1

Drowning in Data, Dying of Thirst for Knowledge

Information may be the most valuable commodity in the modern world. It can take many different forms: accounting and payroll information, information about customers and orders, scientific and statistical data, graphics, and multimedia, to mention just a few. We are virtually swamped with data, and we cannot (or at least we'd like to think about it this way) afford to lose it. As a society, we produce and consume ever increasing amounts of information, and database management systems were created to help us cope with informational deluge. These days we simply have too much data to keep storing it in file cabinets or cardboard boxes, and the data might come in all shapes and colors (figuratively speaking). The need to store large collections of persistent data safely, and "slice and dice" it efficiently, from different angles, by multiple users, and update it easily when necessary, is critical for every enterprise.

Besides storing the information, which is what electronic files are for, we need to be able to find it when needed and to filter out what is unnecessary and redundant. With the informational deluge brought about by Internet *findability*, the data formats have exploded, and most data comes unstructured: pictures, sounds, text, and so on. The approach that served us for decades — shredding data according to some predefined taxonomy — gave in to the greater flexibility of unstructured and semistructured data, and all this can still fit under the umbrella of a database (a broader concept than the "data banks" of the 1970s).

The databases evolved to accommodate all this, and their language, which was designed to work with characters and numbers, evolved along with it. The concept of gathering and organizing data in a database replaced with the concept of a data hub ("I might not have it, but I know where to find it") with your data at the core, surrounded with ever less related (and less reliable) data at the rim.

When does data transform into information? When it is organized and is given a context. Raw data collection does not give you much. For example, the number 110110 could be a decimal number 54 in binary representation; November 1, 2010, the date of D. Hamilton Jackson Memorial Day commemorating establishment of the first press in the U.S. Virgin Islands; House Committee Report #110 for the 110th U.S. Congress (2007–2008), you get the idea.

To transform data into information, you can aggregate the data, add context, cross-reference with other data, and so on. This is as far as databases can take you. The next step, transforming information into knowledge, normally requires human involvement.

DATA DELUGE AND INFORMATIONAL OVERLOAD

One of the reasons behind building a database of your information is to filter the information specific to your needs, to separate the wheat from the chaff. Anybody who uses Internet search engines such as Google or Bing can attest that results brought back are far from being unambiguous because the search engine tries to find the best matches in the sea of relevant, tangentially relevant, and absolutely irrelevant information. Your database is created to serve your unique needs: to track your sales, your employees, and your book collection. In doing so, it might reach out and get some additional information (for example, getting a book's information from Amazon.com), but it will be information specific to your particular needs.

Another important aspect of the database is security. How secure do you need your data? Can anybody see it and modify? Does it need to be protected from unauthorized access due to compliance requirements and simply common sense?

Database management systems, otherwise known as DBMSs, answer all these questions, and more.

Database Management Systems (DBMSs)

What makes a database management system a system? It's a package deal: You get managed storage for your data, security, scalability, and facilities to get data in and out, and more. These are things to keep in mind when selecting a DBMS. The following sections describe a few of the factors that you should consider.

Storage Capacity

Will the selected DBMS be sufficient for current and future needs? If you intend to store your favorite recipes or manage your home library, you might decide to use a desktop database such as Microsoft Access. When you need to store terabytes of information (for example, New York Stock Exchange financial transactions for the last 50 years), you should shop for an enterprise class DBMS such as Oracle, Microsoft SQL Server, or IBM DB2.

Number of Users

If you are the only user of your database, you might not need some of the features designed to accommodate concurrent data use in your database. The current version of Microsoft Access, for instance, supports up to 255 concurrent users (in practice, actual numbers will depend on many factors, including network, bandwidths, and processing power). And with advanced clustering technologies, there is theoretically no limit on the number of users in an enterprise DBMS such as Oracle.

Security

How secure do you want your data to be? You might not be overly concerned if your favorite recipes are stolen, but you'd want your banking or health information to be as secure as possible (and there are regulations to mandate certain levels of protection for various kinds of data collected). One of the major differentiators between enterprise class DBMSs and their desktop counterparts is a robust, finely grained security implementation. A simple file that is a Microsoft Access database is more insecure than a server-based IBM DB2 installation with multiple levels of protection.

Performance

How fast does your database need to be? Can you wait minutes for the information to come back, or must you have a subsecond response, as in a stock trading platform? The answers tie into the question about concurrent users and also scalability. Some DBMSs are inherently slower than the others, and should not be deployed in environments they cannot handle.

Scalability

As Yogi Berra used to say, “Predictions are hard, especially about the future.” Databases must be able to accommodate changing business needs. While one cannot anticipate all the changes down the road, one could make an educated guess based upon likely scenarios and industry trends. Your business will change (growths, acquisitions), and your database needs will change with it. You can bet that your data will live longer than the database it lives in. The operating system might change (mainframe, UNIX/Linux, Windows); the programming environments might change (COBOL, C/C++, Java, .Net); regulations might change, but your data must endure, and not entirely for sentimental reasons.

Any of the modern enterprise DBMSs will get a decent score on any of these factors; ultimately, your business needs will dictate the technology choice. Expert advice will be needed for large production deployment, and qualified database administrators to keep your database in the best shape possible. Once you master the language, your data could be transformed into information; it will be up to you to take it to the next level: knowledge.

Costs

Of course, it is important to consider costs associated with installing and operating a database. Vendors might charge hundreds of thousands of dollars for an enterprise class DBMS or it could be had for free as an open source DBMS. Remember: “There ain’t no such thing as a free lunch.” An open source DBMS might save you money in upfront costs, but would quickly catch up in expertise, time, tools availability, and maintenance costs later on. The total cost of ownership (TCO) must be considered for every DBMS installation.

Recording Data

As far as recorded history goes, humans kept, well, records. Some philosophers even argue that one of the major differences between humans and animals is the ability to record (and recall) past events.

Oral Records

In all probability, oral records were the first kind of persistent storage that humans mastered. The information was transmitted from generation to generation through painstaking memorization; mnemonic techniques such as melody and rhyming were developed along the way. Information transmitted orally was highly storage-dependent, and could deteriorate (as in a game of Chinese whispers) or disappear altogether after an unfortunate encounter by the bearer with a lion, a shark, or a grizzly bear.

Pictures

Pictures such as petroglyphs or cave paintings were much sturdier and somewhat less dependent on vagaries in an individual's fate. They were recorded on a variety of media: clay, stone, bark, skin; and some have survived to the modern age. Unfortunately, much of the context for these pictures was lost, and their interpretation became a guessing game for the archeologists.

Written Records

The beginning of written records, first pictographs and then hieroglyphs, dates back to around 3000 BC, when the Sumerians invented wedge-shaped writing on clay tablets, or cuneiform. This activity gradually evolved into a number of alphabets, each with its own writing system, some related, some autochthonous. This opened the door to storing textual information in pretty much the same form that we use even now. The medium for the writing records also improved over time: clay, papyrus, calf skin, silk, and paper.

Printed Word

Recording and disseminating the information was a painstakingly manual process. Each record had to be copied by hand, which severely limited access to information. The next step was to automate the process with printing. First came woodblock printing, with the earliest surviving example in China dating back to 220 AD. This sped up the process dramatically; a single woodblock could produce hundreds of copies with relatively little effort. The invention of movable type, first by the Chinese and Koreans (1040 and 1230, respectively) and then by Johannes Gutenberg in 15th century Europe, led to dramatically increased access to information through automated duplication. Still, single storage (book) could only be used by a single user (reader) at a time, and searching was a painstaking manual process, even with invention of indexing systems (a list of keywords linked to the pages where these keywords were used).

All of the Above

Technological advances made it possible to accumulate information in a variety of media (text, pictures, and sounds). Not until electronic data storage was developed did it become possible to store them all together and cross-reference them for later automated retrieval. The data had to be digitized first.

