Chapter 1

Relational Database Fundamentals

In This Chapter

- Organizing information
- Defining "database" in digital terms
- Deciphering DBMS
- Comparing database models
- Defining "*relational* database" (can you relate?)
- Considering the challenges of database design

SQL (pronounced *ess-que-ell,* not *see'qwl,* though database geeks still argue about that) is a language specifically designed with databases in mind. SQL enables people to create databases, add new data to them, maintain the data in them, and retrieve selected parts of the data. Introduced in 1970, SQL has grown and advanced over the years to become the industry standard. It is governed by a formal standard maintained by the International Standards Organization (ISO).

Various kinds of databases exist, each adhering to a different model of how the data in the database is organized.

SQL was originally developed to operate on data in databases that follow the *relational model.* Recently, the international SQL standard has incorporated part of the *object model,* resulting in hybrid structures called object-relational databases. In this chapter, I discuss data storage, devote a section to how the relational model compares with other major models, and provide a look at the important features of relational databases.

Before I talk about SQL, however, I want to nail down what I mean by the term *database.* Its meaning has changed, just as computers have changed the way people record and maintain information.

Keeping Track of Things

Today people use computers to perform many tasks formerly done with other tools. Computers have replaced typewriters for creating and modifying documents. They've surpassed electromechanical calculators as the best way to do math. They've also replaced millions of pieces of paper, file folders, and file cabinets as the principal storage medium for important information. Compared to those old tools, of course, computers do much more, much faster — and with greater accuracy. These increased benefits do come at a cost, however: Computer users no longer have direct physical access to their data.

When computers occasionally fail, office workers may wonder whether computerization really improved anything at all. In the old days, a manila file folder only "crashed" if you dropped it — then you merely knelt down, picked up the papers, and put them back in the folder. Barring earthquakes or other major disasters, file cabinets never "went down," and they never gave you an error message. A hard-drive crash is another matter entirely: You can't "pick up" lost bits and bytes. Mechanical, electrical, and human failures can make your data go away into the Great Beyond, never to return.

Taking the necessary precautions to protect yourself from accidental data loss allows you to start cashing in on the greater speed and accuracy that computers provide.

If you're storing important data, you have four main concerns:

- Storing data has to be quick and easy, because you're likely to do it often.
- The storage medium must be reliable. You don't want to come back later and find some (or all) of your data missing.
- Data retrieval has to be quick and easy, regardless of how many items you store.
- You need an easy way to separate the exact information you want *now* from the tons of data that you *don't* want right now.

State-of-the-art computer databases satisfy these four criteria. If you store more than a dozen or so data items, you probably want to store those items in a database.

What Is a Database?

The term *database* has fallen into loose use lately, losing much of its original meaning. To some people, a database is any collection of data items (phone books, laundry lists, parchment scrolls . . . whatever). Other people define the term more strictly.

In this book, I define a *database* as a self-describing collection of integrated records. And yes, that does imply computer technology, complete with programming languages such as SQL.

A *record* is a representation of some physical or conceptual object. Say, for example, that you want to keep track of a business's customers. You assign a record for each customer. Each record has multiple *attributes,* such as name, address, and telephone number. Individual names, addresses, and so on are the *data.*

A database consists of both data and *metadata.* Metadata is the data that describes the data's structure within a database. If you know how your data is arranged, then you can retrieve it. Because the database contains a description of its own structure, it's *self-describing.* The database is *integrated* because it includes not only data items but also the relationships among data items.

The database stores metadata in an area called the *data dictionary,* which describes the tables, columns, indexes, constraints, and other items that make up the database.

Because a flat file system (described later in this chapter) has no metadata, applications written to work with flat files must contain the equivalent of the metadata as part of the application program.

Database Size and Complexity

Databases come in all sizes, from simple collections of a few records to mammoth systems holding millions of records.

A *personal database* is designed for use by a single person on a single computer. Such a database usually has a rather simple structure and a relatively small size. A *departmental* or *workgroup database* is used by the members of a single department or workgroup within an organization. This type of database is generally larger than a personal database and is necessarily more complex; such a database must handle multiple users trying to access the same data at the same time. An *enterprise database* can be huge. Enterprise databases may model the critical information flow of entire large organizations.

