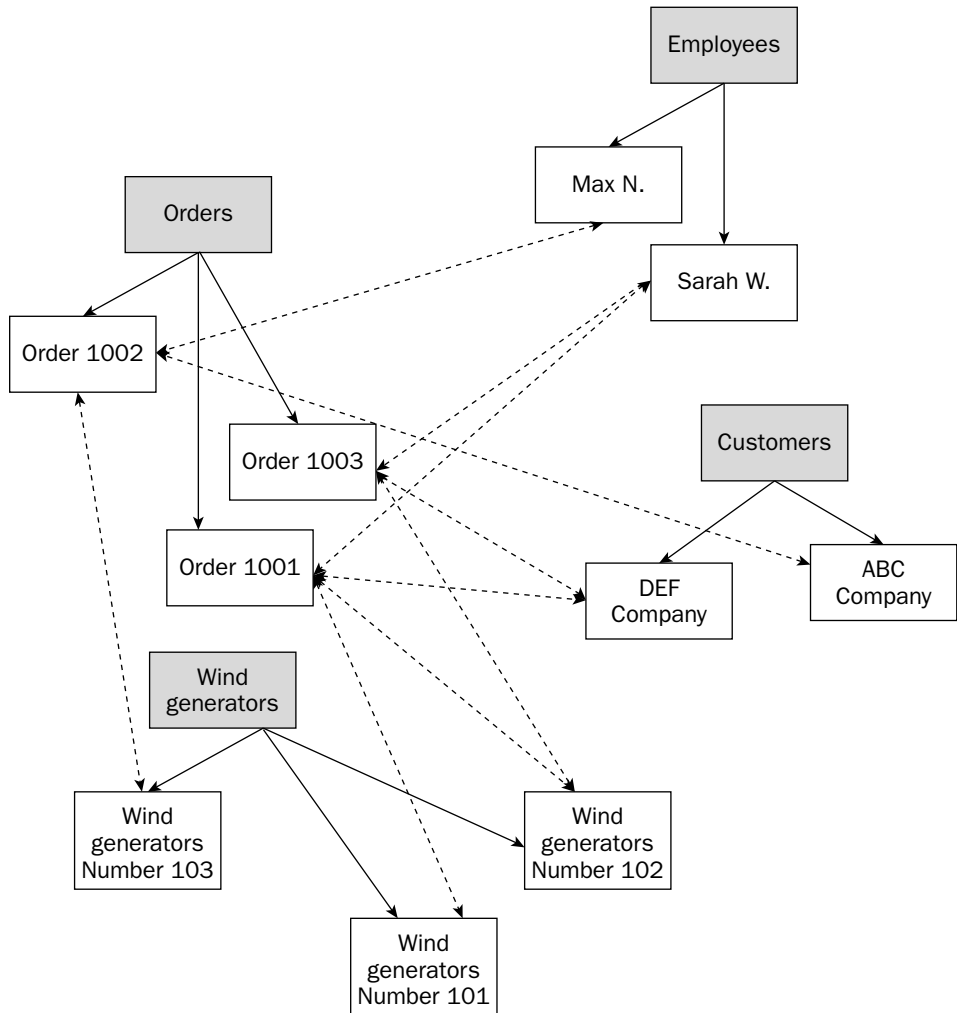


Figure 1-2

If each of these categories operated without interaction with each other, then the need for the network model would be minimal. However, if you consider the fact that each order is related to the employee who took the order, the customer who bought the order, and the wind generator model that was purchased, you can see that the hierarchical model is inadequate to support the complex relationships that exist between records. For example, Sarah W. took Order 1001 for the DEF Company. The company bought two wind generators: models 101 and 102. As a result, the Order 1001 record is related to the Sarah W. record, the DEF Company record, the Wind Generator Number 101 record, and the Wind Generator Number 102 record.

The network model still has many of the disadvantages of the hierarchical model, but it provides far more flexibility in allowing programmers to navigate through records. Despite the flexibility, developers must still program record navigation within the application. In addition, any changes to the database or application can result in complicated updates. A database must be well planned in advance, taking into account record navigation at the application level.

The Relational Model

Because of the limitations of the hierarchical and network models, a new model began gaining momentum in the late 1970s, and by the end of the 1980s, emerged as the standard for the next generation of databases. The *relational* data model represents a radical departure from the rigid structures of the hierarchical and network models. Applications accessing a hierarchical database rely on a defined implementation of that database, and the database structure must be hard-coded into the application's programming language. If the database changes, the application must change.

However, a relational database is independent of the application. It's possible to modify the database design without affecting the application because the relational model replaces the parent-child framework with a structure based on rows and columns that form tables of related data. As a result, you can define complex relationships between the tables, without the restrictions of earlier models.

For example, suppose you want to change the original wind generator database that you saw in Figure 1-1 to a relational database. The database might include a table for the individual parts and a table of the individual categories of parts, as shown in Figure 1-3. As you can see from the illustration, the Parts table includes a list of parts from different areas of the wind generator. The table could contain every part for the entire wind generator, with each row in the table representing a specific part, just as a record in the hierarchical database represents a specific part. For example, the guy wire assembly (in the first row of the Parts table) is a component of the tower assembly, and the generator (in the fifth row) is a component of the power assembly.

Each row in the Parts table represents one part. The part is assigned a unique part ID, a name, and a reference to the category to which it belongs. The Categories table lists each category. Note that the last column in the Parts table references the first column in the Categories table. A relationship exists between these two tables. For instance, the brushes product has been assigned a PartID value of 1004. If you look in the CatID column for this product, you see that it contains a value of 504. If you now look at the Categories table, you find that 504 refers to the Generator category, which is itself a part. Because of this structure, programmers are less restricted when moving through data, resulting in applications that can be more flexible when retrieving information and databases that can better accommodate change after the applications have been written.

Don't be concerned if you do not fully grasp the concepts of tables, rows, and columns or the relationships between the tables. Chapter 4 discusses the relational model in greater detail.

Parts			Categories		
PartID	PartName	CatID	CatID	CatName	Parent
1001	Guy wire assembly	503	501	Wind Generator 101	NULL
1002	Magnet	504	502	Rotor assembly	501
1003	Regulator	505	503	Tower assembly	501
1004	Brushes	504	504	Generator	506
1005	Generator	506	505	Dump load system	506
1006	Dump load system	506	506	Power assembly	501
1007	Power assembly	501			
1008	Tower assembly	501			
1009	Rotor assembly	501			

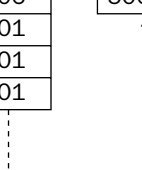


Figure 1-3

As the popularity of the relational model has grown, so too has the number of database products that use this model to store and manage data. Included in the family of relational products are DB2, Oracle, SQL Server, and of course, MySQL. However, a relational database alone is not enough to provide the type of data management, storage, connectivity, security, analysis, and manipulation that is required of a dynamic information store. For this, you need a complete management system that works in conjunction with the relational database to provide the full spectrum of database services.

Database Management Systems

Most databases rely on a database management system to manage the data stored within the system's databases and to make the data available to users who need access to specific types of information. A DBMS is made up of a comprehensive set of server and client tools that support various administrative and data-related tasks. For example, most DBMSs provide some type of client tool that allows you to interact directly with the data stored in a database.

At the very least, a DBMS must store data and allow data to be retrieved and modified in a way that protects the data against operations that could corrupt or insert inconsistencies into the data. However, most systems provide many more capabilities. In general, nearly any comprehensive DBMS supports the following types of functionality:

- Managing storage
- Maintaining security
- Maintaining metadata
- Managing transactions
- Supporting connectivity
- Optimizing performance
- Providing back-up and recovery mechanisms
- Processing requests for data retrieval and modification

The extent to which a DBMS supports a particular functionality and the exact nature in which that functionality is implemented is specific to the DBMS. For any one system, you must refer to the product documentation to determine what and how specific functionality is implemented.

The MySQL RDBMS

As database models evolved, so too did the DBMS products that supported the various types of databases. It's not surprising, then, that if there are DBMSs, there are RDBMSs. MySQL is such a system, as are Oracle, DB2, SQL Server, and PostgreSQL. These products, like any DBMS, allow you to access and manipulate data within their databases, protect the data from corruption and inconsistencies, and maintain the metadata necessary to define the data being stored. The primary difference, then, between a DBMS and a RDBMS is that the latter is specific to relational databases. It supports not only the storage of data in table-like structures, but also the relationships between those tables.

Emerging as a major player in the RDBMS market is MySQL. As with other RDBMS products, MySQL provides you with a rich set of features that support a secure environment for storing, maintaining, and accessing data. MySQL is a fast, reliable, scalable alternative to many of the commercial RDBMSs available today. The following list provides an overview of the important features found in MySQL:

- ❑ **Scalability:** MySQL can handle large databases, which has been demonstrated by its implementation in organizations such as Yahoo!, Cox Communications, Google, Cisco, Texas Instruments, UPS, Sabre Holdings, HP, and the Associated Press. Even NASA and the US Census Bureau have implemented MySQL solutions. According to the MySQL product documentation, some of the databases used by MySQL AB, the company that created MySQL, contain 50 million records, and some MySQL users report that their databases contain 60,000 tables and 5 billion rows.
- ❑ **Portability:** MySQL runs on an assortment of operating systems, including Unix, Linux, Windows, QS/2, Solaris, and MacOS. MySQL can also run on different architectures, ranging from low-end PCs to high-end mainframes.
- ❑ **Connectivity:** MySQL is fully networked and supports TCP/IP sockets, Unix sockets, and named pipes. In addition, MySQL can be accessed from anywhere on the Internet, and multiple users can access a MySQL database simultaneously. MySQL also provides an assortment of application programming interfaces (APIs) to support connectivity from applications written in such languages as C, C++, Perl, PHP, Java, and Python.
- ❑ **Security:** MySQL includes a powerful system to control access to data. The system uses a host- and user-based structure that controls who can access specific information and the level of access to that information. MySQL also supports the Secure Sockets Layer (SSL) protocol in order to allow encrypted connections.
- ❑ **Speed:** MySQL was developed with speed in mind. The amount of time it takes a MySQL database to respond to a request for data is as fast as or faster than many commercial RDBMSs. The MySQL Web site (www.mysql.com) provides the results of numerous benchmark tests that demonstrate the fast results you receive with a MySQL implementation.
- ❑ **Ease of use:** MySQL is simple to install and implement. A user can have a MySQL installation up and running within minutes after downloading the files. Even at an administrative level, MySQL is relatively easy to optimize, especially compared to other RDBMS products.
- ❑ **Open-source code:** MySQL AB makes the MySQL source code available to everyone to download and use. The open-source philosophy allows a global audience to participate in the review, testing, and development of code. (See the Open-Source Movement section below for information about open-source technology.)

Introducing the MySQL Relational Database Management System

As you can see, MySQL can provide you with a fast, reliable solution to your database needs. Not only is it easy to use and implement, it offers the advantages and flexibility of an open-source technology. You can download the MySQL distribution files directly from the MySQL Web site, and start using the product immediately.