Analog versus Digital Data

Up until the invention of the first computers, most information was created and stored in human-readable format. Various mechanical systems were invented to facilitate storage and retrieval of the information, but the information itself remained analogous: print, painting, and recorded sound. Sounds recorded on LP disks are analog, and sounds recorded on CD are digital. The most dedicated audiophiles claim that a CD is but an approximation of the real sound (and they are correct), but most people do not notice the difference. One cannot deny the convenience afforded by a digital CD (or, better yet, an audio file stored on one's computer).

The idea to represent data in binary format came independently to several people around the world, with MIT engineer Claude Shannon formulating principles of binary computation in 1938, and German scientist Konrad Zuse creating a fully functional binary computer in 1941. It turns out that a binary system is uniquely suited for the electrical signal processing; it was humans' turn to adapt to a machine.

The familiar letters and punctuation were translated into combinations of ones and zeroes, starting with the Extended Binary Coded Decimal Interchange Code (EBCDIC), developed by IBM in the early 1950s; through the American Standard Code for Information Interchange (ASCII) character-encoding scheme introduced in the early 1960s; to the advent of Unicode, which made its debut in 1991. The latter system was designed to accommodate every writing system on Earth, and can currently represent 109,000 characters covering 93 distinct scripts.

While initial efforts were focused on representing characters and numbers, the other types of data were not far behind. Pictures and then sounds became digitized and eventually made their way into databases.

To Store or Not to Store?

In 1956, IBM was selling five megabyte persistent storage drives for a whopping $10,000 per megabyte (no wonder it had to make this agonizing decision to store dates as two digits instead of four; also known as the Y2K problem); this came down to just under $200 per megabyte in 1981 (Morrow Designs). In August 2010, a Western Digital 1 terabyte hard drive was selling for $70, which translates into 122 megabytes per one cent!

When storage was dear, people had to be very selective about what data they wanted to keep; with costs plummeting, we've set our sights on capturing and storing *everything*.

The Holy Grail of the DBMS for years was to structure and organize data in a format that computers could manipulate; the preferred way was to collect the data and sort it, and then store it in bits and pieces into some sort of a database (it was called a *data bank* in those days, with policies to match). You had to own all your data. With the proliferation of the Internet, this is no longer the case. Distributed data is now the norm; instead of bringing the data in, you might choose to store information about where the data could be found and leave it at that.

Of course, you may need to keep some of your data closer to the vest (financial data and personal data, for example). Storing the actual data will give you full control of how this data is accessed and modified; this is what databases do best.

With all this dizzying variety of data formats, one needs to make a decision on how this data is to be stored. Despite advances in processing unstructured data, organizing it into taxonomies (a process called *data modeling*; see Chapters 2 and 3 for more information) has distinct benefits both in speed and flexibility. Breaking your data down into the smallest bits and pieces requires a lot of upfront effort, but it gives you an ability to use it in many more ways than when stored as monolithic blocks. Compare a Lego bricks castle with a premolded plastic castle. The latter stays a castle forever, while the former could be used to build a racing car model, if needed. The tradeoffs between structured and unstructured data (and everything in-between) will be discussed in Chapter 11.

Relational Database Management Systems

This book is about SQL, the language of relational databases, or relational database management systems (RDBMSs). Since the theoretical foundations was laid down in the 1970s by Dr. Codd, quite a few implementations have come into existence, and many more are yet to come.

Many people consider DB2 to be the granddaddy of all databases, given that the very term *relational* was introduced by IBM researcher Dr. Edgar Frank Codd in 1969, when he published his paper, "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks" in an IBM research report. This assertion is contested by others who point to Oracle's version 2 commercial release in 1979; Multics Relational Data Store sold by Honeywell Information Systems in 1976; or the Micro DBMS experimental designs (pioneering some of the principles formulated by Dr. Codd two years later) of the University of Michigan from 1968 (the last instance of Micro DBMS in production was decommissioned in 1998). The RDBMS road is marked by a multitude of milestones (and an occasional gravestone) of other RDBMS products, including IBM PRTV (1976); IBM SQL/DS (1980); QBE(1976); Informix (1986); Sybase (1986); Teradata (1979); and Ingres, an open source project that gave inspiration to many other successful systems such as PostgreSQL (1996), Nonstop SQL (1987), and Microsoft SQL Server (1988) — to mention but a few. These systems used different dialects of primordial SQL: SEQUEL, QUEL, Informix-SQL, and so on. It was not until 1987 when the first attempt was made to standardize the language; arguably, the battle is still going on.

The current RDBMS market is split among heavyweight proprietary relational databases Oracle (48 percent), IBM (25 percent), and Microsoft (18 percent); smaller proprietary systems Teradata and Sybase, each with a distant 2 percent; and the other vendors, as well as open source databases, comprising about 10 percent of the total market.

For a sizeable enterprise, selecting a database foundation for their applications is a decision not to be taken lightly. Not only does it cost tens of thousands of dollars in upfront licensing fees for the software, and hundreds of thousands of dollars in maintenance and support fees, but it is also an important factor in determining the overall enterprise architecture that aligns all other investments in software, hardware, and human resources. Although migrating from one RDBMS to another became easier in recent years, still the mere thought of it might give your CFO nightmares.

IBM DB2 LUW

IBM is a long-term front-runner in the RDBMS arena, from the mainframe world with the MVS family of operating systems, to z/OS, and later to UNIX and Windows. The current version is IBM DB2 9.7 LUW (Linux, UNIX, and Windows).

The IBM DB2 9.7 keeps the absolute record in transaction processing speed (see Chapter 9 for more information) and comes in a variety of editions, from Advanced Server Enterprise to a free (albeit limited) DB2 Express-C edition used to run samples provided with this book.

DB2 in its version 9.7 is still only compliant with the ANSI/ISO SQL 92 Entry standard (see later in this chapter) and supports some of the more advanced features from other standards organizations such as the Open Geospatial Consortium, JDBC, X/Open XA, as well as bits and pieces of the latest SQL:2008 Standard. In addition to its own built-in procedural extension language, SQL PL, it also provides support for Oracle's PL/SQL, Java, and even Microsoft's .NET family languages for creating stored procedures (see Chapter 4 for more information).

Oracle

Oracle traces its roots back to the first release of Oracle version 2 in 1979, initially for older VAX/VMS systems, with UNIX support following in 1983. Over the years, it added support for most of the features specified in SQL Standard, culminating in the latest release of Oracle 11g, which claims compliance with the "many features" of the latest release of SQL:2008 Standard.

Oracle holds second place in the high-performance transaction processing benchmarking and is at the center of the company's ecosystem. It is a secure, robust, scalable, high-performance database that has dominated the UNIX market for decades. In addition to SQL support, it comes with a built-in procedural language, PL/SQL (see Chapter 4 for more information on procedural extensions), as well as support for general programming languages such as Java.

At of the time of this writing, the latest version is Oracle 11g; the free express edition is available only for Oracle 10g, which has some limits on the data storage size and number of processors (CPUs) the RDBMS is capable of utilizing. The express edition has full support for all SQL features discussed in this book.

Microsoft SQL Server

SQL Server began as partnership between Sybase, Microsoft, and Ashton-Tate, with the initial idea to adapt existing UNIX-only Sybase SQL Server to then-new IBM operating system OS/2. Ashton-Tate later dropped out of the partnership, and the IBM OS/2 operating system faded into oblivion. Microsoft and Sybase were to share the world, being careful not to step on each other's toes. Microsoft was to develop and support SQL Server on Windows and OS/2, and Sybase was to take over UNIX platforms. The partnership formally ended in 1994, although at its core, Microsoft SQL Server still used fair chunks of Sybase technology. In 1998, beginning with the release of Microsoft SQL Server 7.0, the last traces of Sybase legacy were eliminated, and a brand spanking new RDBMS set out to conquer the world (the Windows world, that is). As of today, Microsoft holds about 20 percent of the RDBMS market, though on Windows it reigns supreme.

The latest version as of this writing is Microsoft SQL Server 2008 Release 2; a limited Express edition available for free that supports all features of SQL covered here.

