Chapter 1

SQL and Relational Database Management Systems

Information may be the most valuable commodity in the modern world. It can take many different forms — such as accounting and payroll information, information about customers and orders, scientific and statistical data, graphics, or multimedia. 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. These days we simply have too much data to keep storing it in file cabinets or cardboard boxes. The need to store large collections of persistent data safely, "slice and dice" it efficiently from different angles by multiple users, and update it easily when necessary is critical for every enterprise. That need mandates the existence of databases, which accomplish all the tasks listed, and then some. To put it simply, a *database* is just an organized collection of information — with emphasis on "organized."

A more specific definition often used as a synonym for "database" is *database management system* (DBMS). That term is wider and, in addition to the stored information, includes some methods to work with data and tools to maintain it.

NOTE **DBMS can be defined as a collection of interrelated data plus a set of programs to access, modify, and maintain the data. More about DBMS later in this chapter.**

IN THIS CHAPTER

- **Understanding databases**
- **Characteristics of a good database**
- **The database market**
- **Real-life database examples**
- **Brief database history**
- **SQL standards and implementation differences**

Desirable Database Characteristics

There are many differing ideas about what a database is and what it should do. Nevertheless, all modern databases should have at least the following characteristics.

Sufficient capacity

A database's primary function is to store large amounts of information. For example, an order management system for a medium-sized company can easily grow into gigabytes or even terabytes of data; the bigger the company, the more data it needs to store and rely upon. A company that wants to keep historical (archival) data will require even more storage space. The need for storage capacity is growing rapidly; the recent change in how audio and video files are being used is a good example of that. Just five or six years ago, storing movie files in a database, editing and remixing them using online tools, and sharing them on the Internet would sound almost unrealistic, and today it is our day-to-day reality. Just think of the popularity of YouTube and similar services (such as Eyespot, Grouper, Jumpcut, VideoEgg, and so on).

Adequate security and auditing

As was noted previously, enterprise data is valuable and must be stored safely. That means protection of the stored data not only from malicious or careless human activities, such as unauthorized logins, accidental information deletions or modifications, and so on, but also from hardware failures and natural disasters. For many companies, database security is not a "nice-to-have" feature, but rather a mandatory requirement regulated by a number of different level standards, including federal laws, such as the Sarbanes-Oxley Act of 2002 (Sarbox) and the Health Insurance Portability and Accountability Act of 1996 (HIPAA).

Multiuser environment

It is also important to note that, in order to be useful, the information stored in a database must be accessible to many users simultaneously at different levels of security, and, no matter what, the data must stay consistent. For example, if two users try to change the same piece of information at the same time, the result can be unpredictable (such as data corruption), so situations like that have to be handled appropriately by internal database mechanisms. Also, certain groups of users may be allowed to modify several pieces of information, browse other parts of it, and be prevented from even viewing yet another part. (Some company data, such as employee and customer personal information, can be strictly confidential with very restricted access.)

Effectiveness and searchability

Users need quick access to the data they want. It is very important not only to be able to store data, but also to have efficient algorithms to work with it. For example, it would be unacceptable for users to have to scroll through each and every record to find just one order among millions stored in the database — the response to someone's querying the database must be fast, preferably instantaneous.

NOTE **As an analogy, suppose you wanted to find all the occurrences of the word "object" in a book. You could physically browse through the entire book page by page until you reach the end. Or you could use the index and determine that the word is used on pages 245, 246, and 348. This situation is comparable to using bad or good programming algorithms.**

Scalability

Databases must be flexible and easily adaptable to changing business needs. That primarily means the internal structure of database objects should be easily modifiable with minimum impact on other objects and processes. For example, to add a field in a pre-SQL database, you would have to bring the whole dataset offline — that is, make it inaccessible to users, modify it, change and recompile related programs, and so on. This is covered in more detail in the "Database Legacy" section of this chapter.

Another scalability aspect is that data typically lives longer than the hardware and software used to access and manipulate it, so it would not be very convenient to have to redesign the entire database to accommodate the current "flavor-of-the-month" development environment, for example, in case of a takeover or when company management suddenly decided to switch the production environment from Java to C#.

In addition, a small private company with a handful of customers can gradually grow into a large corporation with hundreds of thousands or even millions of users. In this case, it would be very useful to have the ability to start with a simplified, inexpensive (or even free) database edition on a small server with the ability to switch easily to the same vendor's Enterprise version on a powerful multiprocessor machine.

NOTE **A very popular and recent alternative to the idea of a powerful multiprocessor server is the concept of "grid" computing when a virtual supercomputer is being assembled from multiple nodes where each node is a relatively small server. The nodes can be easily added or removed from the system, which significantly improves its scalability.**

User friendliness

Databases are not just for programmers and technical personnel (some would say not for programmers — period). Nontechnical users constitute the majority of all database users nowadays. Accountants, managers, salespeople, doctors, nurses, librarians, scientists, technicians, customer service representatives — for all these and many more people, interaction with databases is an integral part of their work. That means data must be easy to manipulate. Of course, most users will access it through a graphical user interface with a predefined set of screens and limited functionality, but ad hoc database queries and reports are becoming more and more popular, especially among sophisticated, computer-literate users.

NOTE **Consider this. An order management application has a screen to view all orders and another window to browse customers. It can also generate a number of reports, including one to analyze orders grouped by customer. Accountant Jerry is working on a report for his boss and needs to find the 10 customers with the highest debt. He can request a new report from the IT department, but it will take days (or even weeks) because of bureaucratic routine, programmers' busyness, or something else. The knowledge of SQL can help Jerry to create his own ad hoc query, get the data, and finish his report. The use of a GUI tool such as TOAD or WinSQL would make Jerry's task even easier — in this case, he doesn't have to know the exact SQL syntax. The tool can build and execute the SQL statement he needs if only Jerry could specify the tables and columns he wants to retrieve as well as the relationships between these tables.**

Selecting Your Database Software

Every single DBMS on the market follows essentially the same basic principles. There is a wide variety of database products on the market, and it is very difficult for a person without a solid database background to make a decision on what would be the right product to learn or use. The database market is chock-full of different relational database management systems (RDBMSs): IBM DB2, Oracle, Microsoft SQL Server, Sybase, MySQL, and PostgreSQL, to name just a few.

No two systems are exactly alike: There are relatively simple-to-use systems, and there are some that require serious technical expertise to install and operate; some products are free, and others are fairly expensive — all in addition to a myriad of other little things such as licensing, availability of expertise, and so on. There is no single formula to help you in the DBMS selection process, but rather, there are several aspects to consider when making the choice. Here are the most common considerations.