What Is a Database Management System?

Glad you asked. A *database management system (DBMS)* is a set of programs used to define, administer, and process databases and their associated applications. The database being managed is, in essence, a structure that you build to hold valuable data. A DBMS is the tool you use to build that structure and operate on the data contained within the database.

You can find many DBMS programs on the market today. Some run only on mainframe computers, some only on minicomputers, and some only on personal computers. A strong trend, however, is for such products to work on multiple platforms or on networks that contain all three classes of machines. An even newer trend is to distribute data over a *storage area network* (SAN) or even to store it out on the Internet.

A DBMS that runs on platforms of multiple classes, large and small, is called *scalable.*

Whatever the size of the computer that hosts the database — and regardless of whether the machine is connected to a network — the flow of information between database and user is always the same. Figure 1-1 shows that the user communicates with the database through the DBMS. The DBMS masks the physical details of the database storage so that the application only has to concern itself with the logical characteristics of the data, not with how the data is stored.

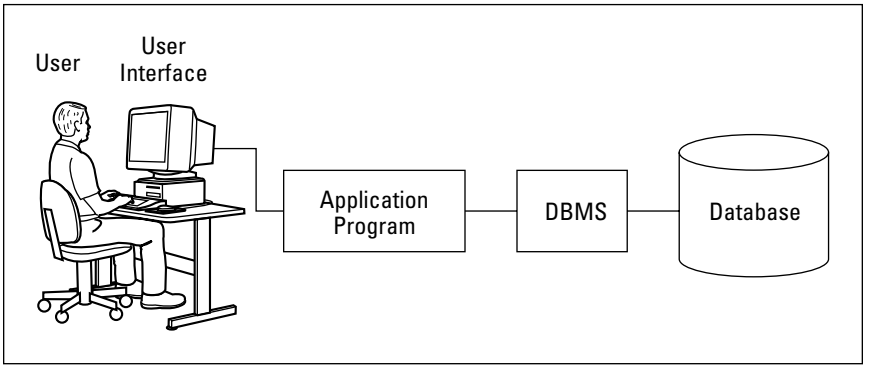

Figure 1-1: Block diagram of a DBMS-based information system.

The value is not in the data, but in the structure

Years ago, some clever person calculated that if you reduce human beings to their components of carbon, hydrogen, oxygen, and nitrogen atoms (plus traces of others), they would be worth only 97 cents. However droll this assessment, it's misleading. People aren't composed of mere isolated collections of atoms. Our atoms combine into enzymes, proteins, hormones, and many other substances that would cost millions of dollars per ounce on the pharmaceutical market. The precise structure of these combinations of atoms is what gives them greater value. By analogy, database structure makes possible the interpretation of seemingly meaningless data. The structure brings to the surface patterns, trends, and tendencies in the data. Unstructured data — like uncombined atoms — has little or no value.

Flat Files

Where structured data is concerned, the flat file is as simple as it gets. No, a flat file isn't a folder that's been squashed under a stack of books. *Flat files* are so called because they have minimal structure. If they were buildings, they'd barely stick up from the ground. A flat file is simply a collection of data records, one after another, in a specified format — the data, the whole data, and nothing but the data — in effect, a list. In computer terms, a flat file is simple. Because the file doesn't store structural information (metadata), its overhead (stuff in the file that is not data but takes up storage space) is minimal.

Say that you want to keep track of the names and addresses of your company's customers in a flat file system. The system may have a structure something like this:

```
Harold Percival  26262 S. Howards Mill Rd  Westminster  CA92683
Jerry Appel      32323 S. River Lane Rd    Santa Ana    CA92705
Adrian Hansen    232 Glenwood Court        Anaheim      CA92640
John Baker       2222 Lafayette St         Garden Grove CA92643
Michael Pens     77730 S. New Era Rd       Irvine       CA92715
Bob Michimoto    25252 S. Kelmsley Dr      Stanton      CA92610
Linda Smith      444 S.E. Seventh St       Costa Mesa   CA92635
Robert Funnell   2424 Sheri Court          Anaheim      CA92640
Bill Checkal     9595 Curry Dr             Stanton      CA92610
Jed Style        3535 Randall St           Santa Ana    CA92705
```

As you can see, the file contains nothing but data. Each field has a fixed length (the `Name` field, for example, is always exactly 15 characters long), and no structure separates one field from another. The person who created the database assigned field positions and lengths. Any program using this file must "know" how each field was assigned, because that information is not contained in the database itself.