Microsoft Access

Microsoft Access, known lately as Microsoft Office Access, is a desktop relational database (relatively relational, as some might quip). It purports to be an integrated solution combining elements of a relational database engine, application development infrastructure (complete with built-in programming language and programming model), and reporting platform. Unlike other RDBMSs discussed in the book, this is a file-based database and as such has inherent limitations in performance and scalability. For example, while the latest version theoretically allows for up to 255 concurrent users, in practice anything more than a dozen users slows the performance to a crawl. It also supports only a subset of SQL Standard, as well as a number of features available in its own environment only.

One of the features is linking in tables from remote databases that allow it to be used as an application front end to any ODBC/OLEDB-compliant database.

PostgreSQL

PostgreSQL evolved from a project at the University of California at Berkeley lead by Michael Stonebraker, one of the pioneers of the relational databases theory. The principles that went into the original Ingres project, and its successor PostgreSQL, also found their way into many other RDBMs products such as Sybase, Informix, EnterpriseDB, and Greenplum.

The first version of PostgreSQL (with this exact name) came in 1996; it was released in version 6.0 the next year, and remained an open source project maintained by group of dedicated developers. There are numerous commercial versions of PostgreSQL; most notable is EnterpriseDB, a private company that offers enterprise support (along with variety of proprietary management tools) for the product and has convinced many high-profile customers such as Sony and Vonage to rely on an open source RDBMS for some critical enterprise class applications.

PostgreSQL is arguably the closest in terms of support for the SQL standards in addition to a number of features found nowhere else. Unlike its peers (such as MySQL), it provided referential integrity and transactional support from the beginning. It also comes with built-in support for the PL/pgSQL procedural extension language, as well as the capability to adapt virtually any other language to the same purpose.

MySQL

MySQL was first developed by Michael Widenius and David Axmark back in 1994, with its first release in 1995. It was initially positioned as a lightweight, fast database to serve as the back end for data-driven websites. Even though it was lacking many features of the more mature RDBMS products, it was fast in serving information and "good enough" for many scenarios. (To be really fast, MySQL can bypass referential integrity constraints and ditch transactional support; see Chapters 3 and 10 for additional information.) Plus you could not beat the price; it was free. No wonder it grew up to be the most popular relational database among small- and medium-sized users. There were a number of other free database products on the market that lacked features, near-commercial polish, or both. Not one of the big guys — Oracle, IBM, Microsoft, and Sybase — offered free express versions of their respective RDBMSs back then. MySQL was acquired by Sun Microsystems in 2008, which was subsequently swallowed by Oracle.

Currently, Oracle offers a commercially supported version of MySQL as well as a Community Edition. Following this acquisition, a number of fork versions sprang up, such as MariaDB and Percona Server, committed to maintain free status under the General Public License (GPL), one of the least restrictive open source licenses.

The latest released version of MySQL is 5.5, with version 6 on the horizon. It is multiplatform (Linux/UNIX/Windows), and supports most of the features of SQL:1999; some of the features depend on the selected options (for example, a storage engine).

The storage engine option is a feature unique to MySQL, which allows handling of different table types differently. Each engine comes with unique capabilities and limitations (transactional support, index clustering, storage limits, and so on). A database table could be created with different storage engine options, with the default being MyISAM engine.

HSQLDB and OpenOffice BASE

Hyper Structured Query Language Database (HSQLDB), a relational database management system implemented in the Java programming language, is available as open source under the Berkley Software Distribution (BSD) license (meaning pretty much free for all).

This is a default RDBMS engine shipped with the OpenOffice.org BASE, a desktop database positioned to compete in the same market as Microsoft Access. It is a relational database, robust, versatile, and reasonably fast, and is supported on multiple platforms including Linux, various flavors of UNIX, and Microsoft Windows. It claims to be almost fully compliant with SQL:1992 Standard, which covers most of the SQL subset discussed in this book.

An adaptation of HSQLDB serves as an embedded back end to the OpenOffice.org suite component BASE and became part of the suite starting with version 2.0. Like Microsoft Access, the OpenOffice BASE can connect to a variety of RDBMSs, provided that there is a suitable driver; a number of Java Database Connectivity (JDBC) and ODBC (Open Database Connectivity) drivers are available and ship with the product.

Following Oracle's acquisition of OpenOffice and its uncertain status as an open source project under Oracle's patronage, the OpenOffice.org community decided to start a new project called LibreOffice*, with the intent of implementing all the functionality of OpenOffice as free software under the original BSD license.*

Relational databases are not the only game in town. Some of the older technologies, seemingly forever defeated by relational database theory, came back, helped by ever faster/cheaper hardware and software innovations. The quest for better performance and ease of creating applications spawned research into columnar and object-oriented databases, frameworks that make the "all data in one bucket" approach workable, domain-specific extensions (such as geodetic data management or multimedia), and various data access mechanisms. We discuss this topic in Chapter 12.

WHAT IS SQL?

Before the advent of commercially available databases, every system in need of persistent storage had no choice but to implement its own, usually in some proprietary file format (binary or text) that only this application could read from and write to. This required every application that used these files to be intimately familiar with the structure of the file, which made switching to a different storage all but impossible. Additionally, you had to learn a vendor-specific access mechanism to be able to use it. Relational model dealt with complexities of data structures, organizing data on logical level, but it had nothing to say about the specifics of storage and retrieval except that it had to be set-based and follow relational algebra rules. Left to their own devices, the early RDBMSs implemented a number of languages, including SEQUEL, developed by Donald D. Chamberlin and Raymond F. Boyce in the early 1970s while working at IBM; and QUEL, the original language of Ingres. Eventually these efforts converged into a workable SQL, the Structured Query Language.

SQL is a RDBMS programming language designed to define relational constructs (such as schemas and tables) and provide data manipulation capabilities. Unlike many programming languages in general use, it does not exist outside the relational model. It cannot create stand-alone programs; it can only be used inside RDBMSs. This is a declarative type of language. It instructs the database about what you want to do, and leaves details of implementation (*how* to do it) to the RDBMS itself. In Chapter 2, we will go over the elements of the language in detail.

From the very beginning there were different dialects bearing the same SQL name, some of them quite different from each other. This worked for the vendors, as it assured lock-in to specific technology, but it also defied the purpose of creating SQL in the first place.

The SQL Standard

To bring greater conformity among vendors, the American National Standards Institute (ANSI) published its first SQL Standard in 1986 and a second widely adopted standard in 1989. ANSI released updates in 1992, known as SQL92 and SQL2, and again in 1999: SQL99 and SQL3. Each time, ANSI added new features and incorporated new commands and capabilities into the language.

The ANSI standards formalized many SQL behaviors and syntax structures across a variety of products. These standards become even more important as open source database products (such as MySQL, mSQL, and PostgreSQL) grow in popularity and are developed by virtual teams rather than large corporations.

The SQL Standard is now maintained by both ANSI and the International Standards Organization (ISO) as ISO/IEC 9075 standard. The latest released standard is SQL:2008, and work is underway to release the next version of the standard to accommodate new developments in the way RDBMSs collect and disseminate data.

Dialects of SQL

Even with a standard in place, the constantly evolving nature of the SQL Standard has given rise to a number of SQL dialects among the various vendors and products. These dialects most commonly evolved because the user community of a given database vendor required capabilities in the database before the ANSI committee created a standard. Occasionally, though, a new feature is introduced by the academic or research communities due to competitive pressures from competing technologies. For example, many database vendors are augmenting their current programmatic offerings with Java (as is the case with Oracle and Sybase) or .Net (Microsoft's SQL Server Integration Services, embedded common language runtime [CLR]).

Nonetheless, each of these procedural dialects includes conditional processing (such as that controlled through IF … THEN statements), control-of-flow functions (such as WHILE loops), variables, and error handling. Because ANSI had not yet developed a standard for these important features at the time, RDBMS developers and vendors were free to create their own commands and syntax. In fact, some of the earliest vendors from the 1980s have variances in the most fundamental language elements, such as SELECT, because their implementations predate the standards. Some popular dialects of SQL include the following:

➤ **PL/SQL —** Found in Oracle. PL/SQL, which stands for Procedural Language/SQL and contains many similarities to the general programming language Ada; IBM DB2 added (limited) support for Oracle's PL/SQL in version 9.5.