Market share

According to a study by IDC, a subsidiary of International Data Group (IDG), in 2006 the three major DBMS vendors shared over 84 percent of the database market. Oracle accounted for 44.3 percent, IBM about 21.2 percent, and Microsoft SQL Server 18.6 percent. Sybase ranked fourth with 3.2 percent, followed by Teradata (2.8 percent); the rest of the market (less than 10 percent) is shared among dozens (or maybe hundreds) of small vendors or nonrelational "dinosaurs."

It's also worth noticing that the share of the "top three" is constantly growing (at the expense of their smaller competitors). In 1997, the combined share of the "big three" was less than 70 percent. In 2001, it increased to about 80 percent, and today the number is well over 85 percent.

The total market for RDBMS software grew by 14.3 percent from 2005 to 2006. The leader here is Microsoft with a 25-percent growth rate, followed by Oracle (14.7 percent). IBM is slightly below the average with about 12 percent.

The share of all open-source RDBMS vendors is quite insignificant, and according to Gartner, Dataquest research was less than 1 percent of the database market in 2005; however, the growth rate among RDBMS vendors was about 47 percent, thanks to few popular products such as MySQL and PostgreSQL.

Total cost of ownership

The prices for the three major implementations are comparable but could vary depending on included features, number of users, and computer processors from under a thousand dollars for a standard edition with a handful of licenses to hundreds of thousands or even millions for enterprise editions with unlimited user access. Many small database vendor implementations are free; moreover, during the last few years, all "top three" vendors released their own versions of free RDBMS. Oracle has offered a starter XE database since 2005; Microsoft has SQL Server Express available at no cost; and IBM recently released an Express-C edition of DB2.

Skills are a different story. Database expertise is a costly thing and usually is in short supply. On average, Oracle expertise is valued a little higher than comparable expertise for Microsoft SQL Server or DB2. The total cost of ownership (TCO) analysis released by vendors themselves tends to be biased, so use your best judgment and do your homework before committing your company to a particular vendor. Make no mistake about it — this is a long-term commitment, as switching the database vendors halfway into production is an extremely painful and costly procedure.

Support and persistence

One may ask, why spend thousands of dollars on something that can be substituted with a free product? The answer is quite simple: For a majority of businesses, the most important thing is support. They pay money for company safety and shareholders' peace of mind, in addition to all the bells and whistles that come with an enterprise-level product with a big name. (As the adage goes: "No one was ever fired for buying IBM.") First, they can count on relatively prompt support by qualified specialists in case something goes wrong. Second, the company management can make a reasonable assumption that vendors such as IBM, Microsoft, and Oracle will still be around 10 years from now. (Nobody can guarantee that, of course, but their chances definitely look better against the odds of their smaller competitors.) In addition, "free" products rarely scale as well as the costlier products, and are rarely as manageable, as robust, or as clever at optimizing wide varieties of queries over many orders of magnitude of data size. So, the less expensive (and sometimes free) products by smaller database vendors might be acceptable for small businesses, nonprofit organizations, or noncritical projects, but very few serious companies would even consider using them for, say, their payroll or accounting systems.

NOTE **We should mention that more and more companies are using open-source RDBMS for certain tasks as the products become more reliable and more functional. Good examples include Sony Online Entertainment switching to EnterpriseDB (an enterprise-class relational database management system built on PostgreSQL) or Google, Yahoo, and Ticketmaster using MySQL for their key projects. Also, RDBMS tools vendors have started to provide support for some popular open-source database products. For example, TOAD by Quest Software, a database tool popular among Oracle developers, is now available for MySQL. In addition, some serious companies specializing in software support now offer their services for certain open-source products.**

Major DBMS Implementations

One book cannot possibly cover all existing database implementations, so we've decided to concentrate on "the big three": Oracle; IBM DB2 for Linux, UNIX, and Windows; and Microsoft SQL Server. These implementations have many common characteristics. They are all industrial-strength, enterprise-level relational databases (the relational database model and SQL standards are covered later in this chapter). They use Structured Query Language (SQL) standardized by

the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO). All three are able to run on the Windows operating system. Oracle also is available on virtually any UNIX flavor, Linux, Apple Mac OS X Server, IBM z/OS. OpenVMS; DB2 for Linux, UNIX, and Windows runs on AIX, HP-UX, Solaris, Linux, and Microsoft Windows.

NOTE **ANSI is a private, nonprofit organization that administers and coordinates the U.S. voluntary standardization and conformity assessment system. ANSI's mission is to enhance both the global competitiveness of U.S. business and the U.S. quality of life by promoting and facilitating voluntary consensus standards and conformity assessment systems, and safeguarding their integrity. ANSI (`www.ansi.org`) was founded October 18, 1918, and is the official U.S. representative to ISO (`www.iso.org`) and some other international institutions.**

The problem is that none of the databases mentioned earlier is 100-percent ANSI SQL-compliant. Each of these databases shares the basic SQL syntax (although some diversity exists even there), but the language operators, naming restrictions, internal functions, data types (especially date- and time-related), and procedural language extensions are implemented differently.

CROSS-REF **See Chapter 14, "Stored Procedures, Triggers, and User-Defined Functions," for more information on the SQL procedural extensions.**

Table 1-1 compares some data on maximum name lengths supported by different database implementations.

In an ideal world, the standards would rule supreme, and SQL would be freely shared among different implementations for the benefit of humanity. Unfortunately, the reality looks somewhat different. While it is possible to distill a standard SQL understood by all database vendors' products, anything other than some very trivial tasks would be more quickly and efficiently accomplished with implementation-specific features.

TABLE 1-1

Maximum Name-Length Restrictions for Some Database Objects

	IBM DB2 9.5	Microsoft SQL Server 2008	Oracle 11G
Table name length (characters)	128	128	30
Column name length (characters)	128	128	30
Constraint name length (characters)	128	128	30
Index name length (characters)	128	128	30
Number of table columns	1012	1024	1000

Real-Life Database Examples

To say that databases are everywhere would be an understatement. They virtually permeate our lives. Online stores, health care providers, clubs, libraries, video stores, beauty salons, travel agencies, phone companies, and government agencies such as FBI, INS, IRS, and NASA all use databases. These databases can be very different in their nature from one another and usually have to be specifically designed to cater to some special customer needs. This section describes some examples.