Such low overhead means that operating on flat files can be very fast. On the minus side, however, application programs must include logic that manipulates the file's data at a very detailed level. The application must know exactly where and how the file stores its data. Thus, for small systems, flat files work fine. The larger a system is, however, the more cumbersome a flat-file system becomes.

Using a database instead of a flat-file system eliminates duplication of effort. Although database files themselves may have more overhead, the applications can be more portable across various hardware platforms and operating systems. A database also makes writing application programs easier because the programmer doesn't need to know the physical details of where and how the data is stored.

Databases eliminate duplication of effort, because the DBMS handles the data-manipulation details. Applications written to operate on flat files must include those details in the application code. If multiple applications all access the same flat-file data, these applications must all (redundantly) include that data-manipulation code. If you're using a DBMS, however, you don't need to include such code in the applications at all.

Clearly, if a flat-file-based application includes data-manipulation code that only runs on a particular hardware platform, migrating the application to a new platform is a headache waiting to happen. You have to change all the hardware-specific code — and that's just for openers. Migrating a similar DBMS-based application to another platform is much simpler — fewer complicated steps, fewer aspirin consumed.

Database Models

Different as databases may be in size, they are generally always structured according to one of three database models:

- **Hierarchical:** These databases arrange their data in a simple hierarchical structure that allows fast access. They suffer from redundancy problems and their structural inflexibility makes database modification difficult.
- **Network:** Network databases have minimal redundancy but pay for that advantage with structural complexity.

- **Relational:** These databases store their data in tables that are related to each other. Nowadays, new installations of database management systems are almost exclusively of the relational type. Organizations that already have a major investment in hierarchical or network technology may add to the existing model, but groups that have no need to maintain compatibility with such so-called *legacy* systems nearly always choose the relational model for their databases.

The first databases to see wide use were large organizational databases that today would be called *enterprise databases,* built according to either the hierarchical model or the network model. Systems built according to the relational model followed several years later. SQL is a strictly modern language; it applies only to the relational model and its descendant, the object-relational model. So here's where this book says, "So long, it's been good to know ya," to the hierarchical and network models.

New database management systems that aren't based on the relational model probably conform to the (newer) object model or the (hybrid) object-relational model.

Relational model

Dr. E. F. Codd of IBM first formulated the relational database model in 1970, and this model started appearing in products about a decade later. Ironically, IBM did not deliver the first relational DBMS. That distinction went to a small start-up company, which named its product Oracle.

Relational databases have almost completely replaced earlier database types. That's largely because you can change the structure of a relational database without having to change or modify applications that were based on the old structures. Suppose, for example, that you add one or more new columns to a database table. You don't need to change any previously written applications that process that table — unless, of course, you alter one or more of the columns that those applications have to use.

Of course, if you remove a column that an existing application has to use, you experience problems no matter what database model you follow. One of the quickest ways to make a database application crash is to ask it to retrieve a kind of data that your database doesn't contain.

Why relational is better

In applications written with DBMSs that follow the hierarchical or network model, database structure is *hard-coded* into the application. That is, the

application is dependent on the specific physical implementation of the database. If you add a new attribute to the database, you must change your application to accommodate the change, whether or not the application uses the new attribute. An unmodified application will expect the data to be arranged according to the old layout, so it will produce garbage when it writes data into the file that now contains the new attribute.

Relational databases offer structural flexibility; applications written for those databases are easier to maintain than similar applications written for hierarchical or network databases. That same structural flexibility enables you to retrieve combinations of data that you may not have anticipated needing at the time of the database's design.

Components of a relational database

Relational databases gain their flexibility because their data resides in tables that are largely independent of each other. You can add, delete, or change data in a table without affecting the data in the other tables, provided that the affected table is not a *parent* of any of the other tables. (Parent-child table relationships are explained in Chapter 5, and no, they don't involve discussing allowances over dinner.) In this section, I show what these tables consist of and how they relate to the other parts of a relational database.

