4

Designing a Relational Database

Chapter 1 introduced you to databases and databases management systems. As you'll recall from that discussion, a database is a collection of related data organized and classified in a structured format that is defined by metadata. Not all databases are structured the same, though, as can be attested to by the different data models that have emerged over the years. Yet many of these models — and subsequently the systems that were built on them — lacked the flexibility necessary to support increasingly sophisticated software applications. One data model emerged that addressed the limitations of its predecessors and provided the flexibility necessary to meet the demands of today's application technologies. This model — the relational model — has become the standard on which most database management systems are now built.

MySQL is one of the database management systems based on the relational model. As a result, to design effective databases, you should have a good understanding of that model and how it applies to database design. To that end, this chapter provides you with a conceptual overview of the relational model and explains the components that make up a relational database. The chapter also discusses how data is organized in the relational model and how tables of data are related to one another. Specifically, the chapter covers the following topics:

❑   You are introduced to the relational model and the components that make up that model.

❑   You learn how data in a relational structure is organized according to normal forms, which are prescriptive methods for organizing data in a relational database.

❑   You are provided with the information necessary to identify the relationships between tables in a relational database, including one-to-one, one-to-many, and many-to-many relationships.

❑   You learn how to create a data model. The process includes identifying entities, normalizing data, identifying relationships, and refining the data model.

The Relational Model

The relational model first entered the database scene in 1970, when Dr. E. F. Codd published his seminal work, "A Relational Model of Data for Large Shared Data Banks" in the journal *Communication of*

the ACM, Volume 13, Number 6 (June 1970). In this paper, Codd introduced a relational data structure that was based on the mathematical principles of set theory and predicate logic. The data structure allowed data to be manipulated in a manner that was predictable and resistant to error. To this end, the relational model would enforce data accuracy and consistency, support easy data manipulation and retrieval, and provide a structure independent of the applications accessing the data.

At the heart of the relational model — and of any relational database — is the *table,* which is made up of a set of related data organized in a column/row structure, similar to what you might see in a spreadsheet program such as Microsoft Excel. The data represents the physical implementation of some type of object that is made up of a related set of data, such entities as people, places, things, or even processes. For example, the table in Figure 4-1 contains data about authors. Notice that all the values in the table are in some way related to the central theme of authors.

Column name

Authors table

AuthID	AuthFN	AuthMN	AuthLN	Born	Died
1001	Edith	NULL	Wharton	1862	1937
1002	Kate	NULL	Chopin	1850	1904
1003	T.	S.	Eliot	1888	1965
1004	Joyce	Carol	Oates	1938	NULL
1005	Samuel	L.	Clemens	1835	1910

Rows

Primary key   Columns

Figure 4-1

Each table in a relational database is made up of one or more columns. A *column* is a structure in a table that stores the same type of data. For example, the table in Figure 4-1 includes a column named AuthMN. The column is intended to hold a value for each author that is equal to the author's middle name, if one is known. The column can hold only data that conforms to the restrictions placed on that column.

Along with the columns, each table contains zero or more rows. A *row* is a structure in a table made up of a set of related data. A row can be thought of as a record that represents one instance of the object defined by the table. For example, in Figure 4-1 the last row contains data about the author Samuel L. Clemens, who was born in 1835 and died in 1910. The author is identified by an AuthID value of 1005. Together, this set of data makes up one row.

One other item that is important to note in a table is the primary key. A *primary key* is one or more columns in a table that uniquely identify each row so that no two rows are identical. In Figure 4-1, a primary key is defined on the AuthID column. Because this column is the primary key, no values can be repeated in the column. As a result, even if two authors share the same name, same year of birth, and same year of death, the values in the primary key column would provide a unique identifier for each row.

Although the table serves as the foundation for every relational database, the relational model includes other important characteristics, including the normalization of data and the relationships that exist between tables. The following section discusses how data is normalized in a relational database and, after that, reviews the types of relationships that can exist between tables.

Data Normalization

One of the concepts most important to a relational database is that of normalized data. *Normalized data* is organized into a structure that preserves the integrity of the data while minimizing redundant data. The goal of all normalized data is to prevent lost data and inconsistent data, while minimizing redundant data.

