IN THIS CHAPTER

- Working with data files and databases
- Seeing how databases, queries, and database applications fit together
- Looking at different database models
- Charting the rise of relational databases

Chapter 1
Understanding Relational Databases

SQL (pronounced *ess cue el*, but you'll hear some people say *see quel)* is the international standard language used in conjunction with relational databases — and it just so happens that relational databases are the dominant form of data storage throughout the world. In order to understand *why* relational databases are the primary repositories for the data of both small and large organizations, you must first understand the various ways in which computer data can be stored and how those storage methods relate to the relational database model. To help you gain that understanding, I spend a good portion of this chapter going back to the earliest days of electronic computers and recapping the history of data storage.

I realize that grand historical overviews aren't everybody's cup of tea, but I'd argue that it's important to see that the different data storage strategies that have been used over the years each have their own strengths and weaknesses. Ultimately, the strengths of the relational model overshadowed its weaknesses and it became the most frequently used method of data storage. Shortly after that, SQL became the most frequently used method of dealing with data stored in a relational database.

Understanding Why Today's Databases Are Better than Early Databases

In the early days of computers, the concept of a database was more theoretical than practical. Vannevar Bush, the twentieth-century visionary, conceived of the idea of a database in 1945, even before the first electronic computer was built. However, practical implementations of databases — such as IBM's IMS (*I*nformation *M*anagement *S*ystem), which kept track of all the parts on the Apollo moon mission and its commercial followers — did not appear for a number of years after that. For far too long, computer data was still being kept in files rather than migrated to databases.

Irreducible complexity

Any software system that performs a useful function is complex. The more valuable the function, the more complex its implementation. Regardless of how the data is stored, the complexity remains. The only question is where that complexity resides.

Any nontrivial computer application has two major components: the program and the data. Although an application's level of complexity depends on the task to be performed, developers have some control over the location of that complexity. The complexity may reside primarily in the program part of the overall system, or it may reside in the data part. In the sections that follow, I tell you how the location of complexity in databases shifted over the years as technological improvements made that possible.

Managing data with complicated programs

In the earliest applications of computers to solve problems, all of the complexity resided in the program. The data consisted of one data record of fixed length after another, stored sequentially in a file. This is called a *flat file* data structure. The data file contains nothing but data. The program file must include information about where particular records are within the data file (one form of *metadata*, whose sole purpose is to organize the primary data you *really* care about). Thus, for this type of organization, the complexity of managing the data is entirely in the program.

Here's an example of data organized in a flat file structure:

```
Harold Percival26262 S. Howards Mill Rd.Westminster CA92683
Jerry Appel    32323 S. River Lane Road Santa Ana   CA92705
Adrian Hansen  232   Glenwood Court     Anaheim     CA92640
John Baker     2222  Lafayette Street   Garden GroveCA92643
Michael Pens   77730 S. New Era Road    Irvine      CA92715
Bob Michimoto  25252 S. Kelmsley Drive  Stanton     CA92610
Linda Smith    444   S.E. Seventh StreetCosta Mesa  CA92635
Robert Funnell 2424  Sheri Court        Anaheim     CA92640
Bill Checkal   9595  Curry Drive        Stanton     CA92610
Jed Style      3535  Randall Street     Santa Ana   CA92705
```

This example includes fields for name, address, city, state, and zip code. Each field has a specific length, and data entries must be truncated to fit into that length. If entries don't use all the space allotted to them, storage space is wasted.

The flat file method of storing data has several consequences, some beneficial and some not. First, the beneficial consequences:

- **Storage requirements are minimized.** Because the data files contain nothing but data, they take up a minimum amount of space on hard disks or other storage media. The code that must be added to any one program that contains the metadata is small compared to the overhead involved with adding a database management system (DBMS) to the data side of the system. (A *database management system* is the program that controls access to — and operations on — a database.)
- **Operations on the data can be fast.** Because the program interacts directly with the data, with no DBMS in the middle, well-designed applications can run as fast as the hardware permits.

Wow! What could be better? A data organization that minimizes storage requirements and at the same time maximizes speed of operation seems like the best of all possible worlds. But wait a minute . . .

Flat file systems came into use in the 1940s. We have known about them for a long time, and yet today they are almost entirely replaced by database systems. What's up with that? Perhaps it is the not-so-beneficial consequences:

- **Updating the data's structure can be a huge task.** It is common for an organization's data to be operated on by multiple application programs, with multiple purposes. If the metadata about the structure of data is in the program rather than attached to the data itself, *all* the programs that access that data must be modified whenever the data structure is changed. Not only does this cause a lot of redundant work (because the same changes must be

made in all the programs), but it is an invitation to problems. All the programs must be modified in exactly the same way. If one program is inadvertently forgotten, the program will fail the next time you run it. Even if all the programs *are* modified, any that aren't modified exactly as they should be will fail, or even worse, corrupt the data without giving any indication that something is wrong.