- **Transact-SQL —** Used by both Microsoft SQL Server and Sybase Adaptive Server. As Microsoft and Sybase have moved away from the common platform they shared early in the 1990s, their implementations of Transact-SQL have also diverged, producing two distinct dialects of Transact-SQL.
- **SQL PL —** IBM DB2's procedural extension for SQL, introduced in version 7.0, provides constructs necessary for implementing control flow logic around traditional SQL queries and operations.
- **PL/pgSQL —** The name of the SQL dialect and extensions implemented in PostgreSQL. The acronym stands for Procedural Language/postgreSQL.
- **MySQL —** MySQL has introduced a procedural language into its database in version 5, but there is no official name for it. It is conceivable that with Oracle's acquiring the RDBMS it might introduce PL/SQL as part of the MySQL.

Not the Only Game in Town

Over the years there were many efforts to improve upon SQL and extend it beyond original purpose. With the advent of object-oriented programming, there came demand to store objects in the database; proliferation of Internet and multimedia increased demand for storage, indexing and retrieval of the binary information and XML data, and so on. While SQL standards were keeping pace with these and other demands, some decided to create a better mousetrap and came up with some ingenious ideas. For instance, HTSQL is a language that allows you to query data over Internet HTPP protocol; Datalog was envisioned as a data equivalent of Prolog, an artificial intelligence language; and MUMPS (going back to the 1960s!) mixes and matches procedural and data access elements.

The latest entry came from the NoSQL family of databases that depart from conventional relational database theory and eerily reminds us of a data bucket with key/value indexed storage. We will have a brief discussion about evolution of SQL in the last chapter of this book.

LET THERE BE DATABASE!

There is a bit of groundwork to be performed before we could submit our SQL statements to RDBMSs. If you have followed the instructions in Appendix B, complemented by the presentation slides on the accompanying book sites (both at `www.wrox.com` and at `www.agilitator.com`), you should have an up-and-running one (or all) of the RDBMSs used in this book; alternatively, you should have Microsoft Access or OpenOffice BASE installed. Please refer to Appendix B for step by step installation procedures for the RDBMS, and to Appendix A for instructions on how to install the Library sample database.

The following, with minor modifications, will work in server RDBMSs: Oracle, IBM DB2, Microsoft SQL Server, PostgreSQL, and MySQL. In Microsoft Access and OpenOffice BASE/HSQLDB, you'd need to create a project.

The concept of a database, a logically confined data storage (exemplified by the now rarely used term *data bank*), managed by a program is rather intuitive. When using a desktop database such as Microsoft Access, your database is a file that Access creates for every new project you start; the server-based RDBMSs use a similar concept, though the details of implementation are much more complex. Fortunately, the declarative nature of SQL hides this complexity. It tells what needs to be done, not how to do it.

In the beginning, there was a database. The database we will use throughout the book will contain all the books we have on the shelves; a book tracking database that stores titles, ISBN numbers, authors, price, and so on — quite helpful in figuring out what you have.

The following statement creates a database named LIBRARY in your RDBMS (as long as it is Microsoft SQL Server, IBM DB2, PostgreSQL and MySQL; things are a bit different with Oracle, which subscribes to a different notion of what is considered a database; see Appendix A for more details).

```
CREATE DATABASE library;
```

If you have sufficient privileges in the RDBMS instance, the preceding statement will create a database, a logical structure to hold your data, along with all supporting structures, files, and multitudes of other objects necessary for its operations. You need to know nothing about these; all the blanks are filled with default values. Behold the power of a declarative language!

Oracle's syntax would be similar to this:

```
CREATE USER library IDENTIFIED BY discover;
```

With USER being roughly an equivalent of the DATABASE in other RDBMS. A discussion of the similarities and differences between the two are outside scope of this book.

Of course, there is much more to creating a database that would adequately perform in a production environment; there are a myriad of options and tradeoffs to be considered, but the basic data storage will be created and made available to you with these three words.

Once created, a database can be destroyed just as easily, using SQL's DROP statement; you cannot destroy objects that do not exist (and the RDBMS will warn you about it should you attempt to):

```
DROP DATABASE library;
```

In Oracle, of course, you'd be dropping a USER.

Now the database is gone from your server; in Microsoft Access and OpenOffice BASE, this is equivalent to deleting corresponding files.

Due to certain differences in terms of usage across RDBMSs, the concept of database *is different among various proprietary databases. For example, what SQL Server defines as a database is in a way similar to both the SCHEMA and USER in Oracle, but in the context of this book, these differences are not particularly important.*

Creating a Table

Now that we have a database, we can use it to create objects *in* the database, such as a table. A *table* is place where all your data will be stored, and this is where common sense logic and that of RDBMS begin to diverge.

If your refrigerator is anything like ours, you will have all kind of things held to its surface by magnets, some goofy keepsakes from a trip to a zoo, a calendar sent to you by your friendly insurance agent, your kid's school menu (and school attendance phone line), a shopping list, photos of your dog, photos of your children, the pizza hotline… Think of it as your personal database. You could just stick anything to it: text, pictures, calendars, and what not. In contrast, the RDBMS is much more particular. It will ask you to sort your data according to data types. A detailed discussion of data types will take place later, in Chapter 2. Here, we just stick to the data type most intuitively understood and best dealt with by the RDBMS: the text.

Creating a table is just as easy as creating the database in the previous example, with a minor difference of specifying a name for the table column and its data type:

```
CREATE TABLE myLibrary (all_my_books VARCHAR(4000));
```

The column ALL_MY_BOOKS is defined as a character data type (see Chapter 2 for more information of data types), and it can hold as many as 4,000 characters.

As you might have guessed, there is much more to the CREATE TABLE syntax than the preceding example implies. A full syntax listing all options in any given RDBMS would span more than one page, and mastering these options requires advanced understanding of SQL, for which this book is but a first step.

As you'll see in Chapter 2, a table, once created, can be modified (altered), or dropped from the database altogether. The SQL provides you with full control over the database objects, with power to create, change, and destroy.

TRY IT OUT Creating a Database in Microsoft SQL Server 2008

Creating a database is normally a database administrator's task, especially in a production environment; there are too many options and tradeoffs to consider to leave everything set to the default. For our purposes, we can use the basic syntax, however. There are several ways to create a database in Microsoft SQL Server, and using SQL Server Management Studio Express is arguably the easiest one. Follow these steps:

1. Make sure that you have your SQL Server instance up and running (refer to Appendix B for installation instructions).
2. Start SQL Server Management Studio Express by going to the Microsoft SQL Server 2008 menu option (this exercise assumes that SQL Server is installed on your local computer so you can connect automatically with Windows Authentication).

3. The first screen you see is a prompt to connect to your server. If not already filled by default, select the server type Database Engine, the server name .\SQLEXPRESS (if you followed the instructions in Appendix B; otherwise, select another name from the drop-down box; it only displays instances of SQL Server visible from your computer), and authentication set to Windows Authentication.
4. Click the Connect button.
5. SQL Server Management Studio Express will display a window with several panes; for the purposes of this tutorial, we are only interested in the New Query button located in the upper-left corner of the window, right under the File menu (shown in Figure 1-1). Click the New Query button.

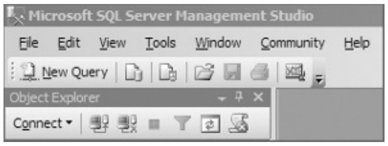

Microsoft SQL Server Management Studio
File Edit View Tools Window Community Help
New Query
Object Explorer
Connect

FIGURE 1-1

6. A new query window would appear in the middle of the window; this is where you will enter your SQL commands.
7. Type in the SQL statement for creating a database:

```
CREATE DATABASE library;
```

8. Click the Execute button located on the upper toolbar, as shown in Figure 1-2.

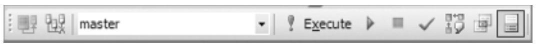


FIGURE 1-2

9. Observe the message “Command(s) completed successfully” in the lower pane, Messages tab.
10. Your newly created database will appear on the Databases list in the pane on the left, with the title Object Explorer (see Figure 1-3). Click the plus sign next to the node Databases node.