NOTE **Most relational databases can be divided into two main categories according to their primary function — *online transaction processing* (OLTP) and *data warehouse* systems. OLTP typically has many users simultaneously creating and updating individual records; in other words, it's volatile and computation-intensive. A data warehouse is a database designed for information processing and analysis, with a focus on planning for the future rather than on day-to-day operations. The information in these is not going to change very often, which ensures the information consistency (repeatable result) for the users. In the real world, most systems are hybrids of these two, unless specifically designed as a data warehouse.**

Order management system database

A typical database for a company that sells building materials might be arranged as follows: The company must have at least one customer. Each customer in the database is assigned one or more addresses, one or more contact phones, and a default salesperson who is the liaison between the customer and the company. The company sells a variety of products. Each product has a price, a description, and some other characteristics. Orders can be placed for one or more products at a time. Each product logically forms an order line. When an order is complete, it can be shipped and then invoiced. Invoice numbers and shipment numbers are populated automatically in the database and cannot be changed by users. Each order has a status assigned to it: `COMPLETE`, `SHIPPED`, `INVOICED`, and so on. The database also contains specific shipment information (bill of lading number, number of boxes shipped, dates, and so on). Usually one shipment contains one order, but the database is designed in such a way that one order can be distributed among more than one shipment. In addition, one shipment can contain more than one order. Some constraints also exist in the database. For example, some fields cannot be empty, and some fields can contain only certain types of information.

You already know that a database is a multiuser environment by definition. It's a common practice to group users according to the functions they perform and security levels they are entitled to. The order management system described here could have three different user groups: sales department clerks enter or modify order and customer information; shipping department employees create and update shipment data; and warehouse supervisors handle products. In addition, all three user groups view diverse database information from different angles, using Web-based reports and ad hoc queries.

This book uses a sample order management system database called ACME for examples and exercises. ACME database is a simplified version of a real production database. ACME has only 13 tables, but an actual production database would easily have several hundred.

CROSS-REF See Appendix B and Appendix F for more detailed descriptions of the ACME sample database and how to install it.

Health care provider database

A health care provider company has multiple offices in many different states. Many doctors work for the company, and each doctor takes care of multiple patients. Some doctors just work in one office, and others work in different offices on different days. The database keeps information about each doctor, such as his or her name, address, contact phone numbers, area of specialization, and so on. Each patient can be assigned to one or more doctors. Specific patient information is also kept in the database (names, addresses, phone numbers, health record birth dates, history of appointments, prescriptions, blood tests, diagnoses, and so on). Customers can schedule and cancel appointments and order prescription drugs either over the phone or using the company Web site. Some restrictions apply — for example, to see a specialist, the patient needs an approval from his or her primary physician; to order a prescription, the patient should have at least one valid refill left; and so on.

Now, what are the main database user groups? Patients should be able to access the database using a Web browser to order prescriptions and make appointments. This is all that patients may do in the database. Doctors and nurses can browse information about their patients, write and renew prescriptions, schedule blood tests and X-rays, and so on. Administrative staff (such as receptionists or pharmacy assistants) can schedule appointments for patients, fill prescriptions, and run specific reports.

Again, in real life this database would be far more complicated and would have many more business rules, but this should give you a general idea of what kind of information a database could contain.

The health provider and order management system databases are both examples of a typical *hybrid* database (although the former is probably closer to an OLTP).

Video sharing and editing database

Online video sharing services, such as YouTube, Eyespot, and Jumpcut, use databases to store, retrieve, and edit video files. The users should be able to save their videos in different formats, share them with others or make them private, and delete the video clips when they are no longer needed.

The two main requirements for such a database are capacity (the media files can be quite big, and the number of video-sharing service members is growing every month) and performance (it is not enough just to store a huge number of large binary files; the users should be able to find the file they need and to play it in their Web browser).

This database is also a hybrid database, but in this case, it's closer to a data warehouse — the ability to search and retrieve data is more important than a capability to update and insert new records quickly.

Scientific database

A database for genome research and related research in molecular and cellular biology can be a good example of a scientific database. It contains gene catalogs for completely sequenced genomes and some partial genomes, genome maps and organism information, and data about sequence similarities among all known genes in all organisms in the database. It also contains information on molecular interaction networks in the cell and chemical compounds and reactions.

This database has just one user group — all researchers have the same access to all the information. This is an example of a data warehouse.

Nonprofit organization database

A database of an antique automobile club can be pretty simple. Also, such an organization would not typically have too many members, so the database is not going to be very large. You need to store members' personal information such as addresses, phone numbers, areas of interest, and so on. The database might also contain the information about the automobiles (brand, year, color, condition, and so on). The automobiles are linked to their owners (members of the club). Each member can have one or more vehicle, and a vehicle can be owned by just one member.

The database would only have a few users — possibly, the chairman of the club, an assistant, and a secretary.

The last two examples are not business-critical databases and don't have to be implemented on expensive enterprise software. The data still has to be kept secure and should not be lost, but in case of, let's say, hardware failure, it probably can wait a day or two before the database is restored from a backup. So, the use of a free database, such as MySQL, PostgreSQL, or even nonrelational Posgres is appropriate. Another good choice might be Microsoft Access, which is part of Microsoft Office Tools (Professional Edition or better). Microsoft Access works well with up to 15 users.

Database Legacy

Flat file, hierarchy, and network databases are usually referred to as *legacy* databases. They represent the ways people used to organize information in prehistoric times — about 40 years ago.

Flat file databases

The flat file database was probably one of the earliest database management systems. The idea behind flat file is a very simple one: one single, mostly unstructured data file. It mirrors "ancient" pre-computer data storage systems: notebooks, grocery lists, and so on. You could compare it to a desk drawer that holds virtually everything — such as bill stubs, letters, and small change. While requiring very little effort to put information *in*, such a "design" becomes a nightmare to get the information *out*, as you would have to scroll through each and every record searching for the right one. Putting relevant data into separate files and even organizing

them into tables (think of a file cabinet) alleviates the problem somewhat but does not remove the major obstacles: data redundancy (the same information, or even worse, *supposedly the same* information, might be stored more than once in different files), slow processing speed ("I know it was there somewhere . . ."), error-prone storage, and retrieval. Moreover, it required intimate knowledge of the database structure to work at all — it would be utterly useless to search for, say, orders information in the expenses file.

For example, you could design a flat database system for an order entry system that gathers information about customers, orders the customers have placed, and products the customers have ordered. If data is accumulated sequentially, your file will contain information about customers, then orders and products, then about some new customer, and so on — all in the order the data is entered (see Table 1-2). Just imagine a task of extracting any meaningful information from this mess, not to mention that a lot of the cells will remain empty. (For "Ace Hardware," what would you put in the Quantity column; or for "Nails," what would you put in the Address column?)