- **Flat file systems provide no protection of the data.** Anyone who can access a data file can read it, change it, or delete it. A flat file system doesn't have a database management system, which restricts access to authorized users.
- **Speed can be compromised.** Accessing records in a large flat file can actually be slower than a similar access in a database because flat file systems do not support indexing. Indexing is a major topic that I discuss in Book 2, Chapter 3.
- **Portability becomes an issue.** If the specifics that handle how you retrieve a particular piece of data from a particular disk drive is coded into each program, what happens when your hardware becomes obsolete and you must migrate to a new system? All your applications will have to be changed to reflect the new way of accessing the data. This task is so onerous that many organizations have chosen to limp by on old, poorly performing systems instead of enduring the pain of transitioning to a system that would meet their needs much more effectively. Organizations with legacy systems consisting of millions of lines of code are pretty much trapped.

In the early days of electronic computers, storage was relatively expensive, so system designers were highly motivated to accomplish their tasks using as little storage space as possible. Also, in those early days, computers were much slower than they are today, so doing things the fastest possible way also had a high priority. Both of these considerations made flat file systems the architecture of choice, despite the problems inherent in updating the structure of a system's data.

The situation today is radically different. The cost of storage has plummeted and continues to drop on an exponential curve. The speed at which computations are performed has increased exponentially also. As a result, minimizing storage requirements and maximizing the speed with which an operation can be performed are no longer the primary driving forces that they once were. Because systems have continually become bigger and more complex, the problem of maintaining them has likewise grown. For all these reasons, flat file systems have lost their attractiveness, and databases have replaced them in practically all application areas.

Managing data with simple programs

The major selling point of database systems is that the metadata resides on the data end of the system rather than in the program. The program doesn't have to know anything about the details of how the data is stored. The program makes *logical* requests for data, and the DBMS translates those logical requests into commands that go out to the physical storage hardware to perform whatever operation has been requested. (In this context, a *logical request* asks for a specific piece of information, but does not specify its location on hard disk in terms of platter, track, sector, and byte.) Here are the advantages of this organization:

- Because application programs need to know only what data they want to operate on, and not where that data is located, they are unaffected when the physical details of where data is stored changes.
- Portability across platforms, even when they are highly dissimilar, is easy as long as the DBMS used by the first platform is also available on the second. Generally, you don't need to change the programs at all to accommodate various platforms.

What about the disadvantages? They include the following:

- Placing a database management system in between the application program and the data slows down operations on that data. This is not nearly the problem that it used to be. Modern advances, such as the use of high speed cache memories have eased this problem considerably.
- Databases take up more space on disk storage than the same amount of data would take up in a flat file system. This is due to the fact that metadata is stored along with the data. The metadata contains information about how the data is stored so that the application programs don't have to include it.

Which type of organization is better?

I bet you think you already know how I'm going to answer this question. You're probably right, but the answer is not quite so simple. There is no one correct answer that applies to all situations. In the early days of electronic computing, flat file systems were the only viable option. To perform any reasonable computation in a timely and economical manner, you had to use whatever approach was the fastest and required the least amount of storage space. As more and more application software was developed for these systems, the organizations that owned them became locked in tighter and tighter to what they had. To change to a more modern database system requires rewriting all their applications from scratch and

reorganizing all their data, a monumental task. As a result, we still have legacy flat file systems that continue to exist because switching to more modern technology isn't feasible, both economically and in terms of the time it would take to make the transition.

Databases, Queries, and Database Applications

What are the chances that a person could actually find a needle in a haystack? Not very good. Finding the proverbial needle is so hard because the haystack is a random pile of hay with individual pieces of hay going in every direction, and the needle is located at some random place among all that hay.

A flat file system is not really very much like a haystack, but it does lack structure — and in order to find a particular record in such a file, you must use tools that lie outside of the file itself. This is like applying a powerful magnet to the haystack to find the needle.

Making data useful

For a collection of data to be useful, you must be able to easily and quickly retrieve the particular data you want, without having to wade through all the rest of the data. One way to make this happen is to store the data in a logical structure. Flat files don't have much structure, but databases do. Historically, the hierarchical database model and the network database model were developed before the relational model. Each one organizes data in a different way, but all three produce a highly structured result. Because of that, starting in the 1970s, any new development projects were most likely done using one of the aforementioned three database models: hierarchical, network, or relational. (I explore each of these database models further in the "Examining Competing Database Models" section, later in this chapter.)

Retrieving the data you want — and only the data you want

Of all the operations that people perform on a collection of data, the retrieval of specific elements out of the collection is the most important. This is because retrievals are performed more often than any other operation. Data entry is done

only once. Changes to existing data are made relatively infrequently, and data is deleted only once. Retrievals, on the other hand, are performed frequently, and the same data elements may be retrieved many times. Thus, if you could optimize only one operation performed on a collection of data, that one operation should be data retrieval. As a result, modern database management systems put a great deal of effort into making retrievals fast.

Retrievals are performed by queries. A modern database management system analyzes a query that is presented to it and decides how best to perform it. Generally, there are multiple ways of performing a query, some much faster than others. A good DBMS consistently chooses a near-optimal execution plan. Of course, it helps if the query is formulated in an optimal manner to begin with. (I discuss optimization strategies in depth in Book 7, which covers database tuning.)