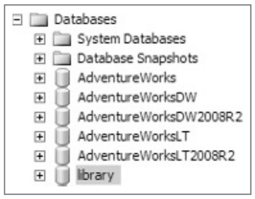


FIGURE 1-3

How It Works

Microsoft SQL Server takes out much of the complexity from creating the database process. Behind the scenes, the SQL Server created several files on the hard drive of your computer (or on an external storage device), created dozens of entries in the Windows registry and the SQL Server–specific configuration files, and created additional supporting objects for the database operations (you can take a look at these by expanding the node LIBRARY in your newly created database).

By omitting all optional configuration options, your database was created using all the default values: storage file names, locations, and initial sizes; collation orders; and so on. While this is not a recipe for creating an optimally performing database (see Chapter 9 for optimization considerations), it will be adequate for the purposes of this book.

Getting the Data In: INSERT Statement

The myLibrary table in our LIBRARY database is now ready to be populated with data, which is a task for the INSERT statement. Since the stated purpose of our database is to keep track of the books, let’s insert some data using one of the books we do have on our shelf, *SQL Bible*. Here is some data.

```
SQL Bible by Alex Kriegel Boris M. Trukhnov Paperback: 888 pages
Publisher: Wiley; 2 edition (April 7, 2008)  Language: English
ISBN-13: 978-0470229064
```

This is a lot of information and all in one long string of characters. The INSERT statement would look like follows:

```
INSERT INTO myLibrary VALUES ('SQL Bible by Alex Kriegel Boris M. Trukhnov
Paperback: 888 pages Publisher: Wiley; 2 edition (April 7,2008)
Language:English ISBN-13: 978-0470229064');
```

The keywords INSERT, INTO, and VALUES are the elements of the SQL language and together instruct the RDBMS to place the character data (in the parentheses, surrounded by single quotation marks) into the myLibrary table. Note that we did not indicate the column name; first because we have but a single column in which to insert, and second because RDBMS is smart enough to figure out what data goes where by matching a list of values to the implied list of columns. Both parentheses and quotation marks are absolutely necessary: the former signifies a list of data to be inserted, and the latter tells the RDBMS that it is dealing with text (character data type).

In database parlance, we have created a record in the table. There are many more books on the shelf, so how do we enter them? One way would be to add all of them on the same line, creating a huge single record. Although that is possible, within limits, it would be impractical, creating a pile of data not unlike the refrigerator model we discussed earlier: easy to add and difficult to find. Do I hear "multiple records"? Absolutely!

The previous statement could be repeated multiple times with different data until all books are entered into the table; creating a new record every time. Instead of a refrigerator model with all data all in one place, we moved onto "chest drawer model" with every book having a record of its own.

TRY IT OUT Inserting Data into a Column

Make sure you are at the step where you can enter and execute SQL commands. Repeat Steps 1 through 6 of the first Try It Out exercise and then run these statements to insert four records in your single table, single column database:

1. Type in (or download from a website) the following queries:

```
USE library;
INSERT INTO myLibrary VALUES ('SQL Bible by Alex Kriegel Boris M. Trukhnov
Paperback: 888 pages Publisher: Wiley; 2 edition (April 7,2008) Language:English
ISBN-13:     978-0470229064');

INSERT INTO myLibrary VALUES ('Microsoft SQL Server 2000 Weekend Crash Course by
Alex Kriegel Paperback: 408 pages Publisher: Wiley (October 15, 2001)
Language:English ISBN-13: 978-0764548406');

INSERT INTO myLibrary (all_my_books ) VALUES ('Letters From The Earth by Mark Twain
Paperback: 52 pages Publisher: Greenbook Publications, LLC (June 7, 2010)
Language:English ISBN-13: 978-1617430060');

INSERT INTO myLibrary (all_my_books ) VALUES ('Mindswap by Robert Sheckley
Paperback: 224 pages Publisher: Orb Books (May 30, 2006)
Language:English ISBN-13: 978-0765315601');
```

2. Click the Execute button located on the upper toolbar, as shown on Figure 1-2.
3. Observe four confirmations "(1 row(s) affected)" in the Messages tab in the lower window.

How It Works

The INSERT statement populates columns in the table by creating a record, a single row of data. The list of columns could be omitted as the list of values corresponds exactly to the list of columns (see later in this chapter for more information). If a column is specified, it has to appear in parentheses without any quotation marks; and the corresponding data goes into the list after the VALUES keyword, in parentheses, with quotation marks around the data to indicate the character nature of the value.

Give Me the World: SELECT Statement

Now that we have our data, we could query it to find out exactly what we have. The SELECT statement will help us to get the data out of the table; all we need is to tell it what table and what column.

```
SELECT all_my_books FROM myLibrary;
```

While it did produce a list of the books' information, it is far cry from being useful. Let's face it; it is a mess of a data, and the only advantage from being stored in a relational database is that it can be easily recalled, and possibly printed. What about search? To find out whether you have a specific book, you'd have to pull all the records and *manually* go over each and every one of them! Hardly a result you would expect from a sophisticated piece of software, which is RDBMS.

We need a way to address specific keywords in the records that we store in the table, such as the book title or ISBN number. A standard programming answer to this problem is to parse the record: chop it into pieces and scroll them in a loop looking for a specific one, repeating this process for each record in the table. The SQL cannot do any of this without vendor-specific procedural extensions. This would defy declarative nature of the language and would require intimate understanding of the data structure. Let's take another look at the first record of data we entered:

```
SQL Bible by Alex Kriegel Boris M. Trukhnov Paperback: 888 pages
Publisher: Wiley; 2 edition (April 7, 2008)  Language: English
ISBN-13: 978-0470229064
```

How would you go about chopping the record into chunks? What would be the markers for each, and how do you distinguish a book title from an author? Using a blank space for this purpose would put "SQL" and "Bible" into different buckets while they logically belong together. How do we know that "by" is a preposition, and not part of the author's name? The answer comes from the structured nature of SQL, which is, after all, a *structured* query language; we need more columns. Splitting the one unwieldy string into semantically coherent data chunks would allow us to address each of them separately as each chunk becomes a column unto its own. Back to the CREATE TABLE (but let's first drop the existing one):

```
DROP TABLE myLibrary;
```

Create a new one according to the epiphany we just had:

```
CREATE TABLE myLibrary
(
     title         VARCHAR(100)
   , author        VARCHAR(100)
   , author2       VARCHAR(100)
   , publisher     VARCHAR(100)
   , pages         INTEGER
   , publish_date  VARCHAR(100)
   , isbn          VARCHAR(100)
   , book_language VARCHAR(100)
)
```

A single column became eight columns with an opportunity to add a ninth by splitting the authors' first and last names into separate columns (this is part of the data modeling process to be discussed in Chapter 3). For now, we've used the same data type, albeit shortened the number of characters, with a single exception: We made the PAGES column a number for reasons to be explained later in this chapter. You might also consider changing the data type of the column PUBLISH_DATE. Normally, a date behaves differently from a character, and the DBMS offers a date– and time–specific data type.

Now that we don't have to dump all data into the same bucket, we can be much more selective about data types, and use different types for different columns. It is not recommended that you mix up the data types when inserting or updating (see later in this chapter) the columns.

We will revisit data types again later in this chapter, and in more detail in Chapter 2.

You might have noticed that we have two "author" columns in our table now, to accommodate the fact that there are two authors. This raises the question of what to do when there is only one author, or when there are six of them. These questions will be explored in depth in a data modeling session in Chapters 2 and 3; here we just note that unused columns are populated automatically with default values, and if you find yourself needing to add columns to your table often, it might be the time to read about database normalization (see Chapter 3).