Dealing with your relations

At holiday time, many of my relatives come to my house and sit down at my table. Databases have relations, too, but each of their relations has its *own* table. A relational database is made up of one or more relations.

A *relation* is a two-dimensional array of rows and columns, containing single-valued entries and no duplicate rows. Each cell in the array can have only one value, and no two rows may be identical. If that's a little hard to picture, here's an example that will put you in the right ballpark. . . .

Most people are familiar with *two-dimensional* arrays of rows and columns, in the form of electronic spreadsheets such as Microsoft Excel. A major-league baseball player's offensive statistics, as listed on the back of baseball card, are an example of such an array. On the baseball card are columns for

year, team, games played, at-bats, hits, runs scored, runs batted in, doubles, triples, home runs, bases on balls, steals, and batting average. A row covers each year that the player has played in the Major Leagues. You can also store this data in a relation (a table), which has the same basic structure. Figure 1-2 shows a relational database table holding the offensive statistics for a single major-league player. In practice, such a table would hold the statistics for an entire team — or perhaps the whole league.

Figure 1-2: A table showing a baseball player's offensive statistics.

Player	Year	Team	Game	At Bat	Hits	Runs	RBI	2B	3B	HR	Walk	Steals	Bat. Avg.
Roberts	1988	Padres	5	9	3	1	0	0	0	0	1	0	.333
Roberts	1989	Padres	117	329	99	81	25	15	8	3	49	21	.301
Roberts	1990	Padres	149	556	172	104	44	36	3	9	55	46	.309

Columns in the array are *self-consistent:* A column has the same meaning in every row. If a column contains a player's last name in one row, the column must contain a player's last name in all rows. The order in which the rows and columns appear in the array has no significance. As far as the DBMS is concerned, it doesn't matter which column is first, which is next, and which is last. The same is true of rows. The DBMS processes the table the same way regardless of the organization.

Every column in a database table embodies a single attribute of the table, just like that baseball card. The column's meaning is the same for every row of the table. A table may, for example, contain the names, addresses, and telephone numbers of all an organization's customers. Each row in the table (also called a *record,* or a *tuple*) holds the data for a single customer. Each column holds a single *attribute* — such as customer number, customer name, customer street, customer city, customer state, customer postal code, or customer telephone number. Figure 1-3 shows some of the rows and columns of such a table.

The *relations* in this database model correspond to *tables* in any database based on the model. Try to say that ten times fast.

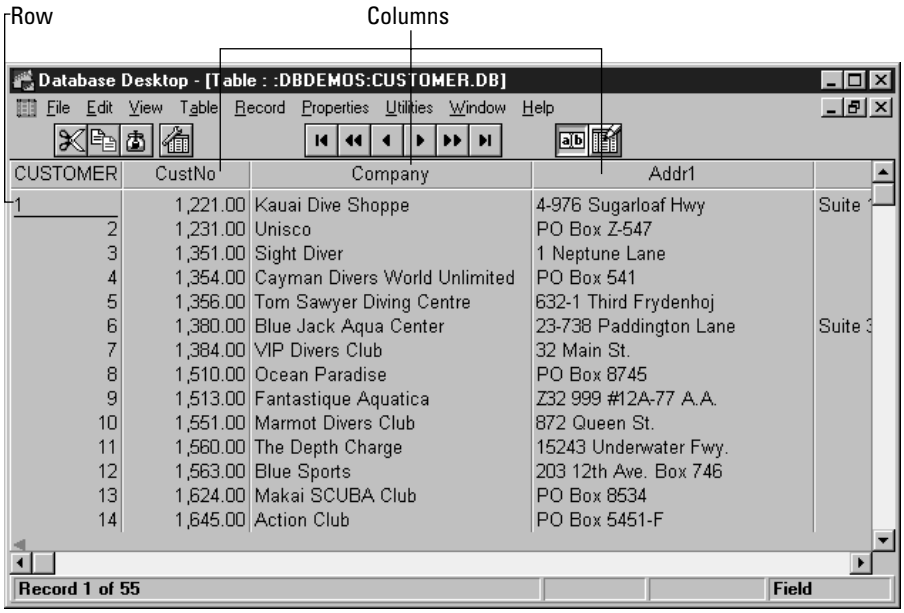

Figure 1-3: Each database row contains a record; each database column holds a single attribute.