THE FIRST DATABASE SYSTEM

The first true database system was developed by IBM in the 1960s in support of NASA's Apollo moon landing program. The number of components in the Saturn V launch vehicle, the Apollo Command and Service Module, and the lunar lander far exceeded anything that had been built up to that time. Every component had to be tested more exhaustively than anything had ever been tested before because each component would have to withstand the rigors of an environment that was more hostile and more unforgiving than any environment that humans had ever attempted to work in. Flat file systems were out of the question. IBM's solution, which IBM later transformed into a commercial database product named IMS (Information Management System), kept track of each individual component, as well as its complete history.

When the ill-fated Apollo 13's main oxygen tank ruptured on the way to the Moon, engineers worked frantically to come up with a plan to save the lives of the three astronauts headed for the Moon. The engineers succeeded and transmitted a plan to the astronauts that worked.

After the crew had returned safely to Earth, querying IMS records about the oxygen tank that failed showed that somewhere between the oxygen tank's manufacture and its installation in Apollo 13, it had been dropped on the floor. Engineers retested it for its ability to withstand the pressure it would have to contain during the mission, and then put it back in stock after it passed the test. But it turns out that in this case, the test did not detect the hidden damage to the tank, and NASA should not have used the oxygen tank on the Apollo 13 mission. The history stored in IMS showed that passing a pressure test is not enough to assure that a dropped tank is undamaged. No dropped tanks were ever used on subsequent Apollo missions.

Examining Competing Database Models

A *database model* is simply a way of organizing data elements within a database. In this section, I give you the details on the three database models that appeared first on the scene:

- **Hierarchical:** Organizes data into levels, where each level contains a single category of data, and parent/child relationships are established between levels
- **Network:** Organizes data in a way that avoids much of the redundancy inherent in the hierarchical model
- **Relational:** Organizes data into a structured collection of two-dimensional tables

After the introductions of the hierarchical, network, and relational models, computer scientists have continued to develop databases models that have been found useful in some categories of applications. I briefly mention some of these later in this chapter, along with their areas of applicability. However, the hierarchical, network, and relational models are the ones that have been primarily used for general business applications.

Looking at the historical background of the competing models

The first functioning database system was developed by IBM and went live at an Apollo contractor's site on August 14, 1968. (Read the whole story in "The first database system" sidebar, here in this chapter.) Known as IMS (Information Management System), it is still (amazingly enough) in use today, over 50 years later, because IBM has continually upgraded it in support of its customers.

TIP

If you are in the market for a database management system, you may want to consider buying it from a vendor that will be around, and that is committed to supporting it for as long as you will want to use it. IBM has shown itself to be such a vendor, and of course, there are others as well.

IMS is an example of a hierarchical database product. About a year after IMS was first run, the network database model was described by an industry committee. About a year after that, Dr. Edgar F. "Ted" Codd, also of IBM, proposed the relational model. Within a short span of years, the three models that were to dominate the database market for decades were spawned.

Quite a few years went by before the object-oriented database model made its appearance, presenting itself as an alternative meant to address some of the deficiencies of the relational model. The *object-oriented database model* accommodates the storage of types of data that don't easily fit into the categories handled by relational databases. Although they have advantages in some applications, object-oriented databases have not captured significant market share. The *object-relational model* is a merger of the relational and object models, and it is designed to capture the strengths of both, while leaving behind their major weaknesses. Now, there is something called the NoSQL model. It is designed to work with data that is not rigidly structured. Because it does not use SQL, I will not discuss it in this book.

The hierarchical database model

The *hierarchical database model* organizes data into levels, where each level contains a single category of data, and parent/child relationships are established between levels. Each parent item can have multiple children, but each child item can have one and only one parent. Mathematicians call this a *tree-structured* organization, because the relationships are organized like a tree with a trunk that branches out into limbs that branch out into smaller limbs. Thus all relationships in a hierarchical database are either one-to-one or one-to-many. Many-to-many relationships are not used. (More on these kinds of relationships in a bit.)

A list of all the stuff that goes into building a finished product— a listing known as a *bill of materials*, or BOM — is well suited for a hierarchical database. For example, an entire machine is composed of assemblies, which are each composed of subassemblies, and so on, down to individual components. As an example of such an application, consider the mighty Saturn V Moon rocket that sent American astronauts to the Moon in the late 1960s and early 1970s. Figure 1-1 shows a hierarchical diagram of major components of the Saturn V.

Three relationships can occur between objects in a database:

- **One-to-one relationship:** One object of the first type is related to one and only one object of the second type. In Figure 1-1, there are several examples of one-to-one relationships. One is the relationship between the S-2 stage LOX tank and the aft LOX bulkhead. Each LOX tank has one and only one aft LOX bulkhead, and each aft LOX bulkhead belongs to one and only one LOX tank.
- **One-to-many relationship:** One object of the first type is related to multiple objects of the second type. In the Saturn V's S-1C stage, the thrust structure contains five F-1 engines, but each engine belongs to one and only one thrust structure.