Now we need to populate our new table. The process is identical to the one described before, only the VALUES list will be longer as it will contain eight members instead of one. All supplied data must be in single quotes with the exception of the one going to PAGES column; quotes signify character data, absence thereof means numbers:

```
INSERT INTO myLibrary VALUES (
    'SQL Bible'
   ,'Alex Kriegel'
   ,'Boris M. Trukhnov'
   ,'Wiley'
   ,888
   ,'April 7,2008'
   ,'978-0470229064'
   ,'English');
```

As long as we keep the order of the values matching the structure of the table exactly, we do not need to spell out the columns, which are the placeholder labels for the data, but if the order is different or if you insert less than a full record (say, three out of eight columns), you must list the matching columns as well:

```
INSERT INTO myLibrary (
     title
   , author
   , book_language
   , publisher
   , pages
   , author2
   , publish_date
   , isbn
)VALUES (
    'SQL Bible'
   ,'Alex Kriegel'
   ,'English'
   ,'Wiley'
   ,888
   ,'Boris M. Trukhnov'
   ,'April 7,2008'
   ,'978-0470229064');
```

Repeat the previous statement with different sets of data for each of the books on the shelf. (Yes, some data entry clerks hate their jobs, too.) Alternatively, you can just download a ready-to-go script from the book's accompanying website, and install it following the instructions in Appendix A. You'll get all you information you need in a structured format, ready to be queried with SQL:

```
INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
 isbn) VALUES ('Microsoft SQL Server 2000 Weekend Crash Course','Alex Kriegel'
,'English','Wiley',408, 'October 15, 2001','978-0764548406');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
 isbn) VALUES ('Mindswap','Robert Sheckley' ,'English','Orb Books',224,'May 30,
2006','978-0765315601');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
 isbn) VALUES ('Jonathan Livingston Seagull','Richard Bach' ,'English','MacMillan',
100, '1972','978-0075119616');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
 isbn) VALUES ('A Short History of Nearly Everything','Bill Bryson'
,'English','Broadway',624, 'October 5, 2010','978-0307885159');
```

What happens if you omit both the column name and the value? The columns listed in the statement will get populated, but the omitted column would stay empty. To signify this emptiness, the SQL marks it as having NULL value.

In the preceding examples, the values for the AUTHOR2 column will be populated with NULL(s). As you will see in Chapter 2, a NULL has a special meaning in the database, and behaves according to rather specific rules.

To save yourself some typing, you might want download scripts for this chapter from `www.wrox.com`*, or from* `www.agilitator.com`*. The installation procedures are described in Appendix A.*

Here is a SELECT query that returns all the records you've entered into the myLibrary table:

```
SELECT  title
      , author
      , author2
      , publisher
      , pages
      , publish_date
      , isbn
      , book_language
) FROM myLibrary;
```

Instead of listing all columns, we could have used a handy shortcut provided by SQL, an asterisk symbol (*) that instructs the RDBMS to fetch back all columns.

```
SELECT * FROM myLibrary;
```

The results of this query eerily resemble what we've just discarded for being unstructured, with a minor distinction: The data is displayed in separate columns. It makes all the difference!

First, we can now combine data in any order by just shuffling the columns around or asking for specific columns instead. For example, to produce a list of authors and titles only, we could just execute this query:

```
SELECT  title
      , author
      , author2
) FROM myLibrary;
```

Second, and much more important, is the ability to address these columns by name in a WHERE clause. This clause serves as a filter, allowing you to select records that match some specified condition, such as all books written by Alex Kriegel or only these published by Wiley. The syntax of the query is very intuitive, and resembles English:

```
SELECT * FROM myLibrary WHERE publisher = 'Wiley';
```

The results of the query list only records where the value stored in the PUBLISHER column equals 'Wiley' (note that the value is also enclosed in single quotes to notify the database that this is a character data type we are comparing).

The WHERE clause allows you to narrow down your search to a specific record or a set of records matching your criteria, as there might be millions of records in your database. This is where power of SQL as a set-based declarative language comes forward. With a simple statement that is not unlike a simple English sentence, you can comb through the records returning only a subset of the result, without worrying how this data is stored, or even where it resides. The previous SELECT statements will return identical results when run in Microsoft Access, Oracle, PostgreSQL, MySQL, SQL Server or IBM DB2.

Another important component of the WHERE clause is the use of operators. The previous query used an equivalence operator, filtering only the records in which the publisher's name equals 'Wiley'. You could just as easily ask for books that were *not* published by Wiley using the non-equal operator:

```
SELECT * FROM myLibrary WHERE publisher <>'Wiley';
```

Several operators could be strung together to provide ever more stringent selection criteria using AND and OR logical operators. For instance, to find a book published by Wiley and written by Alex Kriegel, you might use the following query:

```
SELECT * FROM myLibrary
    WHERE publisher = 'Wiley' AND author= 'Alex Kriegel';
```

The query returned only records satisfying *both* criteria; using the OR operator would bring back results satisfying *either* criterion, and not necessarily together. You need to be careful when using operators as they apply *Boolean* logic to the search conditions, and results might be quite unexpected unless you understand the rules.

The logic of operators will be further explored in Chapter 2, along with syntactical differences among the vendors and precedence rules.

TRY IT OUT Exploring the SELECT Statement

Here, we are going to take SELECT statement for a spin using the Microsoft SQL Server 2008 environment. Repeat Steps 1 through 6 of the first Try It Out exercise to get to the stage where you can enter and execute SQL commands.

1. Type in the following statements to insert data into the table:

```
USE library;
INSERT INTO myLibrary (title, author , book_language , publisher , pages ,
author2 , publish_date , isbn)VALUES ('SQL Bible','Alex Kriegel','English',
'Wiley',888,'Boris M. Trukhnov','April 7,2008' ,'978-0470229064');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('Microsoft SQL Server 2000 Weekend Crash Course','Alex Kriegel'
,'English','Wiley',408, 'October 15, 2001','978-0764548406');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('Mindswap','Robert Sheckley' ,'English','Orb Books',224,'May 30,
2006','978-0765315601');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('Jonathan Livingston Seagull','Richard Bach' ,'English','MacMillan',
100, '1972','978-0075119616');
```

```
INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('A Short History of Nearly Everything','Bill Bryson'
,'English','Broadway',624, 'October 5, 2010','978-0307885159');
```

2. Click the Execute button located on the upper toolbar, as shown earlier in Figure 1-2.
3. Observe five confirmations "(1 row(s) affected)" in the Messages tab in the lower pane window.
4. The following statement will select all rows and all columns from the table (the display of the actual records in these examples are omitted because of space limitations):

```
SELECT * FROM myLibrary;

(5 row(s) affected)
```

5. To narrow the search, add a WHERE clause:

```
SELECT * FROM myLibrary
    WHERE publisher = 'Wiley';

(2 row(s) affected)
```

6. To narrow it even further, specify two filtering criteria in the WHERE clause: only books published by Wiley and only those that have more than 800 pages:

```
SELECT * FROM myLibrary
    WHERE publisher = 'Wiley' and pages > 800;
```

(1 row(s) affected)7. To select only specific columns, execute the following statement:

```
SELECT title , author  FROM myLibrary
title                                             author
-----------------------------------------------  -----------------------
SQL Bible                                         Alex Kriegel
Microsoft SQL Server 2000 Weekend Crash Course    Alex Kriegel
Mindswap                                          Robert Sheckley
Jonathan Livingston Seagull                       Richard Bach
A Short History of Nearly Everything              Bill Bryson

(5 row(s) affected)
```

How It Works

The inserted data is stored in the table, each chunk in a column of its own, together constituting a record; this allows for addressing specific columns by name when selecting the data.

Step 4 instructs the database engine to return all available records from the myLibrary table; instead of listing all columns in the SELECT list, the query uses the asterisk symbol shortcut.

Steps 5 and 6 progressively narrow the returned result set by adding filtering criteria to the query as part of the WHERE clause; they use SQL operators to specify the equality and "greater than" conditions.

The last step demonstrates the ability to select only specific columns for the records returned and addressing them by name. They appear in the order specified in the query regardless of how they were entered or stored in the table.

Good Riddance: the DELETE Statement

Getting rid of unwanted information is just as important as getting it into the database in the first place. In the case of the Library database, a book might be lost or sold, and there is no need to keep the data any longer. The SQL provides a DELETE statement to deal with the situation. To delete all records from a table, you would use the following statement:

```
DELETE FROM myLibrary;
```

There is no need to use FROM keyword in many RDBMS, just a table name would suffice, but some will insist. Now the records are gone, and you have an empty table in the database that you could populate again using the same INSERT scripts found on `www.wrox.com` or `www.agilitator.com`.