The Open-Source Movement

One of the most distinctive features of MySQL compared to RDBMSs such as Oracle and DB2 is that MySQL is an open-source application. As a result, the MySQL source code is available for anyone to use and modify, within the constraints of the GNU General Public License (GPL), an open-source licensing structure that supports the distribution of free software. (GNU, pronounced *Guh-New*, is an acronym for “GNU’s Not Unix.” GNU is an operating system based on the Linux kernel.)

For specific information about the most current MySQL licensing structure, visit the MySQL site at www.mysql.com. For information about the GNU GPL, visit the GNU licensing site at www.gnu.org/licenses.

The open-source nature of MySQL is part of a worldwide movement that promotes the free access of application source code. As a result, users are allowed to download and use open-source applications for free. One of the most well known examples of this technology is the Linux operating system, which has been instrumental in unifying the open-source community and promoting a wider base of users and developers who test and contribute to the operating system’s development. The same is true of MySQL, which is reported to be the most popular open-source RDBMS in the world. As an open-source application, developers everywhere contribute to the development process, and millions of users test new versions of the application as it is being developed.

As applications such as MySQL and Linux continue to see a steady growth in their global user base, so too does the acceptance of the open-source philosophy, evidenced by the increasing number of other types of applications and technologies that now participate in the open-source movement, providing a richer user experience, a more robust developer environment, and a wider spectrum of options for everyone.

The SQL Framework

Soon after the relational data model appeared on the database scene, research began on the development of relational databases. From this research came the realization that traditional programming languages, such as COBOL or Fortran, were not suited to implementing these types of databases and that a special language was needed. Out of these beginnings came SQL, a database-specific language that has become the definitive language of relational databases and that, as a result, has seen widespread implementation and usage, regardless of products, platforms, or operating system environments.

There is some debate about what SQL stands for and how to pronounce it. In some sources, you see SQL defined as an acronym that means Structured Query Language, yet other sources treat SQL as simply the three letters that stand for the language. The American National Standards Institute (ANSI), which published the SQL:2003 standard, makes no mention of “structured query language” and treats SQL simply as the three letters. As a result, no definite resource says that SQL stands for Structured Query Language, despite the fact that many publications define it this way.

Another area of debate that surrounds SQL is whether to pronounce it one letter at a time, as in “S-Q-L,” or to pronounce it as a word, as in “sequel.” This is why some publications, when preceding SQL with an article, use the word an (“an SQL database”) and some use the word a (“a SQL database”). With regard to this particular issue, the SQL:2003 standard prefers “S-Q-L,” so that is the convention used in this book.

What is SQL?

SQL is, above all else, a computer language used to manage and interact with data in a relational database. SQL is the most universally implemented database language in use, and it has become the standard language for database management. SQL works in conjunction with a RDBMS to define the structure of the database, store data in that database, manipulate the data, retrieve the data, control access to the data, and ensure the integrity of the data. Although other languages have been developed to implement the relational model, SQL has emerged as the clear winner.

Nearly all RDBMSs implement some form of SQL in order to manage their relational database. This is true not only for MySQL, but also for SQL Server, DB2, Oracle, PostgreSQL, and all the major players in the world of RDBMSs. However, do not confuse SQL with the programming languages used to develop the RDBMS. For example, MySQL is built with C and C++. The functions that such an application performs in order to support connectivity, provide APIs, enable network access, or interact with client tools are carried out at the C and C++ programming level. The primary purpose of SQL is to allow the RDBMS to interact with the data. The C and C++ environment provides the structure that houses the SQL environment, which itself allows you to define and interact with the data. In other words, the RDBMS facilitates the ability of SQL to manage data.

Figure 1-4 illustrates how SQL interacts with the MySQL RDBMS. In this figure, MySQL is running as a server on a specific platform such as Linux or Unix. The database, stored either internally or externally, hosts the actual database files. Although the RDBMS facilitates (through the C/C++ applications) the creation and maintenance of the database and the data within the database, SQL actually creates and maintains the database and data.

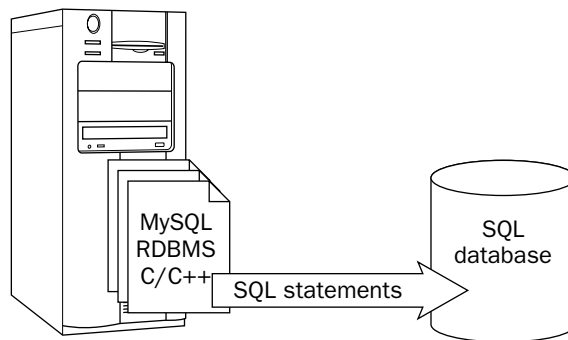


Figure 1-4

SQL, then, is a standardized language, not a stand-alone product, such as MySQL. SQL relies on the interaction with a RDBMS in order to manage data. You cannot develop an SQL-based application, although you can build an application that connects to a database managed by a RDBMS and then sends SQL statements to the database in order to request and modify data. (You learn more about how SQL fits into application development later in the chapter, in the section Data-Driven Applications.) However, despite the inability of SQL to stand on its own, it remains the foundation of most relational databases; therefore, anyone who creates applications that interact with an SQL database should have a basic understanding of SQL.

A Brief History of SQL

After the relational model was introduced to the database development community in the early 1970s, IBM began researching ways to implement that model. IBM's research, referred to as the System/R project, resulted in a prototype of the first RDBMS. As part of the System/R project, IBM produced the first incarnation of a relational database language, which was known Structured English Query Language (SEQUEL). Over the next few years, IBM updated the prototype and released SEQUEL/2, which was later renamed to SQL.

In the late 1970s, IBM released System R to a number of its customers for evaluation. The development and release of System R brought with it increased attention to relational databases, RDBMSs, and SQL, and reconfirmed to the public IBM's commitment to the relational model. Soon a group of engineers formed Relational Software, Inc., a company whose primary goal was to develop a RDBMS system based on SQL. Before long, the company released its own product — Oracle — the first commercial RDBMS to hit the market. It wasn't until 1981 that IBM released their first commercial RDBMS — SQL/DS.

The ANSI Standard

By the mid-1980s, relational databases and SQL had become an industry standard. During this time, the performance of RDBMSs had improved dramatically, and other companies were investing into the relational technologies, either releasing or preparing to release their own SQL-based RDBMSs. However, as SQL became more widely implemented, so too did the need to standardize the language across various products. In an attempt to achieve this goal, ANSI released the first published SQL standard (SQL-86) in 1986, giving SQL official status in the software development industry.

ANSI updated the standard in 1989 (SQL-89) and again in 1992 (SQL-92). SQL-92 represented a major revision to the language and included expanded and improved features, some of which exceeded the capabilities of existing RDBMSs. In fact, SQL-92 was substantially longer than SQL-89 in an attempt to address many of the weaknesses of the earlier standard.

Because of the significant expansion of the standard, SQL-92 defined three levels of conformance:

- ❑ **Entry:** This level represented the most basic stage of conformance, and was based primarily on the SQL-89 standard, with only a few improvements.
- ❑ **Intermediate:** Although this level represented significant advancements to the product, it was believed that most products could achieve compliance.
- ❑ **Full:** A RDBMS had to be in complete compliance with the standard.

To be in conformance to the SQL-92 standard, a RDBMS had to comply with at least the Entry level, which has been achieved by most RDBMSs on the market. However, no known product achieved an Intermediate level of conformance, let alone Full. Part of the problem is that some of the features specified in the Intermediate level met with little interest from users. As a result, RDBMS vendors saw little reason to implement these features.

In 1999, ANSI, along with the International Organization for Standardization (ISO) published SQL:1999, the first complete update to the SQL standard since 1992. However, during those seven years, interim standards were published to incorporate features that RDBMS vendors were already being implementing. These interim publications were then incorporated in the SQL:1999 standard, which represented another significant expansion of the standard.

Because most products reached only an Entry level of conformance to SQL-92, the SQL:1999 standard took a different approach to conformance levels. To be in conformance to the new standard, a RDBMS had to be in compliance with Core SQL. Core SQL contained all the features of Entry level SQL-92, many of the features of Intermediate level, and some of the features of Full level, plus some features new to SQL:1999. In addition to claiming Core SQL conformance, a RDBMS could claim conformance to one of the supported packages. A *package* is a set of features that a vendor could implement in a RDBMS. The SQL:1999 standard supported the following packages:

- PKG001:** Enhanced date/time facilities
- PKG002:** Enhanced integrity management
- PKG003:** OLAP (online analytical processing) facilities
- PKG004:** PSM (persistent stored module)
- PKG005:** CLI (call-level interface)
- PKG006:** Basic object support
- PKG007:** Enhanced object support
- PKG008:** Active database
- PKG009:** SQL/MM (multimedia) support

Most RDBMSs, including MySQL, conform to the Entry level of SQL-92 and achieve some conformance to Core SQL in SQL:1999. However, ANSI and ISO have released yet another version of the standard — SQL:2003. In many ways, the new standard merely reorganizes and makes corrections to SQL:1999. However, the latest standard does include additional features that were not part of SQL:1999, particularly in the area of Extensible Markup Language (XML). As a result, compliance with SQL:1999 does not necessarily imply compliance to SQL:2003.

With the advent of SQL:2003, future releases of RDBMS products will inevitably change, and some vendors are already working on achieving compliance with the new standard. However, as of today, no product claims compliance to SQL:2003.

You can purchase the ANSI/ISO SQL:2003 standard online at the ANSI eStandards Store (<http://webstore.ansi.org>). The standard is divided into 14 parts, which you must purchase individually.

Object-Relational Model

As explained earlier, SQL was developed as a way to implement the relational model. To this end, the language has been quite successful, attested to by its widespread implementation and the commitment of companies such as Microsoft, IBM, Oracle, and MySQL AB to relational databases and SQL. However, most RDBMS vendors have extended the SQL-based capabilities of their products to include features that go beyond the pure relational nature of SQL. Many of these new features are similar to some of the characteristics found in *object-oriented programming*, a type of programming based on self-contained collections of routines and data structures that each perform a specific task. As SQL, as well as various RDBMS products, has become more advanced, it has taken a turn toward object-oriented programming (although, strictly speaking, SQL is far from being an object-oriented language).

Java and C# are both examples of object-oriented programming languages. In these languages, objects interact with one another in a way that addresses complex programming issues that cannot be easily addressed with traditional procedural languages.

A good example of the object-oriented nature of some of the extended features in RDBMSs is the stored procedure. A *stored procedure* is a collection of SQL statements that are grouped together to perform a specific operation. The SQL statements are saved as a single object stored in the database and that users can evoke as needed.

By the mid-1990s, most RDBMS products had implemented some form of the stored procedure. To address this trend, ANSI released in 1996 an interim publication referred to as SQL/PSM, or PSM-96. (PSM refers to *persistent stored module*.) A PSM is a type of procedure or function stored as an object in the database. A *procedure* is a set of one or more SQL statements stored as a unit, and a *function* is a type of operation that performs a specific task and then returns a value.