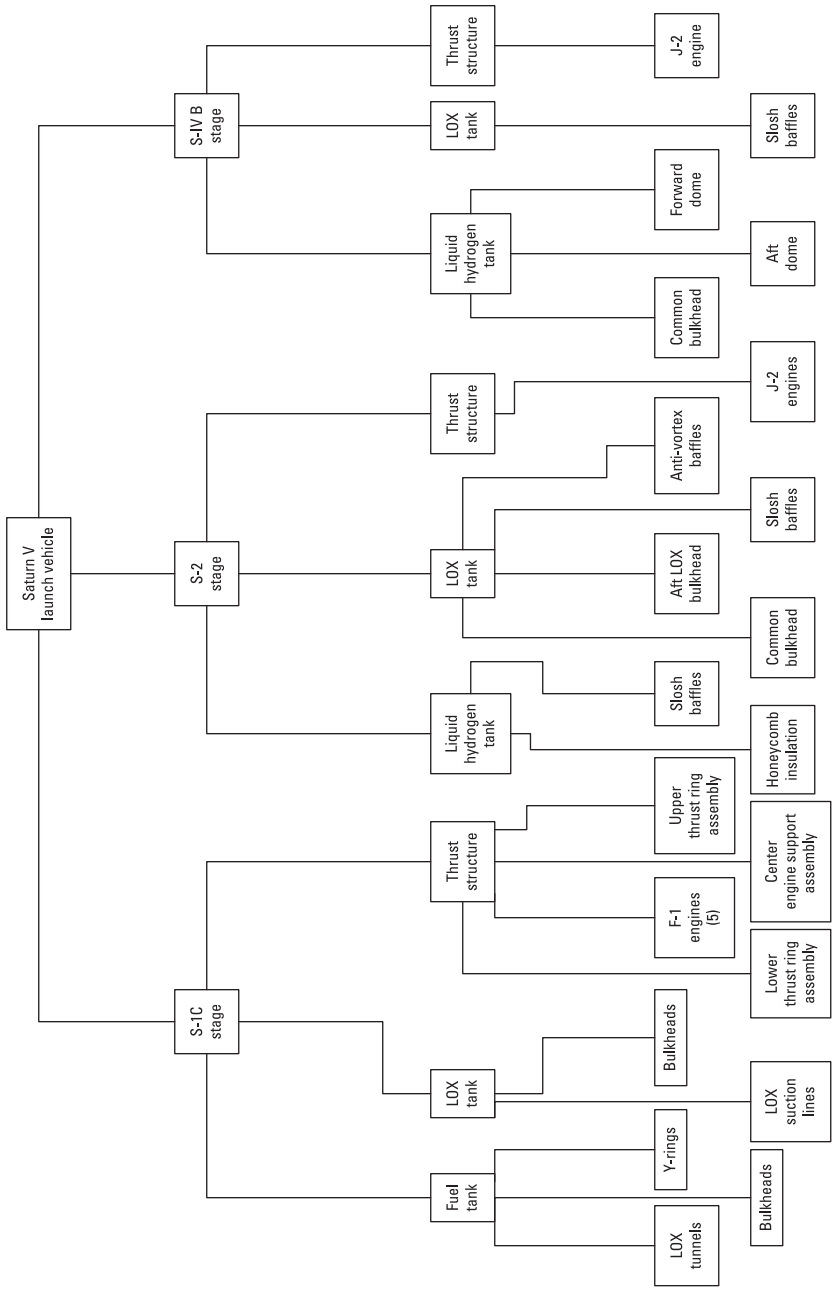

FIGURE 1-1: A hierarchical model of the Saturn V moon rocket.

» **Many-to-many relationship:** Multiple objects of the first type are related to multiple objects of the second type. This kind of relationship is not handled cleanly by a hierarchical database. Attempts to do so tend to be kludgy. One example might be two-inch hex-head bolts. These bolts are not considered to be uniquely identifiable, and any one such bolt is interchangeable with any other. An assembly might use multiple bolts, and a bolt could be used in any of several different assemblies.

A great strength of the hierarchical model is its high performance. Because relationships between entities are simple and direct, retrievals from a hierarchical database that are set up to take advantage of the way the data is structured can be very fast. However, retrievals that don't take advantage of the way the data is structured are slow and sometimes can't be made at all. It's difficult to change the structure of a hierarchical database to address new requirements. This structural rigidity is the greatest weakness of the hierarchical model. Another problem with the hierarchical model is the fact that, structurally, it requires a lot of redundancy, as my next example makes clear.

First off, time to state the obvious: Not many organizations today are designing rockets capable of launching payloads to the moon. The hierarchical model can also be applied to more common tasks, however, such as tracking sales transactions for a retail business. As an example, I use some sales transaction data from Gentoo Joyce's fictitious online store of penguin collectibles. She accepts PayPal, MasterCard, Visa, and money orders and sells various items featuring depictions of penguins of specific types — gentoo, chinstrap, and adelie.