Dissatisfaction with these shortcomings stimulated development in the area of data storage-and-retrieval systems.

NOTE **Excel is often used to create flat file databases.**

Hierarchical databases

The concept of a hierarchical database has been around since the 1960s and — believe it or not — it is still in use. The hierarchical model is fairly intuitive. As the name implies, it stores data in hierarchical structure, similar to that of a family tree, organization chart, or pyramid. You could visualize a computer file system as it is presented through some graphical interface.

The most popular hierarchical database product is IBM's Information Management System (IMS), which runs on mainframe computers. First introduced in 1968, it is still around (after a number of reincarnations), primarily because hierarchical databases provide impressive raw-speed performance for certain types of queries.

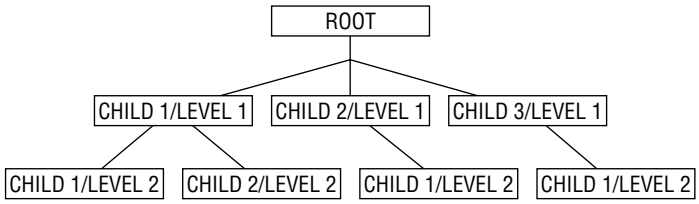
TABLE 1-2

Sample Flat File Records

Name	Type	Address	Price	Quantity
Nails	Product	n/a	100	2000
Ace Hardware	Customer	1234 Willow Ct Seattle, Washington	n/a	n/a
Cedar planks	Product	n/a	2000	5000

FIGURE 1-1

Hierarchical structure



It is based on "parent/child" paradigm in which each parent could have many children but each child has one and only one parent. You can visualize this structure as an upside down tree, starting at the root (trunk) and branching out at many levels (see Figure 1-1).

Because the records in a child table are accessed through a hierarchy of levels, there could not be a record in it without a corresponding pointer record in the parent table — all the way up to the root. You could compare it to a file management system (like a tree view seen in the Microsoft Windows Explorer) — to get access to a file within a directory, you must first open the folder that contains this file.

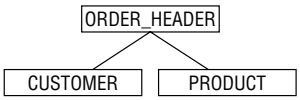
NOTE **The term "table" is not a part of hierarchical database jargon. It was first introduced in the relational database model, discussed later in this chapter. Nevertheless, this book also uses "tables" to refer to hierarchical and network databases' "storage areas" (structured files). After all, a table can be simply defined as a set of elements organized in rows and columns.**

Let's improve upon the previously discussed flat file model. Instead of dumping all of the information into a single file, you are going to split it among three tables, each containing pertinent information: business name and address for the `CUSTOMER` table; product description, brand name, and price for the `PRODUCT` table; and an `ORDER_HEADER` table to store the details of the order.

In the hierarchical database model, redundancy is greatly reduced (compared with the flat file database model). You store information about customer, product, and so on once only. The table `ORDER_HEADER` (see Figure 1-2) would contain pointers to the customer and to the products this customer had ordered. Whenever you need to see what products any particular customer purchased, you start with the `ORDER_HEADER` table, and find the list of IDs for all the customers who placed orders and the list of product IDs for each customer; then, using the `CUSTOMER` table, you find the customer name you are after. Using the products IDs list, you get the description of the products from the `PRODUCT` table.

FIGURE 1-2

Hierarchical database example



Everything works great as long as you are willing to put up with a somewhat nonintuitive way of retrieving information. (No matter what information is requested, you always have to start with the root, which in this example is the ORDER_HEADER table.) Should you need only customers' names, the hierarchical database would be blazingly fast — going straight from a parent table to the child one. To get any information from the hierarchical database, a user has to have an intimate knowledge of the database structure; and the structure itself is extremely inflexible. For example, if you decided that the customers must place an order through a third party, you'd need to rewire all relationships because the CUSTOMER table would not be related to the ORDER_HEADER table anymore, and all your queries would have to be rewritten to include one more step — finding the sales agent who sold this product, and then finding customers who bought it. It also makes obvious the fact that you did not escape the redundancy problem — if you have a customer who places an order through more than one sales agent, you'll have to replicate all the information for each agent in a number of customer tables.

What happens if you need to add a customer that does not have a placed order, or a product that no one yet ordered? You cannot — your hierarchical database is incapable of storing information in child tables without a parent table having a pointer to it. By the very definition of hierarchy, there should be neither a product without an order, nor a customer without an order — which obviously cannot be the case in the real world.

Hierarchical databases handle one-to-many relationships (described in Chapter 2) very well; however, in many cases, you will want the child to be related to more than one parent. Not only could one product be present in many orders, but one order could contain many products. There is no answer (at least not an easy one) within the domain of hierarchical databases.

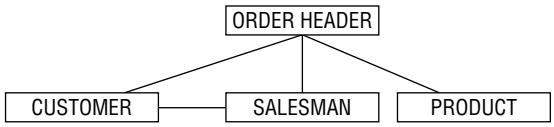
Network databases

Attempts to solve the problems associated with hierarchical databases produced the *network* database model. This model has its origins in the Conference on Data Systems Languages (CODASYL), an organization founded in 1957 by the U.S. Department of Defense. CODASYL was responsible for developing COBOL — one of the first widely popular programming languages — and publishing the Network Database standard in 1971. The most popular commercial implementation of the network model was Adabas (long since converted to the *relational* model).

The network model is very similar to the hierarchical one — it is also based on the concept of parent/child relationships, but it doesn't have the restriction of one child having only one parent. In the network database model, a parent can have multiple children, and a child can have multiple parents. This structure could be visualized as several trees that share some branches. In network database jargon, these relationships came to be known as *sets*.

FIGURE 1-3

Network database example



In addition to the ability to handle a one-to-many relationship, the network database can handle many-to-many relationships.

CROSS-REF **One-to-one, one-to-many, and many-to-many relationships are explained in Chapter 2, "Fundamental SQL Concepts and Principles."**

Also, data access does not have to begin with the root. Instead, you could traverse the database structure starting from any data element and navigating to a related data element in any direction — assuming there are explicit links in both directions (see Figure 1-3).

In this example, to find out what products were sold to what customers, you still would have to start with `ORDER_HEADER` and then proceed to `CUSTOMER` and `PRODUCT` — nothing new here. But things greatly improve for the scenario when customers place an order through more than one agent: you no longer have to go through agents to list customers of the specific product, and you no longer have to start at the root in search of records.

While providing several advantages, network databases share several problems with hierarchical databases. Both are very inflexible, and changes in the structure (for example, adding a new table to reflect changed business logic) may require that the entire database be rebuilt. Also, set relationships and record structures must be predefined.