Enjoy the view

One of my favorite views is of the Yosemite Valley from the mouth of the Wawona Tunnel, late on a spring afternoon. Golden light bathes the sheer face of El Capitan, Half Dome glistens in the distance, and Bridal Veil Falls forms a silver cascade of sparkling water, while a trace of wispy clouds weaves a tapestry across the sky. Databases have views as well — even if they're not quite that picturesque. The beauty of database views is their sheer usefulness when you're working with your data.

Tables can contain many columns and rows. Sometimes all that data interests you, and sometimes it doesn't. Only some columns of a table may interest you, or perhaps you want to see only rows that satisfy a certain condition. Some columns of one table and some other columns of a related table may interest you. To eliminate data that isn't relevant to your current needs, you can create a *view* — a subset of a database that an application can process. It may contain parts of one or more tables.

Views are sometimes called *virtual tables.* To the application or the user, views behave the same as tables. Views, however, have no independent existence. Views allow you to look at data, but views are not part of the data.

Say, for example, that you're working with a database that has a CUSTOMER table and an INVOICE table. The CUSTOMER table has the columns `CustomerID`, `FirstName`, `LastName`, `Street`, `City`, `State`, `Zipcode`, and `Phone`. The INVOICE table has the columns `InvoiceNumber`, `CustomerID`, `Date`, `TotalSale`, `TotalRemitted`, and `FormOfPayment`.

A national sales manager wants to look at a screen that contains only the customer's first name, last name, and telephone number. Creating from the CUSTOMER table a view that contains only the `FirstName`, `LastName`, and `Phone` columns enables the manager to view what he or she needs without having to see all the unwanted data in the other columns. Figure 1-4 shows the derivation of the national sales manager's view.

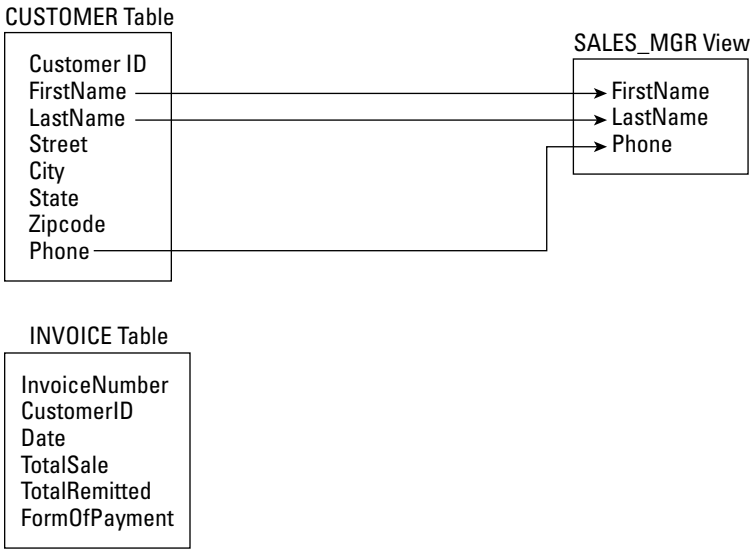

Figure 1-4: The sales manager's view derives from the CUSTOMER table.

A branch manager may want to look at the names and phone numbers of all customers whose zip codes fall between 90000 and 93999 (southern and central California). A view that places a restriction on the rows it retrieves, as well as the columns it displays, does the job. Figure 1-5 shows the sources for the columns in the branch manager's view.

The accounts-payable manager may want to look at customer names from the CUSTOMER table and `Date`, `TotalSale`, `TotalRemitted`, and `FormOfPayment` from the INVOICE table, where `TotalRemitted` is less than `TotalSale`. The latter would be the case if full payment hasn't yet been made. This need requires a view that draws from both tables. Figure 1-6

shows data flowing into the accounts-payable manager's view from both the CUSTOMER and INVOICE tables.