As shown in Figure 1-2, customers who have made multiple purchases show up in the database multiple times. For example, you can see that Lynne has purchased with PayPal, MasterCard, and Visa. Because this is hierarchical, Lynne's information shows up multiple times, and so does the information for every customer who has bought more than once. Product information shows up multiple times too.

REMEMBER

This organization is actually more complex than what is shown in Figure 1-2. Additional "trees" would hold the details about each customer and each product. This duplicate data is a waste of storage space because one copy of a customer's data is sufficient, and so is one copy of product information.

Perhaps even more damaging than the wasted space that results from redundant data is the possibility of data corruption. Whenever multiple copies of the same data exist in a database, there is the potential for modification anomalies. A *modification anomaly* is an inconsistency in the data after a modification is made. Suppose you want to delete a customer who is no longer buying from you. If multiple copies of that customer's data exist, you must find and delete all of them to maintain data integrity. On a slightly more positive note, suppose you just want to update a customer's address information. If multiple copies of the customer's data exist, you must find and modify all of them in exactly the same way to maintain data integrity. This can be a time-consuming and error-prone operation.

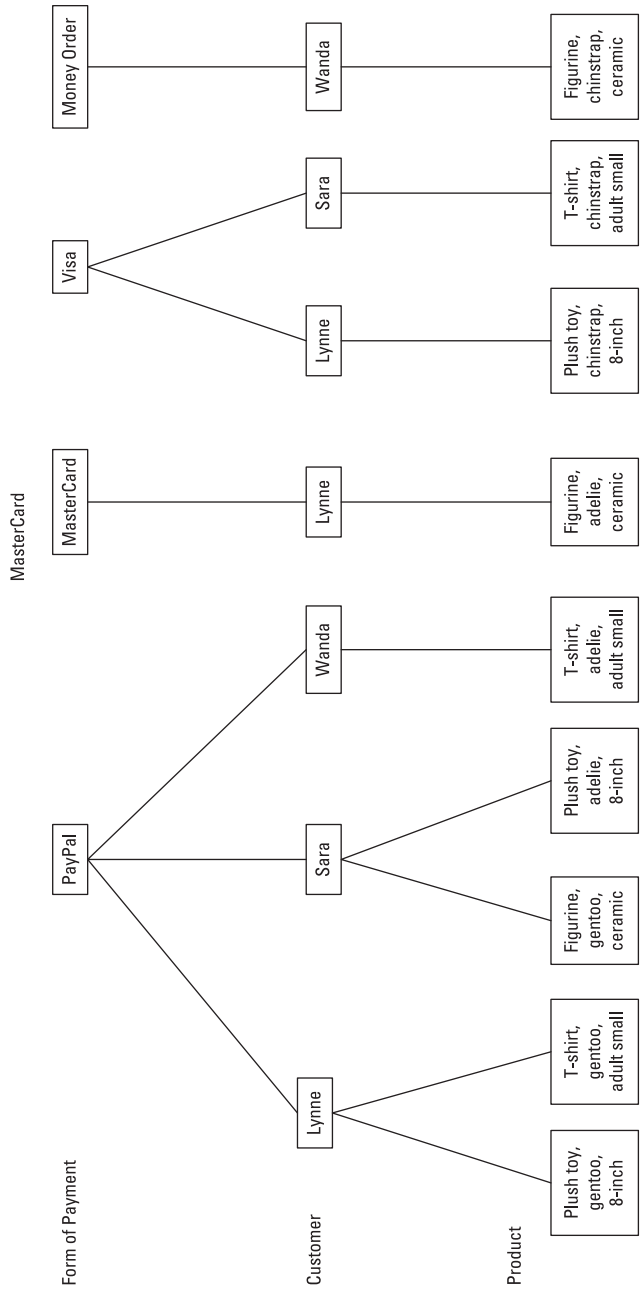

FIGURE 1-2: A hierarchical model of a sales database for a retail business.

The network database model

The network model — the one that followed close upon the heels of the hierarchical, appearing as it did in 1969 — is almost the exact opposite of the hierarchical model. Wanting to avoid the redundancy of the hierarchical model without sacrificing too much in the way of performance, the designers of the *network model* opted for an architecture that does not duplicate items, but instead increases the number of relationships associated with some items. Figure 1-3 shows this architecture for the same data that was shown in Figure 1-2.

As you can see in Figure 1-3, the network model does not have the tree structure with one-directional flow characteristic of the hierarchical model. Looked at this way, it shows very clearly that, for example, Lynne had bought multiple products, but also that she has paid in multiple ways. There is only one instance of Lynne in this model, compared to multiple instances in the hierarchical model. However, to balance out that advantage, there are seven relationships connected to that one instance of Lynne, whereas in the hierarchical model there are no more than three relationships connected to any one instance of Lynne.

REMEMBER

The network model eliminates redundancy, but at the expense of more complicated relationships. This model can be better than the hierarchical model for some kinds of data storage tasks, but worse for others. Neither one is consistently superior to the other.