The major disadvantage of both network and hierarchical databases was that they are programmers' domains. To answer the simplest query, one had to create a program that navigated database structure and produced an output. Unlike SQL, this program was written in procedural, often proprietary, languages and required a great deal of knowledge — of both database structure and underlying operating system. As a result, such programs were not portable and took an enormous (by today's standards) amount of time to write.

Relational Databases

The frustration with the inadequate capabilities of network and hierarchical databases resulted in the invention of the *relational data model*. The relational data model took the idea of the network database several steps further. Relational models — just like hierarchical and network models — are based on two-dimensional storage areas (tables) that can be related based on a common field (or a set of fields). However, these relationships are implemented through column values as opposed to a low-level physical pointer defining the relationship.

NOTE **The common misconception is that the term "relational" comes from the relationships between tables. In fact, the word "relation" is a mathematical term that can be conditionally interpreted as "table."**

Tables

A table is a basic building unit of the relational database. It is a fairly intuitive way of organizing data and has been around for centuries. A table consists of rows and columns (called *records* and *fields* in nonrelational database jargon). Each table has a *unique* name in the database — this must be a unique *fully qualified name* that includes schema or database name as a prefix.

NOTE **The dot (.) notation in a fully qualified name is commonly used in the programming world to describe hierarchy of the objects and their properties. This could refer not only to the database objects but also to the structures, user-defined types, and such. For example, a table field in a Microsoft SQL Server database could be referred to as** `ACME.DBO.CUSTOMER.CUST_ID_N` **where** `ACME` **is a database name,** `DBO` **is the table owner (Microsoft standard),** `CUSTOMER` **is the name of the table, and** `CUST_ID_N` **is the column name in the** `CUSTOMER` **table.**

CROSS-REF **See Chapter 4, "Creating RDBMS Objects" for more information on table and other database object names.**

Each field has a unique name within the table, and every table must have at least one field. The number of fields per table is usually limited, the actual limitation being dependent on a particular implementation. Unlike legacy database structure, records in a table are not stored or retrieved in any particular order. Although records can be arranged in a particular order by means of using a *clustered* index (discussed in Chapter 4, "Creating RDBMS Objects"), the task of sorting the record in an RDBMS is relegated to SQL.

A record is composed of a number of cells, where each cell has a unique name and might contain some data. A table that has no records is called an empty table.

Data within the field must be of the same type — for example, the field `AMOUNT` contains only numbers, and the field `DESCRIPTION` contains only words. The set of the data within one field is said to be a column's *domain*.

NOTE **Early databases — relational or otherwise — were designed to contain only text data. Modern databases store anything that could be converted into binary format: pictures, movies, audio records, and so on.**

Good relational design would make sure that such a record describes an *entity* — another relational database term to be discussed later in the book but worth mentioning here. To put it in other words, the record should not contain irrelevant information: for example, the `CUSTOMER` table deals with the customer information only, so its records should not contain information about, say, products that this customer ordered.

NOTE **The process of grouping the relevant data together, eliminating redundancies along the way, is called *normalization* and is discussed in Chapter 2, "Fundamental SQL Concepts and Principles." It is not part of SQL *per se*, but it does impose limits on the SQL query efficiency.**

There is no theoretical limit to the number of rows a table could have, although some implementations impose restrictions. Also, there are (or at least ought to be) practical considerations to the limits: data retrieval speed, amount of storage, and so on.

Relationships

Tables in RDBMS might or might not be related. As mentioned before, RDBMS is built upon relationships between tables, but unlike in legacy databases (hierarchical and network), these relations are based solely on the values in the table columns — these relationships are meaningful in logical terms, not in low-level computer specific pointers. Let's take the example of our fictitious order entry database (the one that you will design, build, and use throughout the book). The `ORDER_HEADER` table is related to the `CUSTOMER` table since both of these tables have a *common set of values*. The field `ORDHDR_CUSTID_FN` (customer ID) in `ORDER_HEADER` (and its values) corresponds to `CUST_ID_N` in `CUSTOMER`. The field `CUST_ID_N` is declared to be a *primary key* for the `CUSTOMER` table and also is the target for a *foreign key* from the `ORDER_HEADER` table (in the form of the field `ORDHDR_CUSTID_FN`).

Primary key

The *primary key* holds more than one job in an RDBMS. As mentioned previously, this is an important component of relationships. It also can carry ordinary data, such as a department number, part code, or employee ID; but its primary role is to identify each record in a table uniquely.

In the days of legacy databases, the records were always stored in some predefined order — if such an order had to be broken (because somebody had inserted records in a wrong order or business rule was changed), then the whole table (and, most likely, the whole database) had to be rebuilt. The RDBMS abolishes this fixed-order rule for the records, but it still needs some mechanism of identifying the records uniquely, and the primary key, based on the idea of a field (or fields) that contains set unique values, serves exactly this purpose.

NOTE **In an RDBMS, the primary key is just a particular case of *candidate key* — a set of one or more columns that has unique value for every row in this table. Each table can have any number of candidate keys, but only one of them can be declared a primary key.**

By its very nature, the primary key cannot be empty. This means that in a table with a defined primary key, the primary key fields must contain non-null data for each record.

NOTE **Although it is not a requirement to have a primary key on each and every table, it is considered good practice to have one. In fact, many RDBMS implementations would**

warn you if you create a table without defining a primary key. Some purists go even further, specifying that the primary key should be *meaningless* in the sense that they would use some generated unique value (such as `EMPLOYEE_ID`) instead of, say, Social Security numbers (despite that these are unique as well).

A primary key could consist of one or more columns — although some fields may contain duplicate values, their combination (set) is unique through the entire table. A key that consists of several columns is called a *composite key*.

NOTE **Although the primary key is a cornerstone for defining relationships in RDBMS, the actual implementations (especially early ones) have not always provided built-in support for this logical concept. In practice, the task of enforcing uniqueness of a chosen primary key was the responsibility of programmers (requiring them to check for existing values before inserting new records, for example). Today, all major relational database products have built-in support for primary keys — on a very basic level, this means that the database does its own checking for unique constraint violations and will raise an error whenever an attempt to insert a duplicate record is made. Chapter 4, "Creating RDMS Objects," covers this topic in more detail.**

Foreign key

Let's go back to the `CUSTOMER` and `ORDER_HEADER` tables. By now, you should understand why the `CUST_ID_N` was designated as a primary key — it a has unique value, no customer can possibly have more than one ID, and no ID could be assigned to more than one customer. To track what customers placed which orders, you need something that will provide a link between customers and their orders.