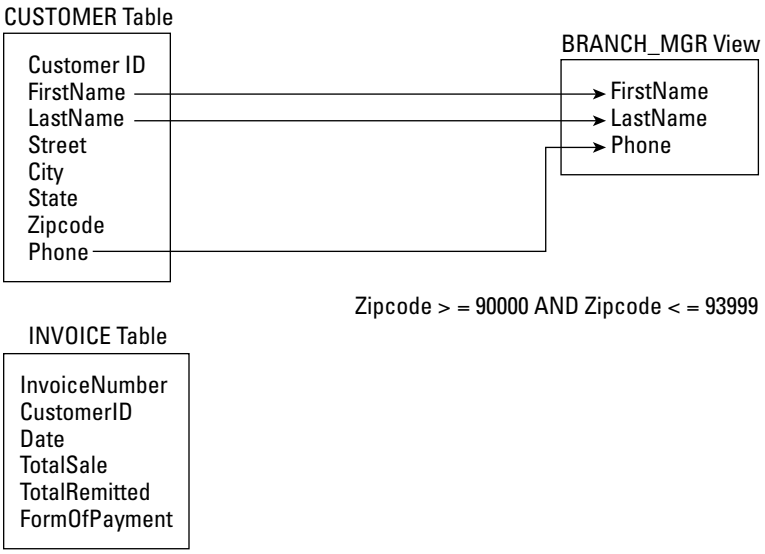

Figure 1-5: The branch manager's view includes only certain rows from the CUSTOMER table.

Views are useful because they enable you to extract and format database data without physically altering the stored data. They also protect the data that you *don't* want to show, because they don't contain it. Chapter 6 illustrates how to create a view by using SQL.

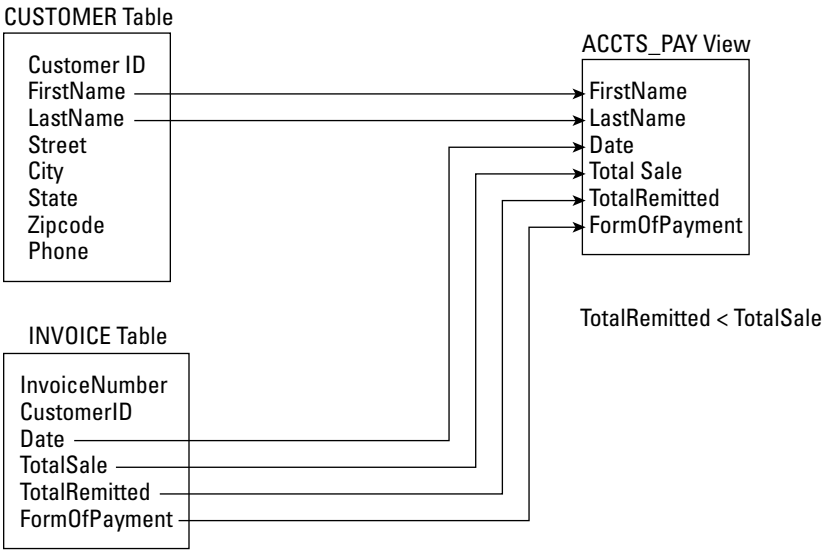

Figure 1-6: The accounts-payable manager's view draws from two tables.

Schemas, domains, and constraints

A database is more than a collection of tables. Additional structures, on several levels, help to maintain the data's integrity. A database's *schema* provides an overall organization to the tables. The *domain* of a table column tells you what values you may store in the column. You can apply *constraints* to a database table to prevent anyone (including yourself) from storing invalid data in the table.

Schemas

The structure of an entire database is its *schema,* or *conceptual view.* This structure is sometimes also called the *complete logical view* of the database. The schema is metadata — as such, it's part of the database. The metadata itself, which describes the database's structure, is stored in tables that are just like the tables that store the regular data. Even metadata is data; that's the beauty of it.

Domains

An attribute of a relation (that is, a column of a table) can assume some finite number of values. The set of all such values is the *domain* of the attribute.

Say, for example, that you're an automobile dealer who handles the newly introduced Curarri GT 4000 sports coupe. You keep track of the cars you have in stock in a database table that you name INVENTORY. You name one of the table columns `Color`, which holds the exterior color of each car. The GT 4000 comes in only four colors: blazing crimson, midnight black, snowflake white, and metallic gray. Those four colors are the domain of the `Color` attribute.