The relational database model

In 1970, Edgar Codd of IBM published a paper introducing the *relational database* model. Initially, database experts gave it little consideration. It clearly had an advantage over the hierarchical model in that data redundancy was minimal; it had an advantage over the network model with its relatively simple relationships. However, it had what was perceived to be a fatal flaw. Due to the complexity of the relational database engine that it required, any implementation would be much slower than a comparable implementation of either the hierarchical or the network model. As a result, it was almost ten years before the first implementation of the relational database idea hit the market.

Moore's Law had finally made relational database technology feasible. (In 1965, Gordon Moore, one of the founders of Intel, noticed that the cost of computer memory chips was dropping by half about every two years. He predicted that this trend would continue. After over 50 years, the trend is still going strong, and Moore's prediction has been enshrined as an empirical law.)

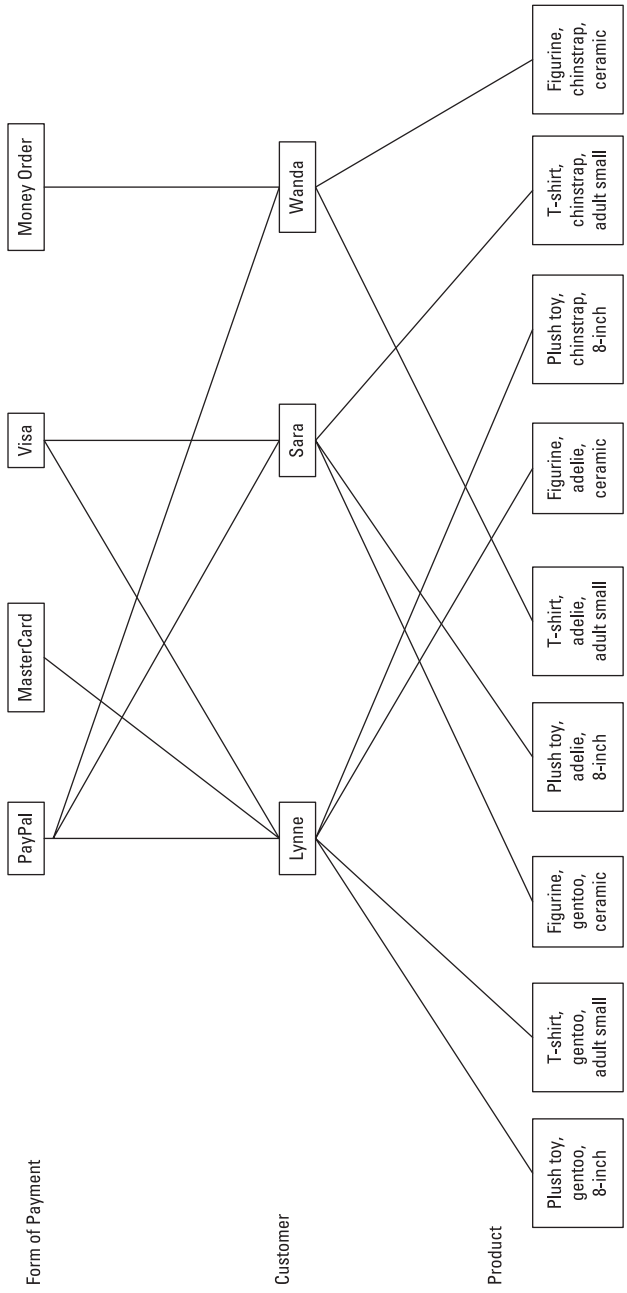

FIGURE 1-3: A network model of transactions at an online store.

IBM delivered a relational DBMS (RDBMS) integrated into the operating system of the System 38 computer server platform in 1978, and Relational Software, Inc., delivered the first version of Oracle — the granddaddy of all standalone relational database management systems — in 1979.

Defining what makes a database relational

The original definition of a relational database specified that it must consist of two-dimensional tables of rows and columns, where the cell at the intersection of a row and column contains an atomic value (where *atomic* means not divisible into subvalues). This definition is commonly stated by saying that a relational database table may not contain any *repeating groups*. The definition also specified that each row in a table be uniquely identifiable. Another way of saying this is that every table in a relational database must have a *primary key*, which uniquely identifies a row in a database table. Figure 1-4 shows the structure of an online store database, built according to the relational model.

The relational model introduced the idea of storing database elements in two-dimensional tables. In the example shown in Figure 1-4, the Customer table contains all the information about each customer; the Product table contains all the information about each product, and the Transaction table contains all the information about the purchase of a product by a customer. The idea of separating closely related things from more distantly related things by dividing things up into tables was one of the main factors distinguishing the relational model from the hierarchical and network models.

Protecting the definition of relational databases with Codd's rules

As the relational model gained in popularity, vendors of database products that were not really relational started to advertise their products as relational database management systems. To fight the dilution of his model, Codd formulated 12 rules that served as criteria for determining whether a database product was in fact relational. Codd's idea was that a database must satisfy all 12 criteria in order to be considered relational.