The `ORDER_HEADER` table has its own primary key — `ORDHDR_ID_N`, which uniquely identifies orders. In addition to that, it will have a foreign key `ORDHDR_CUSTID_FN` field. The values in that field correspond to the values in the `CUST_ID_N` primary key field for the `CUSTOMER` table. Note that, unlike the primary key, the foreign key is not required to be unique — one customer could place several orders.

Now, by looking into the `ORDER_HEADER` table, you can find which customers placed particular orders. The table `ORDER_HEADER` is related to the table `CUSTOMER` through the values of that foreign key. It is easy to find a customer based on orders, or find orders for a customer. You no longer need to know database layout, understand the order of the records in the table, or master some low-level proprietary programming language to query data. You can now run ad hoc queries formulated in a standard English-like language — the Structured Query Language (SQL).

Invasion of RDBMS

In spite of the clear advantages of the relational database model, it took some time for it to become workable. One of the main reasons was the hardware. The logically clear and clean model proved to be quite a task to implement, and even then it required much more in terms of memory and processing power than legacy databases.

The development of relational databases was driven by the needs of big businesses for a medium to gather, preserve, and analyze data. In 1965, Gordon Moore, the cofounder of Intel, made his famous observation that the number of transistors per square inch on integrated circuits (IC) doubles every year. Surprisingly, this rule still holds true. More powerful machines made it feasible to implement and sell RDBMS; cheap memory and powerful processors made them fast; and perpetually growing appetites for information made RDBMS products a commodity, drastically cutting their price down. Today, according to some estimates, less than 8 percent of the market is being held by the database legacy "dinosaurs" — mostly because of significant investment made by their owners more than 25 years ago, and their market share is constantly decreasing.

For better or for worse, relational database systems have come to rule on planet Earth.

Other DBMS Models

At the end of the 1980s, the buzzword was *object-oriented programming* (OOP). For very similar reasons (memory requirements and processing power) as those preventing widespread adoption of RDBMS, object-oriented programming did not take off until well into the 1990s. OOP languages are based on the notion that a programming (or business) problem could be modeled in terms of objects.

While the code of the program remained practically the same, the way the code was organized changed dramatically. It also changed the way programs were constructed, coded, and executed. For programming applications that communicate with the databases, it would be only natural to store objects on an as-is basis instead of disassembling them into text or into their component fields and putting them back together when needed.

A modern RDBMS has the ability to store binary objects (for example, pictures, sounds, and so on). In the case of object-oriented (OO) databases, they need to store conceptual objects: customer, order, and so on. By the end of the 1990s, everything indicated that object-oriented databases were going to be "the next big thing." Surprisingly, since then, the object-oriented database hype has been gradually diminishing. Even though there are several products on the market for pure OODBS (object-oriented database systems), none of these has met with a widespread adoption, and it looks like the emphasis has gradually moved toward OORDBMS (object-oriented relational database systems) that combine object-oriented features within traditional RDBMS attributes.

NOTE **Object-oriented features became part of SQL standards about 1999. All "big three" RDBMS vendors are at least partially compliant with the object-relational standards.**

The other development worth noticing is a wide adoption of eXtensible Markup Language (XML). XML was developed as a logical simplification of SGML (Standard Generalized Markup Language), partly because plain static HTML (HyperText Markup Language) was focused almost entirely on presentation, and a markup language that could accommodate structural and semantic markup was needed. An XML document contains self-describing data in a platform-independent, industry-standard format that makes it easy to transform into different types of documents, to search, or to transfer across heterogeneous networks.

CROSS-REF **XML is discussed in Chapter 15, "XML and SQL."**

XML first became a part of SQL standards in 2003. SQL:2003 contains a separate document (Part 14: "XML-Related Specifications") describing XML standards for SQL. The document was superseded by a major revision in 2006 (mostly for XQuery support), and another revision is pending and is expected to be released sometime in 2008.

NOTE **Every major RDBMS release has either a new version of its product or an add-in to the existing one to handle XML, to comply with SQL:2003 standards to a certain degree.**

While it is impossible to predict what model will emerge as a winner in the future, it seems reasonable to assume that relational databases are here for the long haul and have not yet reached their full potential. SQL as *the* language of the RDBMS will keep its importance in the database world.

Brief History of SQL and SQL Standards

As discussed earlier, pre-relational databases did not have a standard set of commands to work with data. Every database either had its own proprietary language or used programs written in COBOL, C, and so on to manipulate records. Also, the databases were relatively inflexible and did not allow any internal structural changes without bringing the databases offline and rewriting tons of code. That worked more or less effectively until the end of the 1960s, during which time most computer applications were based strictly on batch processing (running from beginning to end without user interaction).

Humble beginnings: RDBMS and SQL evolution

In the early 1970s, the growth of online applications (programs that require user interaction) triggered the demand for something more flexible. The situations in which an extra field was required for a particular record or a number of subfields exceeded the maximum number in the file layout became more and more common.

For example, imagine that the `CUSTOMER` record set has two fixed-length fields, `ADDRESS1` (for billing address) and `ADDRESS2` (for shipping address), and it works for all customers for some period of time. What if a customer, who owns a hardware store, bought another store? Now this record must have more than one shipping address. What if you have a new customer, `WILE ELECTRONICS INC.`, which owns 10 stores? Now, you have two choices. In the first case, you can take the whole dataset offline, modify the layout, and change and recompile all of the programs that work with it. This would satisfy the needs of the customer with 10 stores, but all customers with just one store each would have nine unnecessary fields in their records (see Figure 1-4). Furthermore, you cannot guarantee that, tomorrow, some other customer is not going to buy, say, 25 stores, and then you'll have to start over again. The second choice is to

add 10 identical records for Wile Electronics Inc., completely redundant except for the shipping address (see Figure 1-5). The programs would still have to be changed because otherwise they may return incorrect results.

FIGURE 1-4

Multiple columns to resolve multiple addresses for CUSTOMER

NAME	BILLADDR	SHIPADDR_1	SHIPADDR_2	SHIPADDR_N	SHIPADDR_10
MAGNETICS USA INC.	123 LAVACA ST.	444 PINE ST.			
WILE ELECTRONICS INC.	411 LONDON AVE.	232 EEL ST.	454 OAK ST.	...	999 ELK AVE.