Constraints

Constraints are an important, although often overlooked, component of a database. Constraints are rules that determine what values the table attributes can assume.

By applying tight constraints to a column, you can prevent people from entering invalid data into that column. Of course, every value that is legitimately in the domain of the column must satisfy all the column's constraints. As I mention in the preceding section, a column's domain is the set of all values that the column can contain. A constraint is a restriction on what a column may contain. The characteristics of a table column, plus the constraints that apply to that column, determine the column's domain. By applying constraints, you can prevent users from entering data into a column that falls outside the column's domain.

In the auto dealership example, you can constrain the database to accept only those four values in the `Color` column. If a data entry operator then tries to enter in the `Color` column a value of, for example, `forest green`, the system refuses to accept the entry. Data entry can't proceed until the operator enters a valid value into the `Color` field.

You may wonder what happens when the Curarri AutoWerks decides to offer a forest-green version of the GT 4000 as a mid-year option. The answer is (drum roll, please) job security for database-maintenance programmers. This kind of thing happens all the time and requires updates to the database structure. Only people who know how to modify the database structure (such as you) will be able to prevent a major snafu.

The object model challenges the relational model

The relational model has been fantastically successful in a wide variety of application areas. However, it does not do everything that anyone would ever want. The limitations have been made more visible by the rise in popularity of object-oriented programming languages such as C++, Java, and C#. Such languages are capable of handling more complex problems than traditional languages due to their advanced features, such as user-extensible type systems, encapsulation, inheritance, dynamic binding of methods, complex and composite objects, and object identity.

I am not going to explain all that jargon in this book (although I do touch on some of these terms later). Suffice it to say that the classic relational model doesn't mesh well with many of these features. As a result, database management systems based on the object model have been developed and are available on the market. As yet, their market share is relatively small.

The object-relational model

Database designers, like everyone else, are constantly searching for the best of all possible worlds. They mused, "Wouldn't it be great if we could have the advantages of an object-oriented database system, and still retain compatibility with the relational system that we have come to know and love?" This kind of thinking led to the hybrid object-relational model. Object-relational DBMSs extend the relational model to include support for object-oriented data modeling. Object-oriented features have been added to the international SQL standard, allowing relational DBMS vendors to transform their products into object-relational DBMSs, while retaining

compatibility with the standard. Thus, whereas the SQL-92 standard describes a purely relational database model, SQL:1999 describes an object-relational database model. SQL:2003 has more object-oriented features, and SQL:2008 goes even further in that direction.

In this book, I describe ISO/IEC international standard SQL. This is primarily a relational database model. I also include the object-oriented extensions to the standard that were introduced in SQL:1999, and the additional extensions included in SQL:2003 and SQL:2008. The object-oriented features of the new standard allow developers to apply SQL databases to problems that are too complex to address with the older, purely relational, paradigm. Vendors of DBMS systems are incorporating the object-oriented features in the ISO standard into their products. Some of these features have been present for years, while others are yet to be included.

Database Design Considerations

A database is a representation of a physical or conceptual structure, such as an organization, an automobile assembly, or the performance statistics of all the major-league baseball clubs. The accuracy of the representation depends on the level of detail of the database design. The amount of effort that you put into database design should depend on the type of information you want to get out of the database. Too much detail is a waste of effort, time, and hard drive space. Too little detail may render the database worthless.

Decide how much detail you need now and how much you may need in the future — and then provide exactly that level of detail in your design (no more and no less). But don't be surprised if you have to adjust the design eventually to meet changing real-world needs.

Today's database management systems, complete with attractive graphical user interfaces and intuitive design tools, can give the would-be database designer a false sense of security. These systems make designing a database seem comparable to building a spreadsheet or engaging in some other relatively straightforward task. No such luck. Database design is difficult. If you do it incorrectly, not only is your database likely to suffer from poor performance, but it also may well become gradually more corrupt as time goes on. Often the problem doesn't turn up until after you devote a great deal of effort to data entry. By the time you know that you have a problem, it's already serious. In many cases, the only solution is to completely redesign the database and reenter all the data. The up side is that by the time you finish your second version of the same database, you realize how much better you understand database design.