The SQL/PSM standard defined how to implement PSMs in SQL. Specifically, SQL/PSM included the language necessary to support stored procedures (which were referred to as *SQL-invoked procedures* in the standard). SQL/PSM was later incorporated into the SQL:1999 standard.

The problem that ANSI ran into when trying to standardize the SQL language related to stored procedures is that the way in which stored procedures were implemented from product to product varied widely. As a result, the manner in which stored procedures are called and retrieved can be very different not only between products and the SQL:1999 standard, but also among the products themselves. As a result, the implementation of stored procedures remains very proprietary, with few products conforming to the actual standard.

MySQL currently does not support stored procedures, although SQL AB is including this functionality in version 5.0. The stored procedure functionality is based on the SQL:2003 standard.

The differences among the products extend beyond only stored procedures. Other features have experienced the same fate as stored procedures because so many of these features had been implemented prior to the standardization of related SQL statements. Still, many of SQL's advanced features, with their object-oriented characteristics, are here to stay, as can be seen in both the SQL:1999 and SQL:2003 standards and in the RDBMS products, making SQL an *object-relational* database language and the RDBMS products *object-relational database management systems*.

The Nonprocedural Nature of SQL

Despite the influence of object-oriented programming on SQL, SQL is still very different from other programming languages. Traditional procedural programming languages such as COBOL, Fortran, and C were designed for very specific purposes, none of which were for accessing data. For this reason, SQL was intended for use in conjunction with these languages in order to build applications that could easily access data. For this reason, SQL is not intended for use as a standalone language, which is why it is sometimes referred to as a *sublanguage*. Insufficient for writing complete applications, SQL was designed with the idea that there would always be a *host language* for application building.

Traditional programming languages, which range from Fortran to C, are considered to be *procedural languages*; that is, they define how to carry out an application's operations and the order in which to carry them out. SQL, on the other hand, is nonprocedural in nature. It is concerned primarily with the results of an operation. The host language determines how to process the operation. Of course, this doesn't mean that SQL doesn't include procedural elements. For example, stored procedures are such an element, and certainly RDBMS vendors recognize the need for at least some procedural functionality.

Yet these procedural elements do not make SQL a procedural language. SQL doesn't have many of the basic programming capabilities of the other languages. As a result, you cannot build an application with SQL alone. You must use a procedural language that works in conjunction with SQL to manipulate data stored in a RDBMS.

SQL Statements

SQL is made up of a set of statements that define the structure of a database, store and manage data within that structure, and control access to the data. At the heart of each SQL statement is a syntactical structure that specifies how the statement can be created. The syntax acts as blueprint for building statements that the RDBMS interprets. Most RDBMS products provide little leeway for statements that don't adhere strictly to the syntactical foundations. As a result, you should know how to read and interpret statement syntax if you plan to use SQL in your applications or access data in an SQL database.

Working with Statement Syntax

When you create an SQL statement, you must often rely on the statement syntax defined in a product's documentation. The syntax provides the guidelines you need to create a statement that RDBMS can interpret. For each statement, the syntax — through the use of keywords and symbols — defines the statement's structure, the elements required within the statement, and options you can include to help refine the statement.

When you first look at the complete syntax for any statement, it might seem overwhelming, depending on the statement. For some statements, there are relatively few elements, so interpretation is easy. However, other syntax can be pages long. Despite the complexities of a particular statement, the basic syntax elements are the same, and if you learn how to interpret those elements, you can, with practice, understand any syntax presented to you.

The elements comprising a syntactic structure can vary from reference to reference and from product to product, although in many cases, the symbols used are the same. This book follows ANSI's SQL:2003 standards. You may encounter partial syntax throughout this book. In some cases, there are simply too many syntactic elements, and many of those elements are rarely implemented. Whenever you want to be certain that you're seeing a statement's syntax in its entirety, be certain to check the MySQL documentation.

Introducing the MySQL Relational Database Management System

Below is an example of a statement's syntax so that you can get a better feel for how to compose an SQL statement. The example below is based on the MySQL `CREATE TABLE` statement. The statement varies somewhat from the statement as it's defined in the SQL:2003 standard; however, the basic elements are the same. The following syntax demonstrates all the elements that comprise any SQL statement, in terms of the structure and symbols used:

```
<table definition> ::=
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
(<table element> [{, <table element>}...])
[ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MRG_MYISAM | MYISAM}]

<table element> ::=
{<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]}
| {PRIMARY KEY (<column name> [{, <column name>}...])}
| {INDEX [<index name>] (<column name> [{, <column name>}...])}
```

The syntax shown here does not represent the `CREATE TABLE` statement in its entirety, but it does include the fundamental components. Chapter 5 examines the table definition syntax in far more detail, but for now, the primary concern is that you learn how to interpret SQL statement syntax.

The syntax method employed here is referred to as BNF (Backus Naur Form) notation. Most resources that discuss syntax for SQL statements use BNF notation or something similar to this.

Before examining the syntax example in detail, review the symbols used as part of syntax notation. The following conventions define how to create a statement, based on the meaning of the symbols within the context of the syntax:

- ❑ **Vertical bar (|):** The vertical bar can be interpreted to mean “or.” Whenever you can choose from two or more options, those options are separated with a vertical bar. For example, in the sixth line, you can choose either `NOT NULL` or `NULL`.
- ❑ **Square brackets ([]):** A set of square brackets indicates that the syntax enclosed in those brackets is optional.
- ❑ **Angle brackets (< >):** A set of angle brackets indicates that the syntax enclosed is a placeholder, in which case, you must insert a specific value in place of the angle brackets and the text within those brackets. If the meaning of the placeholder is not self-evident, a later section within the syntax usually defines it.
- ❑ **Curly brackets ({ }):** A set of curly brackets indicates that the syntax enclosed in those brackets should be treated as a unit. As a result, if one element within the brackets is used, all elements are used, unless a vertical bar separates options within the brackets.
- ❑ **Three periods (...):** A set of three periods means that the clause that immediately precedes the periods can be repeated as often as necessary.
- ❑ **Two colons/equal sign (::=):** The colon/equal sign construction defines placeholders. Literally, it is the equivalent to an equal sign. The syntax to the right of the symbols defines the specified placeholder to the left.

Chapter 1

Once you understand how to use these six symbols, you should be able to interpret most syntax. However, one other syntactic element that you should be aware of is the use and placement of keywords. A *keyword* is a reserved word or set of reserved words that are part of the SQL lexicon. The keywords define a statement's action and how that action is carried out. For example, the `CREATE TABLE` keywords indicate that this statement does what you would expect it to do — create a table.

Normally, keywords are represented in all uppercase to distinguish them from placeholders, but SQL is a case-insensitive language, so the use of uppercase is meant only as a way to write more readable code. You could also write `Create Table`, `create table`, or `CREate taBLE`, and MySQL would interpret the code in the same way.

Not only is SQL indifferent to capitalization, it also isn't concerned with tabs, extra spaces, or line breaks. In theory, you could write your entire SQL statement on one line, or you could place each word on separate lines. However, it's recommended that you construct your statements in such a way that they are easy to read and understand, so breaking a statement into several lines is a common approach to take.

Returning to the example syntax and reviewing it line by line, the syntax begins by identifying the type of statement that is being defined:

```
<table definition>::=
```

Literally, the syntax means that the `<table definition>` placeholder is equivalent to the syntax that follows. SQL-related documentation often omits this introductory line, and it is seldom necessary at the beginning of the syntax. Usually, the syntax itself clearly defines the statement's purpose. However, it's included here so that you recognize it should you run into it in SQL documentation. To review the second line of the syntax:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
```

This line represents the actual first part of a `CREATE TABLE` statement. Notice that the keyword `TEMPORARY` separates the `CREATE TABLE` keywords. Because square brackets enclose this keyword, the keyword is optional. You would include it only if you plan to create a temporary table. (Temporary tables are discussed in Chapter 5.) Because of the optional keyword, a table definition can begin with `CREATE TABLE` or `CREATE TEMPORARY TABLE`.

The next part in this line of syntax is the keywords `IF NOT EXISTS`. Again, these keywords are optional and would be included only if you want to check for the existence of a table with the same name. Note, however, that when a set of brackets encloses multiple words in this manner, all the keywords are included or none are included. You would not use `IF`, `NOT`, or `EXISTS` alone within this context of this part of the statement. In other words, you would never create a statement that begins with the following:

```
CREATE TABLE EXISTS <table name>
```

Notice that the final element in this line of syntax is the `<table name>` placeholder. This is the position within the statement in which you provide a name for the table that you're creating. When the table is added to the database, it is assigned the name that you provide here, and this is the name that you use whenever referring to the table. Now look at the next line of syntax:

```
(<table element> [{, <table element>}...])
```


Introducing the MySQL Relational Database Management System

The first thing you might notice is that parentheses enclose all the elements. Whenever you include parentheses in this way (and the parentheses are not enclosed by square brackets), the parentheses are required. As a result, the required elements of this line are (<table element>). Because the <table element> placeholder represents a more complex syntax than merely the name of an object, the placeholder is defined later in the code.

The important point to remember is that at least one <table element> is required, but you can include as many as necessary. However, this is when the syntax gets a little trickier. Notice that several elements are enclosed in square brackets — [{, <table element>}...] — telling you that this part of the syntax is optional. However, curly brackets group together parts of the syntax within the square brackets — {, <table element>} — and they are followed by three periods. The curly brackets mean that the elements within those brackets must be kept together, and the periods mean that the group of elements can be repeated as often as necessary. As a result, whenever you include an additional <table element> in your statement, you must precede it with a comma, but you can do this as many times as necessary. For example, if the statement includes four table elements, your syntax would be as follows:

```
(<table element>, <table element>, <table element>, <table element>)
```

As you can see, when you include more than one <table element>, you must follow each one with a comma, except for the last one. And keep in mind that parentheses must enclose them all. Moving on to the next line of syntax:

```
[ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MRG_MYISAM | MYISAM}]
```

One of the first things that you notice is that square brackets enclose the entire line, which means that the entire line is optional. The line defines the type of table that you plan to create. If you do include this line in your CREATE TABLE statement, then you must include ENGINE = plus one of the table type options. You can tell that you're allowed to select only one option because a vertical bar separates each option. You could read this as BDB or MEMORY or ISAM or INNODB, and so on. For example, if you want to define the table as an INNODB table, you would include the following line in your syntax:

```
ENGINE = INNODB
```

You should now have a basic understanding of how to create a CREATE TABLE statement. However, as you may recall, the <table element> placeholder could not be easily defined by its placement or usage. As a result, the syntax goes on to define the components that can make up a <table element>. You can tell that the syntax defines the <table element> placeholder because it precedes the definition with the actual placeholder, as shown in the first line in the next section of syntax:

```
<table element>::=
```

From this, you know that whatever follows is part of the syntax that defines the <table element> placeholder. Before you look too closely at the first line in the <table element> definition, take a look at all three lines that make up that definition:

```
{<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]}  
| {PRIMARY KEY (<column name> [{, <column name>}...])}  
| {INDEX [<index name>] (<column name> [{, <column name>}...])}
```

Chapter 1

What you might have noticed is that a vertical bar precedes the last two lines and that curly brackets enclose all three lines. This means that each line represents one of the options you can use to define a `<table element>` placeholder. In other words, for each `<table element>` that you include in your `CREATE TABLE` statement, you can define a column, a primary key, or an index.

A primary key is a constraint placed on one or more columns within a table to indicate that the columns act as the primary identifier for each row in that table. Values within a primary key's columns must be unique when taken as a whole. You learn about primary keys in Chapter 5, which discusses how to create a table.

Take a look at the first line of the `<table element>` definition:

```
{<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]}
```

This line defines a column within the table. Each column definition must include a name (`<column name>`) and a data type (`<type>`). A *data type* determines the type of data that can be stored in a table. The line also includes three optional elements. The first of these is `[NOT NULL | NULL]`, which means that you can set a column as `NOT NULL` or `NULL`. A *null* value indicates that a value is undefined or unknown. It is not the same as zero or blank. Instead it means that a value is absent. When you include `NOT NULL` in your column definition, you're saying that the column does not permit null values. On the other hand, the `NULL` option permits null values.

The next optional element in the column definition is `[DEFAULT <value>]`. This option allows you to define a value that is automatically inserted into a column if a value isn't inserted into the column when you create a row. When you include the `DEFAULT` keyword in your column definition, you must include a value in place of the `<value>` placeholder.

The final optional element of the column definition is `[AUTO INCREMENT]`. You include this option in your definition if you want MySQL to automatically generate sequential numbers for this column whenever a new row is added to the table.

With regard to the three options available in the `<table element>` definition, the column definition is the one you use most often. However, as stated above, you can choose any of three options, so take a look at the second line:

```
| {PRIMARY KEY (<column name> [{,<column name>}...])}
```

The purpose of this line is to define a primary key for the table. If you choose this option, you must include the `PRIMARY KEY` keywords and at least one column name, enclosed in parentheses. The elements contained in the square brackets — `[{, <column name>}...]` — indicate that you can include one or more additional columns and that a comma must precede each additional column. For example, if you base your primary key on three columns, your syntax is as follows:

```
PRIMARY KEY (<column name>, <column name>, <column name>)
```

Don't worry if you don't understand how primary keys are created or how they can be made up of multiple columns. Primary keys are discussed in detail in Chapter 5.

Introducing the MySQL Relational Database Management System

Now examine the last optional element in the `<table element>` definition:

```
| {INDEX [<index name>] (<column name> [{, <column name>}...])}
```

This line creates an index. If you use this option, you must include the `INDEX` keyword and at least one column name, enclosed in parentheses. As was the case when creating a primary key, you can also include additional columns, as long as a comma precedes each additional column. However, unlike a primary key, the index name is optional. It's up to you whether you want to name the index, although naming all objects in a database is generally considered a good practice.

You should now have a fairly good sense of how to interpret a statement's syntax. As you have seen from the table definition example, the syntax for an SQL statement can contain many elements. However, once you're comfortable with syntax structure and how symbols define this structure, you should be able to interpret the syntax for nearly any SQL statement (albeit some statements might present a far greater challenge than other statements). The next section discusses how to use this syntax to create an SQL statement.

Creating an SQL Statement

An SQL statement can range from very simple — only a few words — to very complicated. If at any point in the statement-creation process you're uncertain how to proceed, you can refer to the syntax for direction. Even experienced SQL programmers must often refer back to the syntax in order to understand the subtleties of a particular statement, but once you have that syntax as a frame of reference, you're ready to build your statement.

Below is an example of a statement based on the table definition syntax. The following `CREATE TABLE` statement creates a table named `Parts`:

```
/* Creates the Parts table */
CREATE TABLE Parts
(
  PartID INT NOT NULL,
  PartName VARCHAR(40) NOT NULL,
  CatID INT NOT NULL,
  PRIMARY KEY (PartID)
)
ENGINE=MYISAM;
```

The first thing to note is that the `CREATE TABLE` example is a single SQL statement. Notice that it ends with a semi-colon, which is sometimes referred to as a *terminator*. When you access a MySQL database directly (for example, by using the `mysql` client utility), you must terminate each SQL statement with a semi-colon.

As mentioned earlier, SQL is indifferent to extra spaces, tabs, and line breaks. However, the statement is written in such a way as to facilitate readability and to make it easier to explain each element. For example, the table elements are indented and the opening and closing parentheses are placed on their own lines.

Chapter 1

The first line of code reads:

```
/* Creates the Parts table */
```

The code is merely a comment and is not processed by MySQL. (In fact, you normally wouldn't use comments when working with MySQL interactively, but the comment is included here to demonstrate how they work.) It is there only to provide information to anyone who might be viewing the code. Adding comments to your code to explain the purpose of each part or to provide any special information is always a good idea. Comments are particularly useful when updating code you had created in the past or when someone else is working on code that you created. Comments are also very useful if you're trying to debug your code.

You can also create a comment by preceding the text with double dashes (--). However, the comment cannot include any line breaks.

As you can see from the line of code, a comment begins with `/*` and ends with `*/`. Everything between the two symbols, including line breaks, is considered part of the comment and therefore ignored by MySQL when processing the SQL statements.

The next line of code is the first line of the actual `CREATE TABLE` statement:

```
CREATE TABLE Parts
```

As you can see, the line includes the `CREATE TABLE` keywords and the name of the new table — `Parts`. If you refer to the related line of syntax, you can see how to form the `CREATE TABLE` clause :

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
```

Notice that the optional keyword `TEMPORARY` and the optional keywords `IF NOT EXISTS` are not included in the clause, only the required elements. Now take a look at the `<table element>` definitions:

```
(  
    PartID INT NOT NULL,  
    PartName VARCHAR(40) NOT NULL,  
    CatID INT NOT NULL,  
    PRIMARY KEY (PartID)  
)
```

This part of the `CREATE TABLE` statement includes four `<table element>` components. The first three represent column definitions and the last represents a primary key definition. The elements are enclosed in parentheses and separated by commas. If you compare this to the syntax, you can see how the `<table element>` placeholders represent each component:

```
(<table element>, <table element>, <table element>, <table element>)
```

Introducing the MySQL Relational Database Management System

However, as you may recall, this is only a part of the story because the syntax further defines the `<table element>` placeholder to include three options: a column definition, a primary key definition, and an index definition. Take a look at one of the column definitions in the sample CREATE TABLE statement to illustrate how it compares to the syntax:

```
PartID INT NOT NULL,
```

In this case, the column's name is `PartID`, it is configured with `INT` data type (to permit up to four numerals), and null values are not permitted. If you compare this to the syntax, you can see why the column definition is structured as it is:

```
<column name> <type> [NOT NULL | NULL] [DEFAULT <value>] [AUTO_INCREMENT]
```

As you can see, `PartID` is the value used for the `<column name>` placeholder, `INT` is the value used for the `<type>` placeholder, and the only optional element used is `NOT NULL`. The `<type>` placeholder refers to the column's data type, which in this case is `INT`. Notice also that this `<table element>` definition ends with a comma because another `<table element>` definition follows. Now look at the primary key definition:

```
PRIMARY KEY (PartID)
```

This line only includes the `PRIMARY KEY` keywords and one column name. As a result, a comma is not required after the column name. However, parentheses must still enclose the column name. Also, because this is the last `<table element>` definition, you don't need to use a comma after the primary key definition. Compare this to the syntax:

```
PRIMARY KEY (<column name> [{, <column name>}...])
```

As you can see, you use none of the optional elements, only what is essential for a primary key definition. Moving on to the final line of the `CREATE TABLE` statement:

```
ENGINE=MYISAM;
```

This part of the statement defines the type of table that is being created. If you compare this to the syntax, you see that you've chosen one of the available options:

```
[ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MRG_MYISAM | MYISAM}]
```

As the syntax indicates, this entire line of code is optional. However, if you're going to include it, you must include `ENGINE=` and exactly one of the options.

As you can see from this example, creating an SQL statement is a matter of conforming to the structure as it is defined in the statement syntax. Running the `CREATE TABLE` statement in the example above adds a table to your database. From there, you could add data to the table, access that data, and modify it as necessary. Figure 1-5 shows what the table might look like in a MySQL database once you've executed the `CREATE TABLE` statement and then added data to the table.

Parts table

PartID	PartName	CatID
1001	Guy wire assembly	503
1002	Magnet	504
1003	Regulator	505
1004	Brushes	504
1005	Generator	506
1006	Dump load system	506
1007	Power assembly	501
1008	Tower assembly	501
1009	Rotor assembly	501
1010	Hub	611
1011	Shaft assembly	612
1012	Governor	619
1013	Furling tail	612

Figure 1-5

As displayed in the figure, the Parts table includes the three columns — PartID, PartName, and CatID — as well as the sample data. You could have created a table with as many columns as necessary, depending on the design of your database and your requirements for storing data, and you could have added many more rows of data.

MySQL provides a client utility named `mysql`, which allows you to interact directly with MySQL databases. The `mysql` client utility is similar to a command prompt as you would see in an operating system command window. If you were use the client utility to view the contents of the table, you would see the data displayed in a manner similar the following:

```
+-----+-----+-----+
| PartID | PartName          | CatID |
+-----+-----+-----+
| 1001   | Guy wire assembly | 503   |
| 1002   | Magnet            | 504   |
| 1003   | Regulator         | 505   |
| 1004   | Brushes           | 504   |
| 1005   | Generator         | 506   |
| 1006   | Dump load system  | 506   |
| 1007   | Power assembly   | 501   |
| 1008   | Tower assembly   | 501   |
| 1009   | Rotor assembly   | 501   |
| 1010   | Hub               | 611   |
| 1011   | Shaft assembly   | 612   |
| 1012   | Governor         | 619   |
| 1013   | Furling tail     | 612   |
+-----+-----+-----+
13 rows in set (0.03 sec)
```

Introducing the MySQL Relational Database Management System

These results aren't quite as elegant as the table in Figure 1-5, but they display the same information. In addition, this option represents the way you're likely to see data displayed if you're working with MySQL directly.

MySQL, the RDBMS, is very different from mysql the command-line utility. The mysql utility is a client tool that provides an interactive environment in which you can work directly with MySQL databases. In most documentation, including the MySQL product documentation, the utility is shown in all lowercase. You learn more about using the mysql client tool in Chapter 3.

Now that you have an overview of how to use statement syntax to create SQL statements, you can learn about the different types of statements that SQL supports.

Types of SQL Statements

As you work your way through this book, you find that SQL supports many different types of statements and that most of these statements include a number of options. SQL is generally broken down into three categories of statements: data definition language (DDL), data manipulation language (DML), and data control language (DCL). The following three sections discuss each of these statement types and provide you with examples. These examples use the table in Figure 1-5. However, keep in mind that these examples are meant only to introduce you to SQL statements. Each statement is covered in much greater detail later in the book, but the examples help to provide you with an overview of how to create SQL statements.