FIGURE 1-5

Multiple records to resolve multiple addresses for CUSTOMER

NAME	BILLADDR	SHIPADDR
MAGNETICS USA INC.	123 LAVACA ST.	444 PINE ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	232 EEL ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	454 OAK ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	456 WILLOW ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	678 MAPLE AVE.
WILE ELECTRONICS INC.	411 LONDON AVE.	332 WALNUT ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	531 DEER ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	865 CEDAR AVE.
WILE ELECTRONICS INC.	411 LONDON AVE.	911 MYRTLE ST.
WILE ELECTRONICS INC.	411 LONDON AVE.	777 SITKA AVE.
WILE ELECTRONICS INC.	411 LONDON AVE.	999 ELK AVE.

As you can see, most problems are actually rooted in the structure of the database, which usually consisted of just one file with records of a fixed length. The solution is to spread data across several files and reassemble the required data when needed. As discussed earlier in this chapter, hierarchical and network database models attempted to move in this direction, but they still had had too many shortcomings, so the relational model became the most popular technique. The problem just discussed would not be a problem at all in a relational database, where `CUSTOMER` and `ADDRESS` are separate entities (tables), linked via the primary/foreign key relationship (see Figure 1-6). All you have to do is to add as many `ADDRESS` records as you want with a foreign key that refers to its parent (see Figure 1-7).

NOTE **This example might not look very convincing — it might appear that instead of adding new shipping addresses to `CUSTOMER`, were added the same records to a separate `ADDRESS` table. In fact, the difference is huge. In a real-life legacy database, the `CUSTOMER` file would not have just `NAME` and `ADDRESS` fields, but rather, it would contain tons of other information: about orders, products, invoices, shipments, and so on. All that would be repeated every time you accessed the database, even for something as simple as adding a new shipping address.**

FIGURE 1-6

Primary/foreign key relationship between tables

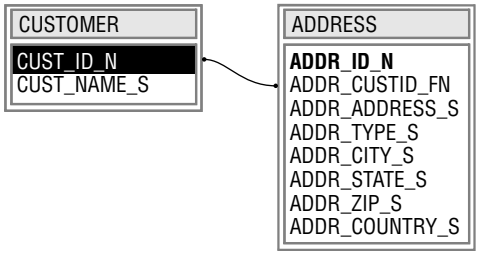


FIGURE 1-7

Resolving the problem of multiple customer addresses within a relational model

CUST_ID_N	CUST_NAME_S
7	WILE ELECTRONICS INC.

ADDR_CUSTID_FN	ADDR_ADDRESS_S	ADDR_TYPE_S
7	411 S LONDON AVE.	BILLING
7	454 OAK ST.	SHIPPING
7	678 MAPLE AVE.	SHIPPING
7	999 ELK AVE.	SHIPPING
7	777 SITKA AVE.	SHIPPING
7	911 MYRTLE ST.	SHIPPING
7	865 CEDAR AVE.	SHIPPING
7	531 DEER ST.	SHIPPING
7	332 WALNUT ST.	SHIPPING
7	456 WILLOW ST.	SHIPPING
7	232 EEL ST.	SHIPPING

Another advantage of the relational schema is simplified logic for the applications that work with data. For example, assume the nonrelational `CUSTOMER` dataset has fields for a maximum of five customer orders. (That easily could be a couple of fields per order, by the way.) If you want to display all orders for a specific customer, a program will have to scroll through all these fields, determine which ones are not empty, and display their contents. In the relational case, all you need to do is display all records in the `ORDER_HEADER` table that have the required customer number. This relational schema made ad hoc query languages relatively easy to write, which eventually led to the appearance of SQL.

The concept of a relational database, and thus SQL, was first introduced in 1970 by IBM researcher Dr. Edward Frank Codd in a paper titled "A Relational Model of Data for Large Shared Data Banks." In simple words, his idea of a relational database model was based on data independence from hardware and storage implementation and a nonprocedural high-level

computer language to access the data. The problem was that IBM already had declared its own product, called IMS, as its sole strategic database product — the company management was not convinced at all that developing new commercial software based on a relational schema was worth the money and the effort. A new database product could also potentially hurt the sales of IMS.

In spite of all that, a relational database prototype called System R was finally introduced by IBM in the late 1970s, but it never became a commercial product and was more of a scientific interest, unlike its database language, SQL (first known as SEQUEL), which eventually became the standard for all relational databases (after years of evolution). Another relational product, called Ingres, was developed by scientists in a government-funded program at the University of California, Berkeley, at about the same time, and had its own nonprocedural language, QUEL, similar in some respects to IBM's SQL.

The first commercial relational database was neither System R nor Ingres. Oracle Corporation released its first product in 1979, followed by IBM's SQL/DS (1980/81) and DB2 (1982/83). The commercial version of Ingres also became available in the early 1980s. Sybase, Inc. released the first version of its product in 1986, and in 1988, Microsoft introduced SQL Server. Many other products by other companies were also released since then, but their share of today's market is minimal.

A brief history of SQL standards

The relational database model was slowly but surely becoming the industry standard in the late 1980s. The problem was that even though SQL became a commonly recognized database language, the differences among major vendors' implementations were growing, and some kind of standard became necessary.

Around 1978, the Committee on Data Systems and Language (CODASYL) commissioned the development of a network data model as a prototype for any future database implementations. This continued work started in the early 1970s with the Data Definition Language Committee (DDLC). By 1982, these efforts culminated in the data definition language (DDL) and data manipulation language (DML) standards proposal. They became standards four years later — endorsed by an organization then known as American National Standards Institute's Technical Committee X3H2 (Database).

NOTE **The name has changed several times. In the advent of the Internet, the characters "X3" (which stood for nothing at all; it was only an identifier) became part of many people's effort to locate pornographic ("XXX") films, so "X3" has been changed to NCITS (National Committee for Information Technology Standards). Just to be sure that nobody got it all figured out, they changed names again just a couple of years later to INCITS, replacing "National" with "International."**

SQL-86/87 (SQL1)

X3H2 was given a mandate to standardize the relational data model in 1982. The project initially was based on IBM SQL/DS specifications, and for some time followed closely IBM DB2

developments. In 1984, the standard was redesigned to be more generic, to allow for more diversity among database products vendors. After passing through all the bureaucratic loops, it was endorsed as an American National Standards in 1986. The International Organization for Standardization ISO adopted the standard in 1987. A revised standard, commonly known as SQL-89, was published two years later.

SQL-89 (SQL1.1)

SQL-89 (or SQL1) is a rather minimalist standard that was established by encircling all RDBMS in existence in 1989. It comprised only a limited number of features related strictly to locating and retrieving information in a relational database. The major commercial vendors could not (and still to a certain degree cannot) agree upon implementation details, so much of the SQL-89 standard is intentionally left incomplete, and numerous features are marked as implementer-defined.