Codd's rules are so stringent, that even today, there is not a DBMS on the market that completely complies with all of them. However, they have provided a good goal toward which database vendors strive.

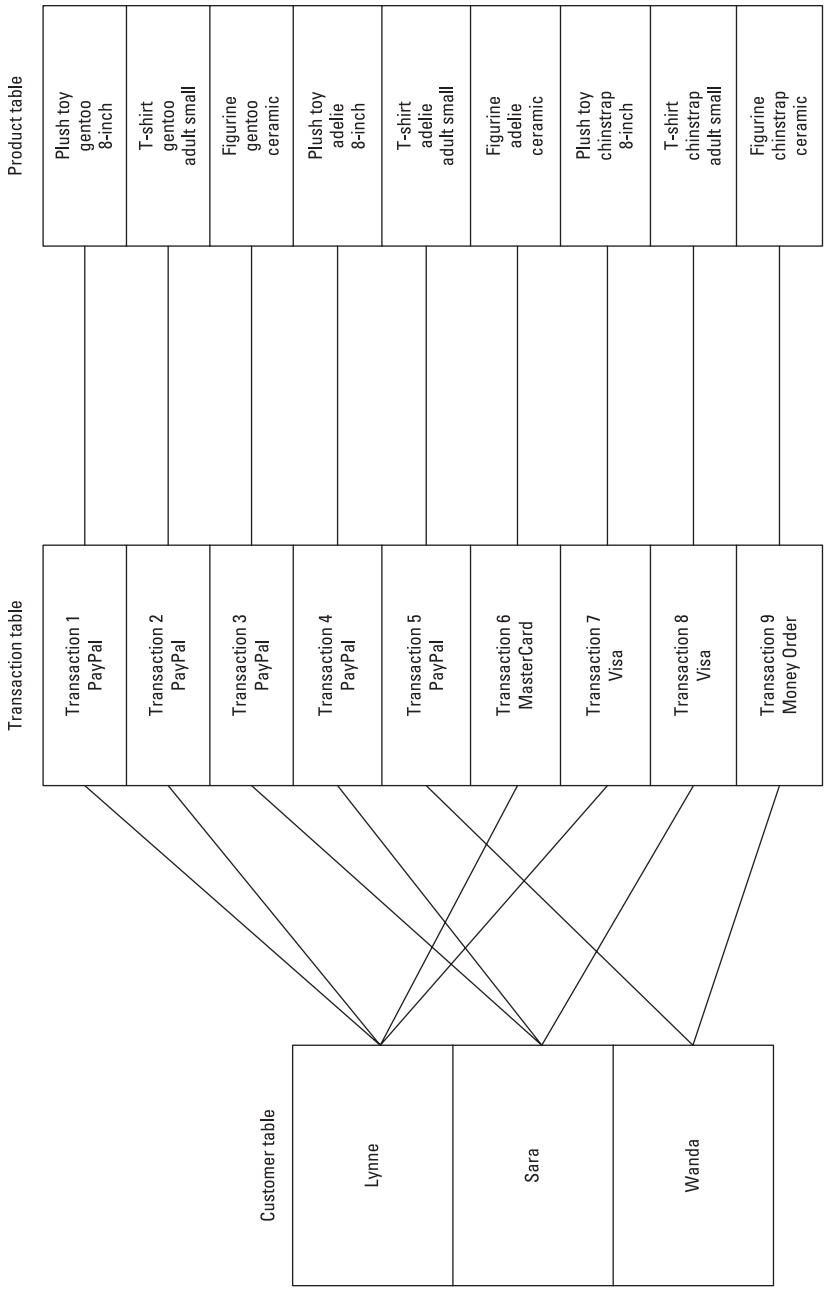

FIGURE 1-4: A relational model of transactions at an online store.

Here are Codd's 12 rules:

1. **The information rule:** Data can be represented only one way, as values in column positions within rows of a table.
2. **The guaranteed access rule:** Every value in a database must be accessible by specifying a table name, a column name, and a row. The row is specified by the value of the primary key.
3. **Systematic treatment of null values:** Missing data is distinct from specific values, such as zero or an empty string.
4. **Relational online catalog:** Authorized users must be able to access the database's structure (its *catalog*) using the same query language they use to access the database's data.
5. **The comprehensive data sublanguage rule:** The system must support at least one relational language that can be used both interactively and within application programs, that supports data definition, data manipulation, and data control functions. Today, that one language is SQL.
6. **The view updating rule:** All views that are theoretically updatable must be updatable by the system.
7. **The system must support set-at-a-time insert, update, and delete operations:** This means that the system must be able to perform insertions, updates, and deletions of multiple rows in a single operation.
8. **Physical data independence:** Changes to the way data is stored must not affect the application.
9. **Logical data independence:** Changes to the tables must not affect the application. For example, adding new columns to a table should not "break" an application that accesses the original rows.
10. **Integrity independence:** Integrity constraints must be specified independently from the application programs and stored in the catalog. (I say a lot about integrity in Book 2, Chapter 3.)
11. **Distribution independence:** Distribution of portions of the database to various locations should not change the way applications function.
12. **The nonsubversion rule:** If the system provides a record-at-a-time interface, it should not be possible to use it to subvert the relational security or integrity constraints.