Can these records be restored? It depends. In order to be able to undo changes made to the data in the RDBMS, you need to perform all operations in the context of a transaction that, at the end, would either commit all the changes (making them permanent) or roll them back (restoring the data to the original state). We will discuss transactional support in Chapter 10.

The DELETE statement could be much more selective in its approach if used together with WHERE clause you encountered earlier. To delete a specific set of records, you need to specify criteria. The following query will indiscriminately delete all records satisfying the WHERE clause condition:

```
DELETE FROM myLibrary
    WHERE publisher = 'Wiley';
```

All Wiley titles will be gone from your table, which might not be quite what you wanted. How do you pinpoint a single record to be removed from possible thousands sitting in your table? You need to specify a set of criteria that *uniquely* identify this record. Here's an example:

```
DELETE FROM myLibrary
    WHERE publisher = 'Wiley' AND pages = 888;
```

You can't get any more unique than this, right? Actually, you can: Although improbable, it is not impossible for a large database to have more than one record satisfying the previous criteria. The better way is to go by ISBN code that is unique:

```
DELETE FROM myLibrary
    WHERE isbn='978-0470229064';
```

What do you do when a record does not contain an easily identifiable unique marker? There are several ways to ensure the uniqueness of a record in the table (see Chapters 3 and 8), but here we'll introduce a concept of a special column which purpose, among the others, will be to uniquely identify records in the table (also called PRIMARY KEY by the initiated). Had you numbered the records as you entered them into the table, there would be an easy way to refer to a specific record; and assuming that your special column does not allow duplicate numbers, there would be no ambiguity in your deleting a single record. Unfortunately, this would require changing the table structure again.

TRY IT OUT Deleting Records from a Table

Let's delete some records from a table created in Microsoft SQL Server 2008. Repeat Steps 1 through 6 of the first Try It Out exercise to get to the stage where you can enter and execute SQL commands.

1. The following query blows all records from the myLibrary table:

```
USE library;
DELETE myLibrary

(5 row(s) affected)
```

2. Click the Execute button located on the upper toolbar, as shown earlier in Figure 1-2.
3. Insert the records anew:

```
USE library;
INSERT INTO myLibrary (title, author , book_language , publisher , pages , author2
, publish_date , isbn)VALUES ('SQL Bible','Alex Kriegel','English','Wiley',888,
'Boris M. Trukhnov','April 7,2008' ,'978-0470229064');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('Microsoft SQL Server 2000 Weekend Crash Course','Alex Kriegel'
,'English','Wiley',408, 'October 15, 2001','978-0764548406');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('Mindswap','Robert Sheckley' ,'English','Orb Books',224,'May 30,
2006','978-0765315601');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('Jonathan Livingston Seagull','Richard Bach' ,'English','MacMillan',
100, '1972','978-0075119616');

INSERT INTO myLibrary(title, author, book_language, publisher, pages, publish_date,
isbn) VALUES ('A Short History of Nearly Everything','Bill Bryson'
,'English','Broadway',624, 'October 5, 2010','978-0307885159');
```

4. Click the Execute button located on the upper toolbar, as shown earlier in Figure 1-2.
5. Delete a more selective group of records: all books with the exception of those published by Wiley. Type in the following SQL statement, and click the Execute button:

```
DELETE myLibrary
    WHERE publisher <> 'Wiley';

(3 row(s) affected)
```

How It Works

The SQL command submitted to the database engine instructs it to delete all records from the myLibrary table. Five records disappear from the database. In order to continue, you must reinsert the records so you have some data with which to work.

Step 5 demonstrates that the records could be deleted, selectively based upon conditions specified in the WHERE clause of the query. Only three of the five satisfied the criterion *WHERE publisher <> 'Wiley'* and were deleted.

One way to add a new column to a table would be to drop the entire table and re-create it from scratch with a new column; in fact, this was the only way for many RDBMSs for a long time. Now, we just alter the table to sneak a column in (or remove it, for that matter). While the complete syntax is rather complex and differs significantly from RDBMS to RDBMS, the basic syntax is deceptively simple:

```
ALTER TABLE myLibrary
ADD COLUMN book_id INTEGER;
```

This will add an empty column to the myLibrary table of the numeric data type INTEGER. (When it comes to computers, numbers are what they understand best; in fact, the numbers are all they understand.) All human-readable characters, sounds, and pictures are internally represented as long chains of binary numbers: ones and zeroes. To add data to this new column we would have to use the UPDATE statement, the subject of the next section in this chapter.

Some of the DBMSs might have a slightly different syntax for adding columns. For instance, Microsoft SQL Server does not need the keyword COLUMN, inferring what needs to be added from the statement itself, so that the query for SQL Server would look like this:

```
ALTER TABLE myLibrary
ADD book_id INTEGER;
```

Deleting unwanted columns from the table is just as easy except you have to use DROP statement:

```
ALTER TABLE myLibrary
DROP COLUMN book_id;
```

Removing a column requires you to know only its name and that of the table of which it is a part. No data type or any other qualifiers are needed. There are ramifications to be considered when modifying table structure, especially when the table is not empty or columns are being used by some other table in the database. Please see Chapters 2, 7, and 8 for more information.

Notice the distinction between the DELETE and DROP statements: You use DELETE to get rid of the data and you use DROP to destroy database objects such as tables, views, procedures, or the database itself. As you'll learn in Chapter 2, these statements belong to different branches of SQL, data manipulation and data definition languages, respectively.

I Can Fix That: the UPDATE Statement

One of the main benefits of electronic data storage is its flexibility, nothing is written in stone, parchment, or even paper. The data can be created, deleted, or modified at will. So far, you've learned how to get the data in and out, and how to get rid of the data. The UPDATE statement allows you to modify data by changing the existing values for the columns. If you have suddenly discovered that the page number you've entered is wrong, you could fix it by running the following statement:

```
UPDATE myLibrary SET pages = 500;
```

Because the column data type is number (INTEGER), there is no need to enclose 500 in brackets (this is the rule for all numeric data types in all RDBMSs).

The problem with the preceding statement is that the value of 500 will be entered into *every record* in the table, hardly a result we've intended. Just as with DELETE, we have to be much more selective when modifying the data, updating only the records we want to update, and leaving the rest alone. This is the job for the WHERE clause, and again we need some marker that would uniquely identify a record:

```
UPDATE myLibrary SET pages = 500
    WHERE isbn='978-0470229064';
```

If you've discovered that you have more than one column to update for the record, you could add all these to the UPDATE comma-separated list:

```
UPDATE myLibrary SET
    pages = 500
  , title = 'SQL Bible, 2nd Edition'
         WHERE isbn='978-0470229064';
```

The UPDATE operation is implemented in such a way as to allow for using the existing data to be used as a filtering criterion. For instance, you could find the book by its title and change the title in the same query:

```
UPDATE myLibrary SET title = 'SQL Bible, 2nd Edition'
         WHERE title = 'SQL Bible';
```

Of course, after the data is changed, the preceding query won't be able to find the same record again using the same WHERE clause criterion. The same principle could be applied when the new data you're supplying includes the exiting data as a component. To add the '2nd Edition' qualifier to 'SQL Bible' we do not have to supply the whole string, just the second part of it, and use the concatenation operator:

```
UPDATE myLibrary SET title = title + ', 2nd Edition'
         WHERE title = 'SQL Bible';
```

The preceding syntax with the plus sign ('+') as concatenation operator is valid in Microsoft SQL Server only. Oracle and PostgreSQL use the || operator; Microsoft Access uses the ampersand (&); and IBM DB2, MySQL, and HSQLDB prefer to use the SQL function CONCAT. See Chapter 2 for information on SQL operators and SQL functions, respectively.