SQL-92 (SQL2)

Because of the aforementioned limitations, the previous standard was revised, and in 1992, the first solid SQL standard, SQL-92 or SQL2, was published. ANSI took SQL-89 as a basis, but corrected several weaknesses in it, filled many gaps in the old standard, and presented conceptual SQL features, which at that time exceeded the capabilities of any existing RDBMS implementation. Also, the SQL-92 standard is over five times longer than its predecessor (about 500 pages more), and has three levels of conformance.

Entry-level conformance is basically improved SQL-89. The differences were insignificant — for example, the specification of the enforcement of `WITH CHECK OPTION` for a view has been clarified.

Intermediate-level conformance was a set of major improvements, including, but not limited to, user naming of constraints; support for varying-length characters and national character sets, case and cast expressions, built-in join operators, and dynamic SQL; and the ability to alter tables, set transactions, use subqueries in updatable views, and use set operators (`UNION`, `EXCEPT`, `INTERSECT`) to combine multiple queries' results.

Full-level conformance included some truly advanced features, including deferrable constraints, assertions, temporary local tables, and privileges on character sets and domains.

Conformance testing was performed by the U.S. Government Department of Commerce's National Institute of Standards and Technology (NIST). The vendors hurried to comply because a public law passed in the beginning of the 1990s required an RDBMS product to pass the tests in order to be considered by a federal agency.

NOTE **In 1996, NIST dismantled the conformance testing program (citing "high costs" as the reason behind the decision). Since then, the only verification of SQL standards compliance has come from the RDBMS vendors themselves, which understandably increased the number of vendor-specific features as well as nonstandard implementation of the standard ones.**

SQL:1999 (SQL3)

SQL:1999 represented the next step in SQL standards development. The efforts to define this standard began over a year before its predecessor — SQL-92 (SQL2) — was adopted. The new standard was developed under the guidance of both the ANSI and ISO committees, and the change introduced into the database world by SQL3 a dramatic shift from a nonrelational to a relational database model. Its sheer complexity is reflected in the number of pages describing the standard — 2,000 pages compared to 120 or so pages for SQL-89 and 628 pages for SQL-92. Some of the defined standards (for example, stored procedures) existed as vendor-specific extensions; some of them (like OOP) are completely new to SQL proper. SQL3 was released as an ANSI/ISO draft standard in 1999; later the same year, its status was changed to a standard level.

SQL3 extends traditional relational data models to incorporate objects and complex data types within the relational tables, along with all supporting mechanisms. It brings into SQL all the major OOP principles, namely *inheritance*, *encapsulation*, and *polymorphism* (all of which are beyond the scope of this book) in addition to "standard" SQL features defined in SQL-92. It provides seamless integration with the data consumer applications designed and implemented in OO languages (SmallTalk, Eiffel, and so on).

NOTE **SQL standards are becoming more and more dynamic. For example, between 1992 and 1999, the years of major SQL standards releases, two parts were added to SQL standards, SQL/CLI and SQL/PSM, in 1995 and 1996, correspondingly.**

SQL:2003

The big topic in SQL:2003 is XML. In addition, there are some minor modifications and "bug fixes" to the SQL:1999 standard, as well as a couple of new features, such as table functions, sequence generators, auto-generated values, identity-columns, and some modifications to DML (`MERGE`)and DDL (`CREATE TABLE LIKE` and `CREATE TABLE AS`) syntax. Also, a couple of obsolete data types have been removed (`BIT` and `BIT VARYING`) and a few new data types have been added (`BIGINT`, `MULTISET`, and `XML`).

SQL:2003 consists of nine parts: SQL/Framework (information common to all parts of the standard), SQL/Foundation (data definition and data manipulation syntax and semantics), SQL/CLI (Call Level Interface / ODBC), SQL/PSM (Persistent Stored Modules — procedural language extensions), SQL/MED (Management of External Data — external OS files access), SQL/OLB (Object Language Bindings — syntax of embedding SQL in Java), SQL/Schemata (Metadata), SQL/JRT (Java Routines and Types), and SQL/XML (XML-related specifications in SQL). It's 3,606 pages, which is almost twice as long as the previous SQL standard.

SQL:2008

Yet another revision of SQL standards is expected to be released in 2008. The XML section, which was already significantly changed and enlarged in 2006, is being revised again. Also, some other minor changes to various sections are pending. The new SQL:2008 features will likely include a new `BINARY` data type, regular expression support, and, possibly, materialized views and `FIRST n` / `TOP n` queries.

NOTE **As of this writing, all major database vendors (Oracle, DB2, and Microsoft SQL Server) have at least partial compliance with SQL:2003, but there is no RDBMS vendor on the market who meets the new standards in full. Each of the vendors has individual features that venture into higher levels of conformance.**

NOTE **SQL ANSI/ISO standards are good to know, and they may help you learn the actual SQL implementations, but it is impossible (or almost impossible) to write a real 100-percent ANSI SQL–compliant production script. You can compare this with knowing Latin, which can help you learn Spanish, Italian, or Portuguese, but people would hardly understand you if you started speaking Latin on the streets of Madrid, Rome, or Lisbon. The main difference is the general direction — Latin is an ancestor of all the aforementioned (and many more) languages, whereas ANSI/ISO SQL standards are a goal for all proprietary SQL flavors.**

Summary

Databases penetrate virtually every branch of human activity, with the relational database management system (RDBMS) becoming the de facto standard. Some legacy database models — hierarchical and network databases — are still in use, but the relational database model holds the lion's share of the market.

The RDBMS resolves some of the inherent problems of the legacy databases, and — with the advent of faster hardware and support from the industry heavyweights — became the staple of every business enterprise. The new object-oriented database systems (OODMS) are evolving, although none has reached the level of acceptance comparable with that of RDBMS. The object-oriented features became a part of SQL standards, and all major vendors are implementing them to certain degree, gradually becoming object-oriented relational database systems (OORDBMS).

XML became a part of SQL standard in 2003, and is becoming more and more popular. Oracle, Microsoft, and IBM, as well as many other smaller vendors, are adding XML features to their RDBMS products.

Most of the existing applications on the database market use SQL as the standard language. There have been four generations of the standard so far: SQL-86/87 and SQL-89, SQL-92, SQL:1999, and SQL:2003 (SQL:2008 is coming) with virtually every RDBMS product being at least partially SQL:2003-compliant.

SQL as the language of the RDBMS has been on the market for almost 30 years, and everything indicates it will be around in the foreseeable future. SQL proved to be user-friendly and flexible enough to accommodate new features, and is constantly evolving, as are the SQL standards.