Using DDL Statements

In MySQL, DDL statements create, alter, and delete data structures within the database. DDL statements define the structure of a MySQL database and determine the type of data that can be stored in the database and how to store that data. Specifically, DDL statements allow you to do the following:

- ❑ Create and remove databases (the `CREATE DATABASE` and `DROP DATABASE` statements)
- ❑ Create, alter, rename, and remove tables (the `CREATE TABLE`, `ALTER TABLE`, `RENAME TABLE`, and `DROP TABLE` statements)
- ❑ Create and remove indexes (the `CREATE INDEX` and `DROP INDEX` statements)

As you move through the book, you learn more about databases, tables, and indexes and are provided with details on how to create statements that allow you to work with those objects. In the meantime, here's a brief description of each of these objects so you have a better idea of the nature of DDL statements:

- ❑ **Database:** As you learned at the beginning of the chapter, a database is a collection of related data organized in a structural format that is described by metadata.
- ❑ **Table:** A table is a grid-like structure that consists of a set of columns and rows that represent a single entity within the database. Each row is a unique record that stores a set of specific data.
- ❑ **Index:** An index is a list of values taken from a specified column. Indexes are used to speed up searches and reduce the time it takes to execute an SQL query.

Chapter 1

Earlier in the chapter, you saw an example of a DDL statement — the `CREATE TABLE` statement — which you used to create a table named `Parts`. Another example of a DDL statement, the following `DROP TABLE` statement removes the `Parts` table from the database:

```
/* Removes the Parts table from the database */  
DROP TABLE Parts;
```

As this example demonstrates, some DDL statements (as well as some DML and DCL statements) can be quite simple. In this case, you need only to specify the table's name along with the keywords `DROP TABLE` to remove its contents from the database. (As you may recall, the first line in this code is merely a comment that provides information about the statement to come.)

In Chapter 5, you learn how to create, alter, rename, and drop tables, as well as how to create and drop databases and indexes. The next section outlines DML statements.

Using DML Statements

Unlike DDL statements, DML statements are more concerned with the data stored in the database than the database structure itself. For example, you can use DDL statements to request information from the database as well as insert, update, and delete data; however, you would not use a DML statement to create or modify the tables that hold the data. Specifically, DML statements allow you to do the following:

- ❑ Query a database for specific types of information from one or more tables (the `SELECT` statement)
- ❑ Insert data into a table (the `INSERT`, `REPLACE`, and `LOAD DATA INFILE` statements)
- ❑ Update existing data in a table (the `INSERT` and `REPLACE` statements)
- ❑ Delete data from a table (the `DELETE FROM` and `TRUNCATE TABLE` statements)

The `SELECT` statement is perhaps the statement that you use more than any other. The `SELECT` statement allows you to retrieve data from one or more tables in a database. Whenever you query a database, you use the `SELECT` statement to initiate that query. The statement can be relatively simple or very complex, depending on the type of information you're trying to retrieve and the degree to which you want to refine your search.

The following `SELECT` statement provides you with an example of how you can retrieve data from the `Parts` table:

```
/* Retrieves data for parts with CatID less than 600 */  
SELECT PartName, PartID, CatID  
FROM Parts  
WHERE CatID < 600  
ORDER BY PartName;
```

Again, a comment that describes the code's purpose introduces the statement. The actual statement begins on the second line and is divided into four clauses, each written on its own line. A clause is simply a section of a statement. The clause is usually referred to by the keyword that starts the clause. For example, the first clause is the `SELECT` statement is the `SELECT` clause. The first clause in any

Introducing the MySQL Relational Database Management System

`SELECT` statement is always the `SELECT` clause, which indicates which columns to include in the query. In this case, the `SELECT` clause retrieves data from the `PartName`, `PartID`, and `CatID` columns. (You can refer back to Figure 1-5 to view the table and its contents.) The `FROM` clause is next, and it provides the name of the table or tables to include in the query. The `WHERE` clause refines the query based on the conditions specified in the clause. In this case, only rows with a `CatID` value less than 600 are included in the query results. This is indicated by the name of the column (`CatID`), the less than operator (`<`), and the number 600. The final clause, `ORDER BY`, determines the order in which the query results list data. This `SELECT` statement lists the query results in alphabetical order based on the values in the `PartName` column.

The intent of providing you an example `SELECT` statement is merely to provide you with a high overview of one type of DML statement. You don't need to worry about all the components of a `SELECT` statement at or any other DML statement at this time. The `SELECT` statement is a complex statement that can contain a considerable number of options and can be structured in many different ways. Chapter 7 provides you with a detailed explanation of the statement's syntax and the various ways that you can construct a statement.

Using the `mysql` command-line utility to execute the `SELECT` statement above provides the following results:

```
+-----+-----+-----+
| PartName          | PartID | CatID |
+-----+-----+-----+
| Brushes           | 1004   | 504   |
| Dump load system  | 1006   | 506   |
| Generator         | 1005   | 506   |
| Guy wire assembly | 1001   | 503   |
| Magnet            | 1002   | 504   |
| Power assembly    | 1007   | 501   |
| Regulator         | 1003   | 505   |
| Rotor assembly    | 1009   | 501   |
| Tower assembly    | 1008   | 501   |
+-----+-----+-----+
9 rows in set (0.03 sec)
```

Notice that the columns appear in the order specified in the `SELECT` clause. Also notice that the query results include only those rows whose `CatID` value is less than 600 and that the rows are ordered according to the values in the `PartName` column.

The `SELECT` statement shown in the example is only one type of DML statement. MySQL also allows you to use DML statements to insert, update, and delete data, which you discover as you progress through the book. The following section examines DCL statements.

Using DCL Statements

As you learned above, DDL statements allow you to define the structure of a MySQL database, and DML statements allow you to access and manipulate data within the database. DCL statements represent yet another function supported by SQL statements: controlling access to a database. Specifically, DCL statements allow you to do the following:

- ❑ Grant access privileges to users (the `GRANT` statement)
- ❑ Revoke access privileges to users (the `REVOKE` statement)

Chapter 1

When you grant access to a database, you can grant access to specific tables and you can assign specific privileges. As a result, you can specify the exact level of access that each user or groups of users should have to specific data. For example, the following `GRANT` statement grants privileges to a user account named ethan.

```
/* grant privileges to ethan */
GRANT SELECT, INSERT
ON test.parts
TO ethan@localhost
IDENTIFIED BY 'pw1';
```

The statement is divided into four clauses. The `GRANT` clause grants the user `SELECT` and `INSERT` privileges. As a result, the user can execute `SELECT` and `INSERT` statements against the specified database. However, the user does not have any other privileges. After the `GRANT` clause, the `ON` clause grants the user privileges to the `Parts` table in the `test` database.

The test database is a sample database that is installed by default when you install MySQL. You can use the database to test the installation and practice using the product. Chapter 2 explains how to install MySQL.

The next clause is the `TO` clause, which identifies the user account to whom privileges are being granted. In this case, the user account ethan is granted privileges from the local computer. This means that the ethan user account can access the `Parts` table from the local computer. Access is granted to this account from any other computer. In addition, if the ethan user account doesn't already exist, the account is created.

The final clause in the `GRANT` statement is the `IDENTIFIED BY` clause. This clause assigns a password to the user account, which, in this case, is `pw1`.

Indeed, DCL statements provide a very different function from DDL and DML statements. In Chapter 14 you learn more about database security and about using DCL languages to manage access to your MySQL databases. However, the next thing this chapter covers is the different ways in which you can execute an SQL statement.

Types of Execution

The SQL:2003 standard defines four methods for executing an SQL statement: direct invocation, module binding, embedded SQL, and call-level interface (CLI). However, not all RDBMS products support all four types of execution. Nearly all RDBMSs support some form of direct invocation, many support embedded SQL statements, few support module binding, and nearly all support the use of CLIs. The primary methods used to execute SQL statements in a MySQL database are direct invocation and CLIs, although limited support exists for embedded SQL. This section provides an overview of all four methods to ensure that you have the full picture of data access and to make sure that you understand how methods for executing SQL statements can differ, regardless of the RDBMS product.

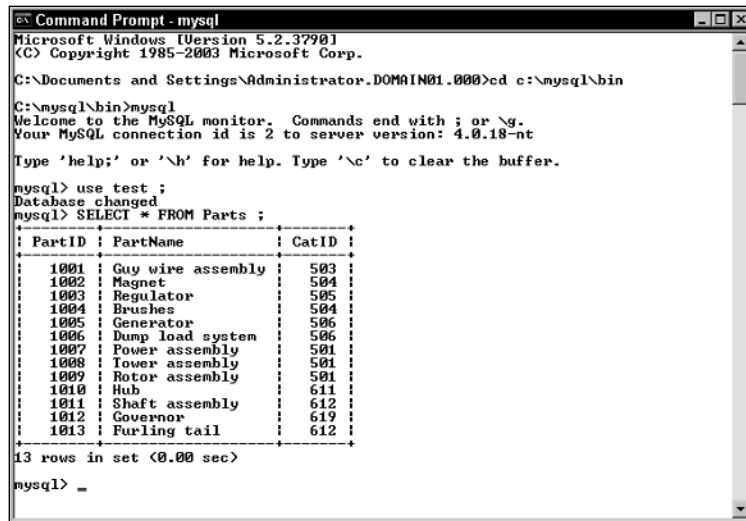
Direct Invocation

Most RDBMS products provide some sort of client application to work interactively with their databases. As a result, you can create ad hoc SQL statements that you can execute at will and receive your results immediately — or as immediately as your hardware, software, and network environments permit. This

Introducing the MySQL Relational Database Management System

process — known as *direct invocation* — is often the most expeditious way to create and modify the database structure; control access to data; and view, insert, update, or delete data.

The nature in which SQL statements are executed and the results displayed depends on the client tool supported for a RDBMS. In MySQL, the primary tool available for directly invoking SQL statements is the `mysql` command-line utility, which you run from the command prompt of the operating system where MySQL is installed. For example, if you run MySQL on Windows Server 2003, you can open a Command Prompt window, change to the `\mysql\bin` directory, and run the `mysql` utility, as shown in Figure 1-6. As you can see, `mysql` has been used to execute a `SELECT` statement against the `Parts` table in the `Test` database. The query results display directly beneath where you enter your statement.



```
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\Documents and Settings\Administrator.DOMAIN01.000>cd c:\mysql\bin

C:\mysql\bin>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 4.0.18-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> use test ;
Database changed
mysql> SELECT * FROM Parts ;
+----+-----+-----+
| PartID | PartName | CatID |
+----+-----+-----+
| 1001 | Guy wire assembly | 503 |
| 1002 | Magnet | 504 |
| 1003 | Regulator | 505 |
| 1004 | Brushes | 504 |
| 1005 | Generator | 506 |
| 1006 | Dump load system | 506 |
| 1007 | Power assembly | 501 |
| 1008 | Tower assembly | 501 |
| 1009 | Rotor assembly | 501 |
| 1010 | Hub | 611 |
| 1011 | Shaft assembly | 612 |
| 1012 | Governor | 619 |
| 1013 | Furling tail | 612 |
+----+-----+-----+
13 rows in set (0.00 sec)

mysql> _
```

Figure 1-6

You can also run the `mysql` utility from a command prompt on a remote computer. The client program must be installed on the remote computer and you must provide the necessary parameters when you launch the utility. The `mysql` tool is discussed in detail in Chapter 3.