So far it was implied that columns are being updated with the same data type: characters to characters and numbers to numbers. What happens when you mix the data type and try to insert or update? For example, what would happen if you tried to update a character column with a number? The answer is the same dreaded "it depends." Some RDBMSs will choke on the incompatible data, and spit out an error message; others will try their best within compatibility limits to convert the data into the data type of the column. The latter *modus operandi* is known as implicit data type conversion, whose uses and misuses will be discussed in Chapter 2.

TRY IT OUT Modifying Table Structure with the ALTER Statement, and Table Data with the UPDATE Statement

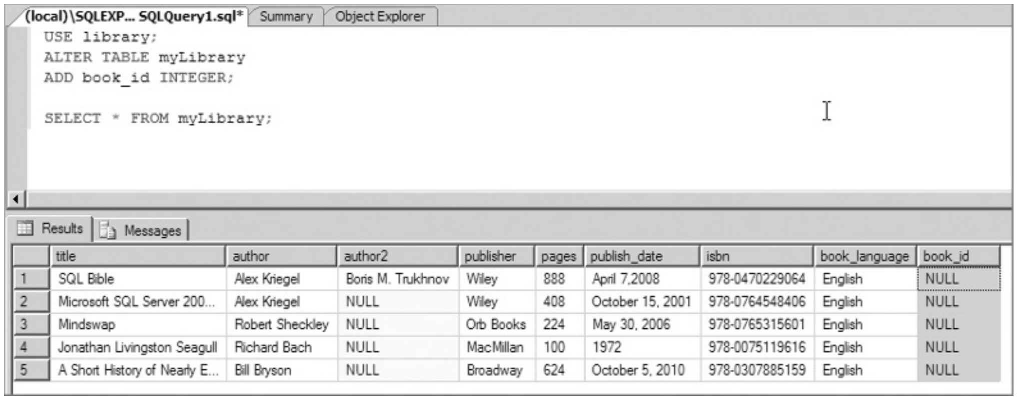
To explore the scenario mentioned earlier, let's add a numeric column to our table and populate it with data in Microsoft SQL Server 2008.

First, we need to make sure we are at the step where we can enter and execute SQL commands. Repeat Steps 1 through 6 of the first Try It Out exercise, repeat the steps to create and populate the myLibrary table as shown in exercises 2 and 3, and then follow these instructions:

1. To add a column to a table, type in the following:

```
USE library;
ALTER TABLE myLibrary
ADD book_id INTEGER;
```

2. Click the Execute button located on the upper toolbar, as shown on Figure 1-2.
3. Observe the message "Command(s) completed successfully" in the lower pane of the Messages tab.
4. Query your table to make sure that the column appears at the end of the data set, and is empty (NULL), as shown in Figure 1-4.



```
USE library;
ALTER TABLE myLibrary
ADD book_id INTEGER;

SELECT * FROM myLibrary;
```

	title	author	author2	publisher	pages	publish_date	isbn	book_language	book_id
1	SQL Bible	Alex Kriegel	Boris M. Trukhnov	Wiley	888	April 7,2008	978-0470229064	English	NULL
2	Microsoft SQL Server 200...	Alex Kriegel	NULL	Wiley	408	October 15, 2001	978-0764548406	English	NULL
3	Mindswap	Robert Sheckley	NULL	Orb Books	224	May 30, 2006	978-0765315601	English	NULL
4	Jonathan Livingston Seagull	Richard Bach	NULL	MacMillan	100	1972	978-0075119616	English	NULL
5	A Short History of Nearly E...	Bill Bryson	NULL	Broadway	624	October 5, 2010	978-0307885159	English	NULL

FIGURE 1-4

5. Now we need to update the new column because all it contains currently is NULL(s). Delete every statement from the query window and type in the following commands:

```
USE library;
UPDATE myLibrary SET bk_id = 1 WHERE isbn='978-0470229064';
UPDATE myLibrary SET bk_id = 2 WHERE isbn='978-0764548406';
UPDATE myLibrary SET bk_id = 3 WHERE isbn='978-0765315601';
UPDATE myLibrary SET bk_id = 4 WHERE isbn='978-0075119616';
```

6. Click the Execute button located on the upper toolbar, as shown on Figure 1-2.
7. Observe four confirmations "(1 row(s) affected)" in the Messages tab in the lower pane window.
8. Verify that the data indeed was inserted by executing a SELECT query against the myLibrary table:

```
USE library;
SELECT bk_id, isbn FROM myLibrary;

bk_id          isbn
-----------    ---------------
1              978-0470229064
2              978-0764548406
3              978-0765315601
4              978-0075119616
NULL           978-0307885159
```

9. The following statement updates all columns in a single query, effectively replacing record #1:

```
USE library;
UPDATE myLibrary SET
    isbn = '978-1617430060'
  , pages = 52
  , title = 'Letters From The Earth'
  , author = 'Mark Twain'
  , author2 = NULL
  , publisher = 'Greenbook Publications, LLC'
  , publish_date = 'June 7, 2010
WHERE bk_id = 1;
```

10. Run the SELECT statement from Step 8 to verify the changes:

```
USE library;
SELECT bk_id, isbn FROM myLibrary;

bk_id          isbn
-----------    ---------------
1              978-1617430060
2              978-0470101865
. . .
NULL           978-0307885159
```

How It Works

The first statement in the batch indicates that the commands are to be executed in the context of the Library database; it only needs to be executed once at the beginning of the session (see Chapter 10 for more information). The ALTER TABLE command adds a column of INTEGER numeric data type to the myLibrary table created in previous exercises; the newly created columns contain only NULL(s) at this point, indicating the absence of any data. The UPDATE statements populate this column for specific records uniquely identified by setting the WHERE clause to filter for the ISBN column in the same table. Without it, the BK_ID column will be updated with the same value for all records.

As you can see from the output produced by the SELECT statement in Step 8, only four records have data in the BK_ID column now; for the rest of the records it is empty.

In Step 9 we are using the UPDATE statement to replace the contents of the entire record, column by column, ending up with a different book in our database. Because the book does not have a co-author, the value is plugged with NULL to indicate absence of any data. Had it been omitted, the column would retain the previous value.

In a multiuser environment, the problems with modifying the data are that somebody else might be reading or modifying it at the same time. This gives rise to a number of potential data integrity problems. The RDBMSs solve this problem with various locking mechanisms discussed in Chapter 10. The trick here is not to overdo it, as locking could potentially slow the database down. A popular open source database (MySQL, for instance) has different storage mechanisms for the databases used mostly to serve the information (SELECT) and those in need of data integrity protection.

SUMMARY

We produce and consume ever-increasing amounts of information, and database management systems were created to help us cope with the informational deluge.

Database management systems (DBMSs) accumulate and manage data in various forms, text, images, and sounds, both structured and unstructured. The underlying format for all electronically stored data is digital. DBMSs built upon the relational model are called RDBMS (Relational Database Management Systems).

The RDBMSs manage both data and access to it, applying security policies, and auditing activity. There is a multitude of databases on the market, from desktop to enterprise class servers, from proprietary to open source. A variety of factors must be considered for each RDBMS package deployment: storage capacity, scalability, security, and costs, to name a few. The most popular enterprise class RDBMS packages include Oracle, IBM DB2, and Microsoft SQL Server; the popular open source contenders are PostgreSQL and MySQL; desktop databases are represented by Microsoft Access and OpenOffice embedded HSQLDB.

The Structured Query Language (SQL) is lingua franca of the relational database management systems (RDBMSs) and has roots in IBM research conducted in the late 1960s. The first attempt to standardize SQL was by the American National Standards Institute (ANSI) in 1986, and the current standard is SQL:2008, endorsed by the International Standards Organization (ISO). Despite the published standard, virtually every RDBMS supports its own dialect of SQL, each being somewhat different in syntax and implementation details. In addition, many RDBMSs support procedural extensions introducing procedural logic in an otherwise set-based declarative language.

For each RDBMS system discussed in the book, the basic element is the table residing in a database. The table organizes data into rows and columns of specific data types; and SQL provides language constructs to insert and manipulate the data trough statements such as INSERT, SELECT, DELETE, and UPDATE.

RDBMSs provide an inherently multiuser environment and facilities to ensure data integrity as different users work with the same data at the same time.