Over and above the original 12 rules, in 1990, Codd added one more rule:

> **Rule Zero:** For any system that is advertised as, or is claimed to be, a relational database management system, that system must be able to manage databases entirely through its relational capabilities, no matter what additional capabilities the system may support.

Rule Zero was in response to vendors of various database products who claimed their product was a relational DBMS, when in fact it did not have full relational capability.

Highlighting the relational database model's inherent flexibility

You might wonder why it is that relational databases have conquered the planet and relegated hierarchical and network databases to niches consisting mainly of legacy customers who have been using them for more than 40 years. It's even more surprising in light of the fact that when the relational model was first introduced, most of the experts in the field considered it to be utterly uncompetitive with either the hierarchical or the network model.

One advantage of the relational model is its flexibility. The architecture of a relational database is such that it is much easier to restructure a relational database than it is to restructure either a hierarchical or network database. This is a tremendous advantage in dynamic business environments where requirements are constantly changing.

The reason database practitioners originally dissed the relational model is because the extra overhead of the relational database engine was sure to make any product based on that model so much slower than either hierarchical or network databases, as to be noncompetitive. As time has passed, Moore's Law has nullified that objection.

The object-oriented database model

Object-oriented database management systems (OODBMS) first appeared in 1980. They were developed primarily to handle nontext, nonnumeric data such as graphical objects. A relational DBMS typically doesn't do a good job with such so-called complex data types. An OODBMS uses the same data model as object-oriented programming languages such as Java, C++, and C#, and it works well with such languages.

Although object-oriented databases outperform relational databases for selected applications, they do not do as well in most mainstream applications, and have

not made much of a dent in the hegemony of the relational products. As a result, I will not be saying anything more about OODBMS products.

The object-relational database model

An *object-relational database* is a relational database that allows users to create and use new data types that are not part of the standard set of data types provided by SQL. The ability of the user to add new types, called *user-defined types,* was added to the SQL:1999 specification and is available in current implementations of IBM's DB2, Oracle, and Microsoft SQL Server.

Current relational database management systems are actually object-relational database management systems rather than pure relational database management systems.

The nonrelational NoSQL model

In contrast to the relational model, a nonrelational model has been gaining adherents, particularly in the area of *cloud computing,* where databases are maintained not on the local computer or local area network, but reside somewhere on the Internet. This model, called the NoSQL model, is particularly appropriate for large systems consisting of clusters of servers, accessed over the World Wide Web. CouchDB and MongoDB are examples of DBMS products that follow this model. The NoSQL model is not competitive with the SQL-based relational model for traditional reporting applications.

Why the Relational Model Won

Throughout the 1970s and into the 1980s, hierarchical- and network-based technologies were the database technologies of choice for large organizations. Oracle, the first standalone relational database system to reach the market, did not appear until 1979, and initially met with limited success.

For the following reasons, as well as just plain old inertia, relational databases caught on slowly at first:

- **The earliest implementations of relational database management systems were slow performers.** This was due to the fact that they were required to perform more computations than other database systems to perform the same operation.

- **Most business managers were reluctant to try something new when they were already familiar with one or the other of the older technologies.**
- **Data and applications that already existed for an existing database system would be very difficult to convert to work with a relational DBMS.** For most organizations with an existing hierarchical or network database system, it would be too costly to make a conversion.
- **Employees would have to learn an entirely new way of dealing with data.** This would be very costly, too.

However, things gradually started to change.

Although databases structured according to the hierarchical and network models had excellent performance, they were difficult to maintain. Structural changes to a database took a high level of expertise and a lot of time. In many organizations, backlogs of change requests grew from months to years. Department managers started putting their work on personal computers rather than going to the corporate IT department to ask for a change to a database. IT managers, fearing that their power in the organization was eroding, took the drastic step of considering relational technology.

Meanwhile, Moore's Law was inexorably changing the performance situation. In 1965, Gordon Moore of Intel noted that about every 18 months to 2 years the price of a bit in a semiconductor memory would be cut in half, and he predicted that this exponential trend would continue. A corollary of the law is that for a given cost, the performance of integrated circuit processors would double every 18 to 24 months. Both of these laws have held true for more than 50 years, although the end of the trend is in sight. In addition, the capacities and performance of hard disk storage devices have also improved at an exponential rate, paralleling the improvement in semiconductor chips.

The performance improvements in processors, memories, and hard disks combined to dramatically improve the performance of relational database systems, making them more competitive with hierarchical and network systems. When this improved performance was added to the relational architecture's inherent advantage in structural flexibility, relational database systems started to become much more attractive, even to large organizations with major investments in legacy systems. In many of these companies, although existing applications remained on their current platforms, new applications and the databases that held their data were developed using the new relational technology.