The `mysql` tool is not limited to `SELECT` statements. You can execute DDL statements such as the `CREATE DATABASE` statement, DMS statements such as the `UPDATE` statement, and DCL statements such as the `REVOKE` statement. However, unlike a `SELECT` statement, which normally returns rows of data, the other statements return only a message that reports the success of the statement executed.

The statement below demonstrates how the `mysql` utility works. Suppose you want to create a table named `Categories`. (You can refer back to Figure 1-3 to view this table.) You can use the `mysql` utility to execute the following statement:

```
/* Creates the Categories table */
CREATE TABLE Categories
(
    CatID INT NOT NULL,
    CatName VARCHAR(40) NOT NULL,
    Parent INT NOT NULL,
    PRIMARY KEY (CatID)
)
ENGINE=MYISAM;
```

As you can see, the statement creates a table that contains three columns: CatID, CatName, and Parent. The statement also defines a primary key on the CatID column and defines the table as type MYISAM. When you execute the statement, the table is added to the database, and mysql returns the following message:

```
Query OK, 0 rows affected (0.06 sec)
```

From this message, you can see that the statement processed with no problem and that the statement affected no rows, which is expected if you're creating a table. If you were to run a statement such as the INSERT statement, the message would reflect the number of rows inserted into the table or the number of rows affected in some other way, depending on the type of statement. In addition to information about whether the query ran without error and the number of rows affected, the message tells you how long it took to process your query, which in this case is .06 seconds.

The greatest advantage, then, of using the mysql utility is that it allows you to execute and receive immediate responses to your SQL statements. However, directly invoking SQL statements through a client tool such as mysql has another advantage—it avoids a condition known as *impedance mismatch*, which occurs when a data type used in a programming language is not compatible with a data type used in a MySQL database.

Whenever you pass data between an application programming language and a MySQL database, you must ensure that the data type used to define the data in the database is compatible with the data type used to define that same data in the application language. If they are not compatible, an impedance mismatch can occur, resulting in an error in your application or in the loss of data.

As you might recall from earlier in the chapter, a data type determines the type of data that can be stored in a table's column. Data types are discussed in greater detail in Chapter 5.

Accessing a MySQL database interactively avoids the issue of impedance mismatch and provides immediate results to your SQL statements. However, only specific types of users, such as database administrators or programmers, generally use this method. The majority of users access data in a MySQL database through the applications in which they're working. Still, for the purposes of this book, you use the mysql utility to learn how to create database objects and manipulate data. Once you have a better understanding of how MySQL implements SQL, you can create applications that pass SQL statements to MySQL databases.

Module Binding

Few RDBMS vendors have implemented module binding in their products, and MySQL is *not* one of them. As a result, this chapter covers the topic only briefly. The only reason it's covered at all is because it is part of the SQL:2003 standard, and you should have as complete a picture of SQL as possible.

Module binding is a type of statement execution in which modules made up of SQL statements are called from within a host programming language. A module includes properties that define the module itself as well as one or more SQL statements that are invoked when the programming language calls the module. The module is stored as an object separate from the host programming language, so the host language contains calls to the module that invoke the SQL statements within the module.

Embedded SQL

At one time, embedded SQL was one of the most common methods used to access data in a database from within a programming language. As the name suggests, *embedded SQL* refers to SQL statements that are embedded directly within the programming language to allow that language to access and modify data within an SQL database. The SQL:2003 standard defines how SQL statements are to be embedded into a language and recommends which languages a RDBMS should support. According to the standard, the supported languages should include C, COBOL, Fortran, and several others.

Despite the SQL:2003 standard's recommendations, few RDBMSs support all the suggested programming languages. Some products support a few of the languages, while others support languages other than the ones listed in the SQL:2003 standard. Regardless of how extensively a RDBMS supports embedded SQL, most systems support at least a couple of languages, and MySQL AB is no exception. MaxDB, an enterprise-level RDBMS, includes a precompiler that allows you to embed SQL in your C and C++ applications. The precompiler removes the SQL statements from the code and places them in a file separate from the original application file. The precompiler then replaces the statements in the original file with calls to the SQL statements.

In order to embed an SQL statement into a C or C++ application, you must precede each statement with the keywords `EXEC SQL` and end the statement with a semi-colon. For example, the following embedded SQL statement inserts data into a table named `Parts`:

```
/* Embed an INSERT INTO statement in the application */
EXEC SQL INSERT INTO Parts
(PartID, PartName, CatID)
VALUES (1014, 'Heater', 505);
```

Notice that you can include comments with your SQL statement. A comment must begin with `/` and end with `*/`.*

There are, of course, more elements to embedded SQL than those shown here, not only with regard to how you embed the actual statements, but also in terms of the various options that you can use when running the precompiler. However, a more thorough discussion of embedded SQL is beyond the scope of this book. This is due primarily to the fact that embedded SQL is used on a limited basis when developing applications that connect to a MySQL database. The method used most commonly for establishing that connection and executing SQL statements is through one of the APIs that MySQL supports.

Call-Level Interface

When the SQL:2003 standard defines the methods available for accessing data in a database, it includes the call-level interface. A *call-level interface* is essentially an API that allows a programming language to communicate directly with an SQL database. A call-level interface, or API, includes a set of routines that the programming language can call in order to facilitate data access from within that language. The routines access the data as defined within the programming language and return that data to the program, where the host programming language can process it.

MySQL supports a number of APIs that allow applications written in various types of programming languages to communicate with MySQL databases. The following descriptions provide an overview of several of these APIs.

- ❑ **C:** The C API, which is distributed with MySQL, is the main programming interface that allows applications to connect to MySQL. Most of the client applications included in the MySQL distribution are written in C and rely on this API. In addition, the other APIs, except those used for Java applications, are based on the C client library, which defines the C API.
- ❑ **ODBC:** MySQL supports Open Database Connectivity (ODBC) through the MySQL AB product MySQL Connector/ODBC. ODBC is a database connectivity standard that allows different types of applications to connect to different types of databases. ODBC-compliant applications can use MySQL Connector/ODBC to connect to MySQL databases.
- ❑ **JDBC:** MySQL supports Java Database Connectivity (JDBC) through the MySQL AB product MySQL Connector/JDBC. JDBC is a database connectivity standard that allows Java applications to connect to different types of databases. JDBC-compliant applications can use MySQL Connector/ODBC to connect to MySQL databases.
- ❑ **PHP:** The PHP API, which is now included with the PHP preprocessor, allows a PHP script on a Web page to communicate directly with a MySQL database. Database connections and requests for data (through SQL statements) are coded directly in the PHP script.

MySQL supports many more APIs than are listed here. However, these are the four that this book is most concerned with because, by the end of the book, you learn how to connect to a MySQL database from within a PHP application, a JDBC-compliant application, and an ODBC-compliant application. (The ODBC and PHP APIs interface with the C client library, which is why the C API is also included here.) For more information about each of these APIs and the other APIs supported by MySQL, see Appendix B.

As already mentioned, the MySQL APIs are the most common method used for accessing data in MySQL databases. An API allows your application to establish a connection to MySQL and its databases, send SQL statements to the databases, and process the results that the statements return. For this reason, this book includes considerable coverage of database connectivity within PHP, JDBC, and ODBC applications. (See chapters 18, 19, and 20, respectively.) In addition, you are introduced to data-driven applications in more detail later in this chapter, where you see an example of how an application uses an API to access data within a MySQL database.

Implementation-Specific SQL

By now, you should have a good overview of SQL and how to create SQL statements. As you have seen, there are several types of SQL statements (DDL, DML, and DCL). For example, the `CREATE TABLE` statement is a type of DDL statement. However, not all `CREATE TABLE` statements are created equally, or any other type of statement for that matter. What this means is that each RDBMS product implements SQL in its own way. Although most of them try to reach at least some conformance to the ANSI standard, many of a statement's details are unique to the product. In other words, a `CREATE TABLE` statement in MySQL is not quite the same as a `CREATE TABLE` statement in Oracle. And the `CREATE TABLE` statements in both products are implemented a little differently from how the SQL:2003 standard defines the statement.

Still, generally a number of similarities exist between statements as they appear in the products and in the SQL standard. The following example takes a close look at the `CREATE TABLE` statement to help illustrate the differences. Earlier in the chapter you learned the basic table definition syntax. The first three lines of the syntax for the actual statement are as follows:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <table name>
(<table element> [{, <table element>}...])
[ENGINE = {BDB | MEMORY | ISAM | INNODB | MERGE | MRG_MYISAM | MYISAM}]
```

The opening line (<table definition>::=) of the table definition is omitted because it's not important to the discussion here.

The statement syntax provided previously was for the `CREATE TABLE` statement as it is used in a MySQL database. When creating a table in MySQL, you can create a basic table or a temporary table, and you can create the table whose creation is dependant on whether a table by that name already exists. In addition, you can select the type of table that you want to create, for example, an `INNODB` table.

The first few lines of the table definition syntax as it appears in the SQL:2003 standard appear below:

```
CREATE [{GLOBAL | LOCAL} TEMPORARY] TABLE <table name>
(<table element> [{, <table element>}...])
[ON COMMIT {PRESERVE|DELETE} ROWS]
```

The basic syntax is the same; however, there are a number of differences in the optional syntax elements. For example, according to the SQL standard, if you create a temporary table, you define it as `GLOBAL` or `LOCAL`. In addition, the SQL standard includes no facility for checking whether a table already exists. (There is no set of `IF NOT EXISTS` keywords.) The options in the third row are also very different between the SQL standard and the statement its use in MySQL. MySQL gives you the ability to define the type of table that you want to create; the SQL standard does not.

Another difference between MySQL and the SQL standard is the `ON COMMIT` option that the standard supports. In actuality, MySQL supports a similar option, but it isn't covered in this chapter. However, MySQL implements this functionality in a manner different from the standard. For more information about all the options of the `CREATE TABLE` statement, as MySQL implements is, see Chapter 5.

Now that you've seen how the `CREATE TABLE` statement can differ between MySQL and the SQL:2003 standard, take a look at yet another version of the statement. The following `CREATE TABLE` syntax shows the first few lines of the `CREATE TABLE` syntax as SQL Server 2000 defines it:

```
CREATE TABLE <table name>
(<table element> [{, <table element>}...])
[ON {<filegroup> | DEFAULT}]
```

The first two lines of the statement contain elements that are basically the same as in MySQL and the SQL:2003 standard. However, the third line, which is optional, contains elements that you won't see in MySQL or in the SQL:2003 standard.

In addition, if you were to compare the `<table element>` definitions for these three syntax examples, you would find a number of differences at this level as well. The point here is that each product implements SQL differently from one another. This can be important if you're writing applications that could be used to access databases in different RDBMS products. In other words, if you're writing an application that uses SQL to access data, you must be sure that you're familiar with how that product implements SQL, which is why SQL statements are covered, as they're implemented in MySQL, in considerable detail.

Data-Driven Applications

The goal for many of you in reading this book is to learn about MySQL so that you can create data-driven applications that can access MySQL databases. For some of you, this may mean installing MySQL, designing and creating databases, populating the databases with data, and administering the MySQL environment. However, many of you may be concerned primarily with how to build applications in a specific programming language that can connect to those databases.

In order to build an effective data-driven application, it's helpful to have a high-level understanding of how these types of applications operate. You've already been introduced to MySQL, relational databases, and SQL, and have been provided an overview of how these components fit together. As you may recall, the MySQL RDBMS provides the structure necessary to support and interact with SQL databases. However, MySQL is only part of the equation in any data-driven application. The application itself must reside within some sort of framework in order to operate and communicate with the database.

A good example of how this structure works is the Web-based application. The application must be called from within the context of a Web server (for example, Apache or Internet Information Services) and must reside in an environment that supports the application's functionality. The next section gives you an example to better illustrate these concepts.

The Application Server

Web-based applications are one of the most widely implemented types of data-driven applications in use today. Nearly every time you access a Web site, you're using an application that in some way interacts with a database. Whether you're shopping online, searching for information, or managing your bank account, you're working with a data-driven application.

Introducing the MySQL Relational Database Management System

Web-based applications based on PHP and MySQL are becoming increasingly popular. PHP is a server-side scripting language that is embedded directly in a Web page. The PHP preprocessor, which includes the MySQL API, processes the PHP script on a Web page on the server and provides Hypertext Markup Language (HTML) output that is then sent to the client computer along with the other HTML elements on the page.

This section outlines an example of a server that is set up to support PHP applications. Figure 1-7 shows how to use PHP, Apache, and Linux together to support a PHP application. At the first layer on any system you're setting up is the operating system, which in this case is Linux; however, this could be any operating system that would support the necessary PHP environment. At the next layer is Apache, a Web server that runs in the Linux environment. Apache supports a variety of Web-based applications, including PHP. The PHP preprocessor runs in conjunction with Apache and includes the PHP MySQL API.

Together, Linux, Apache, and the PHP preprocessor provide the necessary environment to support a PHP application. The application (.php) files are located within the Linux file structure, are hosted within the Apache environment, and are processed through the PHP preprocessor. However, you need one more ingredient to allow the PHP application to communicate with a MySQL database—the PHP MySQL API. The API facilitates the communication with MySQL and its databases, allowing the application to access data within the databases.

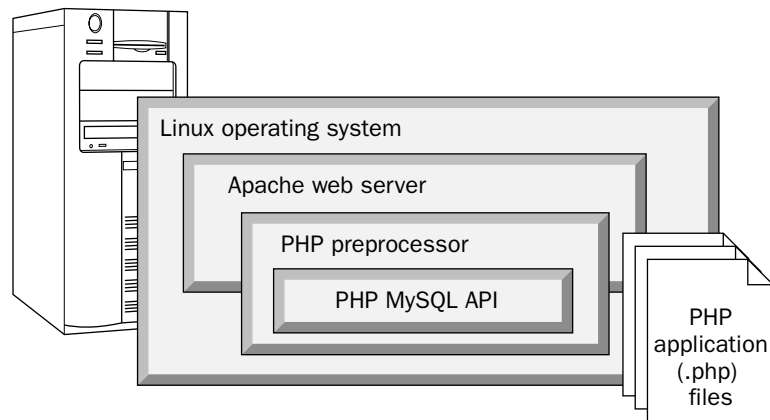


Figure 1-7

Connecting to a MySQL Database

In order to communicate with MySQL, a PHP application must include within its script the elements necessary to establish and maintain a connection with MySQL. You achieve the connection by leveraging the MySQL API, which is based on the C client library in MySQL. When a client computer requests data from a PHP application, the PHP preprocessor executes the PHP code and replaces it with whatever output is produced by executing that code. If code execution includes retrieval from a MySQL database, the data is retrieved and the output is incorporated into the output that the preprocessor produces. The output is rendered as HTML and returned to the client computer requesting the information. If you refer to Figure 1-8, you can get a better idea of how this process works.

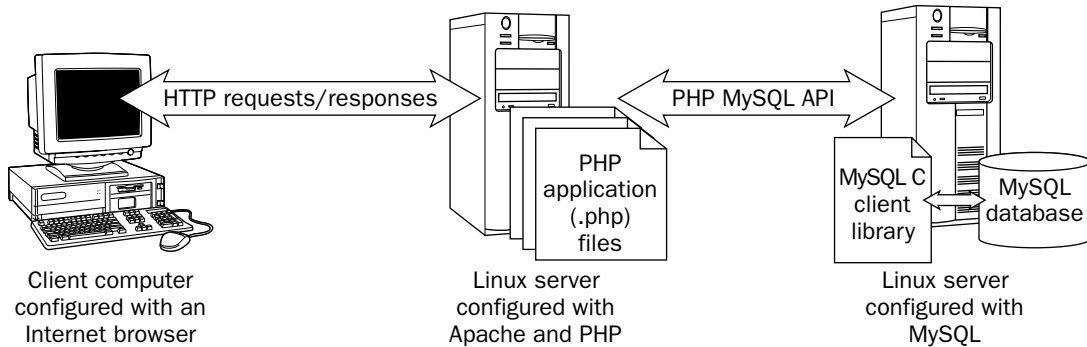


Figure 1-8

As the figure indicates, a number of steps are necessary in order to support a PHP data-driven application. The following list takes a closer look at each step so you can better understand how a PHP application retrieves data:

1. A client computer, running a browser such as Netscape, issues a Hypertext Transfer Protocol (HTTP) request to the PHP application on the Apache Web server. (This would be the same thing as entering a URL in the address box of your browser.) When Apache receives the request, it begins to process the requested page.
2. If a requested page has a .php file extension, the PHP preprocessor kicks in and reads the file. The preprocessor treats everything that is not marked as PHP script as literal text. This would include all your HTML tags that are outside the enclosed PHP script.
3. The PHP preprocessor begins executing the PHP script.
4. If data is requested from a MySQL database, PHP uses the MySQL API to connect to MySQL and request data. The API is based the MySQL C client library, which is part of the MySQL installation.
5. MySQL returns the requested data.
6. The PHP preprocessor incorporates the data retrieved from the MySQL database into the script-execution process and outputs all processed scripts into HTML.
7. The Apache Web server responds to the client computer with an HTML page that includes the information that the PHP preprocessor generates.

Of course, other application programming languages and applications environments work a little differently than what is shown here, but this at least provides you with an overview of a data-driven application. However, note that, although the application server and database server are shown as separate computers, this does not necessarily have to be the case. You can install Apache, PHP, and MySQL on the same computers (which is a typical configuration for developers). In fact, you could also install the browser on the same computer.

Creating a Data-Driven Application

Now that you have an overview of how a data-driven application works, take a look at a sample PHP application that connects to a MySQL database. The following application connects to the Parts table in the MySQL Test database and using a `SELECT` statement to retrieve data from that table.

```
<html>
<head>
<title>The Parts Table</title>
</head>

<body bgcolor="white">

<?php

// Define the variables necessary to access MySQL.
$host="localhost";
$user="ethan";
$pw="pw1";

$db="test";

// Establish a connection to MySQL.
$connection=mysql_connect($host,$user,$pw) or die ("Connection failed!");

// Select the MySQL database.
mysql_select_db($db) or die ("Unable to connect to database.");

// Create an SQL query.
$query="SELECT PartName, PartID FROM Parts ORDER BY PartName";

// Execute the query and store the query results in memory.
$result=mysql_query($query) or die ("Query failed!");

// Process the query results.
if (mysql_num_rows($result) > 0)
{
    echo "<h2>Data from the Parts table<br>in the MySQL Test database<h2>";
    echo "<br>";
    echo "<table cellpadding=10 border=1>";
    echo "<tr bgcolor='#FFCCCC'>
        <td><b>Part Name</b></td>
        <td><b>Part Number</b></td>
    </tr>";
    while($row=mysql_fetch_row($result))
    {
        echo "<tr>";
        echo "<td>" . $row[0] . "</td>";
        echo "<td>" . $row[1] . "</td>";
        echo "</tr>";
    }
    echo "</table>";
}
}
```

```
else
{
    echo "The Parts table is empty.";
}

// Free the memory used for the query results.
mysql_free_result($result);

// Close the connection to MySQL.
mysql_close($connection);

?>

</body>

</html>
```

You can write the application in any text editor, as long as you save the file to the appropriate directory and with a .php extension. The directory to which you save the file depends on your Web server's configuration.

The PHP application above is a data-driven application at its simplest and is not meant to be anything more than that. It appears here only as way to provide you with a basic understanding of how an application connects to a MySQL database and with a more thorough overview of how a data-driven application works. Chapter 18 discusses PHP in greater detail.

As you can see from the application code above, PHP includes a number of elements that allow the application to connect to the database and retrieve data. Examine the code so you can see how to establish the database connection. The file begins with many of the elements typical of an HTML Web page:

```
<html>
<head>
<title>The Parts Table</title>
</head>

<body bgcolor="white">

<?php
```

As you can see, the <html> tag identifies the page, which is followed by a <head></head> section. Next comes the opening <body> tag, which identifies the page's background color. After these tags come the opening <?php tag, which tells the PHP preprocessor should process the following text, up to the closing PHP tag (?>), as PHP script.

If you're not familiar with HTML, you should spend a little time learning about it before you tackle PHP. There are numerous books on the subject and knowledge of HTML can help you better understand how PHP is implemented on an HTML page.

Introducing the MySQL Relational Database Management System

After the opening PHP tag, you can begin to define the PHP elements of your application. In many cases, you want to use variables within your application. A *variable* is a type of placeholder that holds a value in memory during the execution of the script. Variables are particularly useful for information that's repeated or that's likely to change. They're also a useful way to group similar information together, as shown in the following code:

```
// Define the variables necessary to access MySQL.
$host="localhost";
$user="ethan";
$pw="pw1";
$db="test";
```

This code defines four variables:

- ❑ **\$host:** Identifies the server on which MySQL is installed. If you use "localhost," then it refers to the same server where the PHP application resides.
- ❑ **\$user:** Identifies the MySQL user account. In this case, the user account is ethan. This account must already be set up in MySQL before you can use the application.
- ❑ **\$pw:** Identifies the password for the user account named ethan. In this case, the password is pw1.
- ❑ **\$db:** Identifies the database that contains the information that the application must access. In this case, the database is test (a sample database installed by default when you install MySQL).

Once you define your variables, you can use them in functions that establish a connection to MySQL. A *function* is a set of predefined code that performs a specific task. In addition to defining the variables, the code also includes a comment, which is indicated by the double forward slashes that precede the comments. As with SQL statements, comments are a useful way to explain your code and provide information to other developers or reminders to yourself.

Moving on to the next line of code, which uses the `mysql_connect()` function to assign a value to the `$connection` variable:

```
// Establish a connection to MySQL.
$connection=mysql_connect($host,$user,$pw) or die ("Connection failed!");
```

The `mysql_connect()` function uses three of the variables that you defined in the previous section as its arguments. An *argument* is a value that is passed to a function in order for that function to complete its assigned task. The function uses those variables to carry out the task of connecting to the database. The `$connection` variable can then refer to this connection.

The idea of establishing a connection is important to any data-driven application. Most application languages provide some mechanism for opening a connection to a database server. As you can see in the sample PHP application above, the mechanism used in this case is the `mysql_connect` function, which establishes a connection to MySQL, passes the connection parameters to MySQL, and maintains that connection until it is specifically closed.

Chapter 1

Notice that the code also includes an `or die` condition. This is added in case the requested operation fails. In other words, if the `mysql_connect()` function fails to establish a connection to MySQL, PHP returns the phrase "Connection failed!" You can use the `or die` option anywhere an operation should be carried out but could fail. (Using the `or die` method to handle errors is only one method available for checking for errors in PHP.)

The next line of code uses the `mysql_select_db()` function to select the database that contains the targeted data:

```
// Select the MySQL database.
mysql_select_db($db) or die ("Unable to connect to database.");
```

The `$db` variable serves as an argument in the `mysql_select_db()` function to connect to the Test database. Once you select the database, you can define a query, as shown in the following code:

```
// Create an SQL query.
$query="SELECT PartName, PartID FROM Parts ORDER BY PartName";
```

As you can see, you assign the query to the `$query` variable. By assigning the query to a variable, you can call that query in your code whenever necessary. The query in this case is a `SELECT` statement that retrieves data from the `Parts` table. (The `Parts` table must include data in order for this statement to return any information.) Once you define the query, you can use the `mysql_query()` function to execute that query:

```
// Execute the query and store the query results in memory.
$result=mysql_query($query) or die ("Query failed!");
```

The result set (the data returned by the query) that the executed query generates is assigned to the `$result` variable, which processes the query results, as you can see in the following code:

```
// Process the query results.
if (mysql_num_rows($result) > 0)
{
    echo "<h2>Data from the Parts table<br>in the MySQL Test database<h2>";
    echo "<br>";
    echo "<table cellpadding=10 border=1>";
    echo "<tr bgcolor='#FFCCCC'>
        <td><b>Part Name</b></td>
        <td><b>Part Number</b></td>
    </tr>";
    while($row=mysql_fetch_row($result))
    {
        echo "<tr>";
        echo "<td>" . $row[0] . "</td>";
        echo "<td>" . $row[1] . "</td>";
        echo "</tr>";
    }
    echo "</table>";
}

else
{
    echo "The Parts table is empty.";
}
```

Introducing the MySQL Relational Database Management System

The code uses a loop and the `mysql_fetch_row()` function to process the result set. The results are placed into a table that includes a row for each row in the result set. The loop is based on an `if . . . else` construction: *if* the results contain rows, a table is created and the data is displayed, or *else* the user receives a message saying that the Parts table is empty.

This chapter doesn't go into too much detail about creating a loop in order to process a result set. Chapter 18 explains loops and other elements of PHP in greater detail.

Once you process the result set, you can use the `mysql_free_result()` function to free up the memory used for the query results:

```
// Free the memory used for the query results.
mysql_free_result($result);
```

Notice that you again use the `$result` variable as an argument in the `mysql_free_result()` function. This allows you to easily clear the memory used for the result set (which could grow quite large in some cases). Once you clear the memory, you can use the `mysql_close()` function and the `$connection` variable to close the connection to MySQL:

```
// Close the connection to MySQL.
mysql_close($connection);
```

The final pieces of the application are the closing tags for PHP (`?>`), the body section (`</body>`), which encloses all the PHP script, and the HTML page (`</html>`):

```
?>
</body>
</html>
```

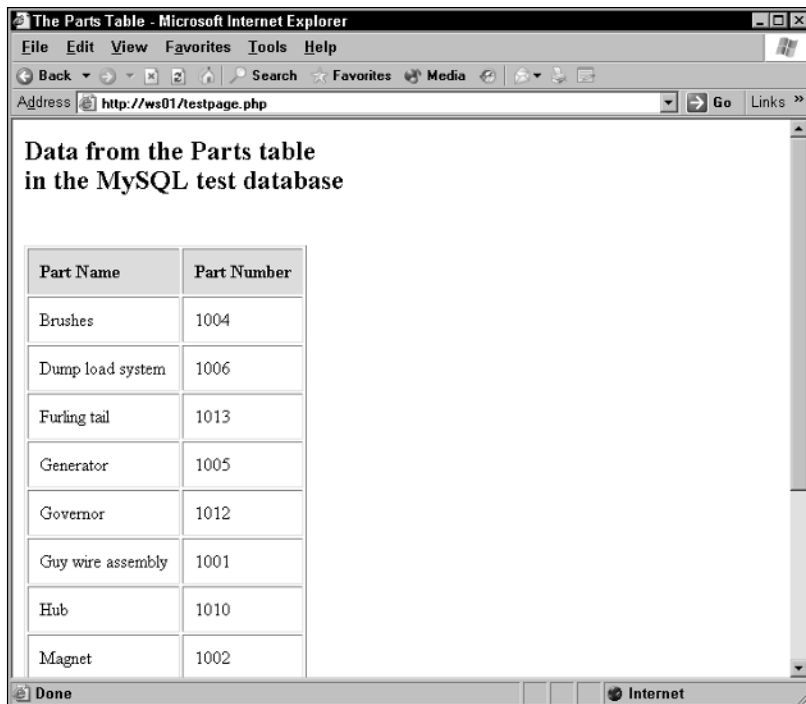
This completes your PHP application. Despite its simplicity, it demonstrates how you would connect to a MySQL database from within a PHP application. If you create applications in other languages, you find similar constructions for establishing a connection and accessing data. Basically, an application, in order to use an API to connect to a MySQL database, must be able to do the following:

- Establish a connection to MySQL.
- Select a database.
- Define an SQL query.
- Execute the query and, if necessary, make the results accessible for processing.
- Process the query results.
- Free up memory and, when appropriate, close connections.

Chapter 1

Now that you've seen the code that makes up a PHP application, take a look at the results you can expect if you implement that application. Figure 1-9 shows how this Web page might look if you access it through a browser. (This example uses Internet Explorer.) As you can see from the figure, the query results display in a table. Because the table contains more rows than can be displayed in the viewable part of the table, you must scroll down to see the rest of the data.

Based on the example provided here, you should now have a better idea of how a data-driven application accesses a MySQL database. At this point, it might be helpful to refer once again to Figure 1-8 and take a look at how the Apache/PHP server implements the PHP application that you just created and how the application uses PHP MySQL API to connect to the MySQL database. As Figure 1-8 illustrates, a number of components make up a data-driven application, including the application files, the preprocessor, the Web server, and the RDBMS, which relies on SQL to provide data access and management. Once you have a good overview on how the pieces fit together, you are better equipped to build applications that can connect to your MySQL database.



Part Name	Part Number
Brushes	1004
Dump load system	1006
Furling tail	1013
Generator	1005
Governor	1012
Guy wire assembly	1001
Hub	1010
Magnet	1002

Figure 1-9

Summary

One of this book's main objectives is to provide you with the examples and hands-on experience you need to work with MySQL and create applications that connect to MySQL databases. And certainly in the chapters that follow, you perform a number of exercises that demonstrate the various concepts that

Introducing the MySQL Relational Database Management System

the chapters introduce. However, this chapter is a bit of a departure from that approach. Although it provided multiple examples to help illustrate various concepts, this chapter contained no hands-on exercises that allowed you try out these concepts. Instead, this chapter focused primarily on the background information you need in order to move through the subsequent chapters.

The reasoning behind this approach is twofold. First, in order to properly understand MySQL, work with MySQL databases, and build data-driven application, you need the background information necessary to provide a solid foundation for performing actual tasks, whether working directly with MySQL or creating applications that access MySQL databases. Without the necessary foundation, you may not thoroughly understand many concepts, making it more difficult to apply what you learn here to other situations

Another reason that this chapter focused on providing you with the background information, rather the delving immediately into hands-on exercises, is because perhaps the most effective way to learn about a technology is to be presented with information in a logical order. For example, it would be counterintuitive to provide you with an in-depth discussion about modifying data without first discussing how to create the tables that hold the data. Likewise, discussing how to create tables before walking you through the installation process and getting you started using MySQL is illogical. (The `CREATE TABLE` statement that you saw in this chapter was provided only as a method to explain how to work with SQL syntax.) For these reasons, this chapter provided you with a solid introduction to the essentials of relational databases, SQL, and data-driven applications, which includes the following concepts:

- ❑ A comprehensive overview of databases, relational databases, and RDBMSs, including the MySQL RDBMS
- ❑ An understanding of the differences between hierarchical, network, and relational databases
- ❑ A foundation in SQL, including its history and characteristics, and in how SQL is implemented in MySQL
- ❑ An overview of how to work with SQL statement syntax and how to create an SQL statement based on that syntax
- ❑ An understanding of the different types of SQL statements (DDL, DML, and DCL) and the different methods used to execute those statements
- ❑ An overview of the components that make up a data-driven application
- ❑ A basic knowledge of how a PHP application connects to a MySQL database to retrieve and process data

With a strong background in each of these areas, you are ready to move on to the rest of the book. In the next chapter, you learn how to install MySQL, and from there, you learn how to use the tools that MySQL provides. Once you install MySQL and understand how to work within the MySQL environment, you can create databases, manage data, and access data from your applications.

Exercises

The following exercises help you build on the information you learned in this chapter. Although this chapter only introduced you to how to create SQL statements and PHP applications, you should still have enough information to respond to the exercises. To view the answers, see Appendix A.

1. Explain the differences between hierarchical, network, and relational databases.
2. Create a two-column table named Employees. Title the first column EmpID, base it on the `INT` data type, and do not permit null values. Title the second column EmpName, base it on the `VARCHAR(40)` data type, and also do not permit null values. The table includes a primary key based on the EmpID column, and the table type is `MYISAM`. How should you define your SQL statement to create the table?
3. Creating a PHP application in which you define the following parameters:

```
$host="localhost";  
$user="linda";  
$pw="password1";
```

You plan to use the `mysql_connect()` function to establish a connection to MySQL. You also plan to use the `mysql_connect()` function to assign a value to the `$connection` variable. In addition, you want to ensure that if the connection fails users receive a message that reads, "Connection failed!" How should you write the PHP code?