A normalized database is one whose tables are structured according to the rules of normalization. These rules — referred to as *normal forms* — specify how to organize data so that it is considered normalized. When Codd first introduced the relational model, he included three normal forms. Since then, more normal forms have been introduced, but the first three still remain the most critical to the relational model.

The degree to which a database is considered normalized depends on which normal forms can be applied. For example, some database designs aim for only the second normal form; however, some databases strive to achieve conformance to the fourth or fifth normal form. There is often a trade-off between strict adherence to the normal forms and system performance. Often, the more normalized the data, the more taxing it can be on a system. As a result, a database design must strike a balance between a fully normalized database and system performance. In most situations, the first three normal forms provide that balance.

First Normal Form

Of all the normal forms, the first is the most important. It provides the foundation on which all other normal forms are built and represents the core characteristics of any table. To be in compliance with the first normal form, a table must meet the following requirements:

❑ Each column in a row must be atomic. In other words, the column can contain only one value for any given row.

❑ Each row in a table must contain the same number of columns. Given that each column can contain only one value, this means that each row must contain the same number of values.

❑ All rows in a table must be different. Although rows might include the same values, each row, when taken as a whole, must be unique in the table.

Take a look at an example to help illustrate these requirements. Figure 4-2 contains a table that violates the first normal form. For example, the fifth row contains two values in the BookTitle column: *Postcards* and *The Shipping News*. Although a value can consist of more than one word, as in *The Shipping News*, only one value can exist in a column. As a result, the BookTitle column for that row is not atomic because it contains two values. In addition, the row as a whole contains more values than the other rows in the table, which also violates the first normal form.

AuthorBook

AuthFN	AuthMN	AuthLN	BookTitle
Hunter	S.	Thompson	Hell's Angels
Rainer	Maria	Rilke	Letters to a Young Poet
Rainer	Maria	Rilke	Letters to a Young Poet
John	Kennedy	Toole	A Confederacy of Dunces
Annie	NULL	Proulx	Postcards, The Shipping News
Nelson	NULL	Algren	Nonconformity

Figure 4-2

Another way in which the table violates the first normal form is found in the second and third rows, which are identical. Duplicate rows can exist for a number of reasons, and without the necessary data to distinguish them, you cannot tell whether this is an error in data entry or whether there are supposed to be two records for this one book. For example, the rows might be duplicated because they refer to different editions of the book, or perhaps the book has been translated into different languages. The point is, each row in a table must be unique.

In order to conform to the first normal form, you must eliminate the duplicate values in the BookTitle column, ensure that each row contains the same number of values, and avoid duplicated rows. One way to achieve the necessary normalization is to place the data in separate tables, based on the objects represented by the data. In this case, the obvious place to start is with authors and books. All data related to authors is placed in one table, and all data related to books is placed in another table, as shown in Figure 4-3. Notice that a row has been created for each book and that a translation-related column has been added for that table. This eliminates the duplicated rows, as long as two books with the same name are not translated into the same language.

Authors

AuthID	AuthFN	AuthMN	AuthLN
1006	Hunter	S.	Thompson
1007	Rainer	Maria	Rilke
1008	John	Kennedy	Toole
1009	Annie	NULL	Proulx
1010	Nelson	NULL	Algren

AuthorBook

AuthID	BookID
1006	14356
1007	12786
1007	14555
1008	17695
1009	19264
1009	19354
1010	16284

Books

BookID	BookTitle	Trans
14356	Hell's Angels	English
12786	Letters to a Young Poet	English
14555	Letters to a Young Poet	French
17695	A Confederacy of Dunces	English
19264	Postcards	English
19354	The Shipping News	English
16284	Nonconformity	English

Figure 4-3

To get around the possibility of two rows containing data about books with the same name and language, an identifying column (BookID) is added to the table and configured as the primary key (shown in gray). Because the column is the primary key, each value in the column must be unique. As a result, even the rows that contain duplicated book titles and languages remain unique from one another (when taken as a whole). The same is true of the Authors table. Because the AuthID column is defined a primary key (shown in gray), authors can share the same name and each row will still be unique.

By creating tables for both authors and books, adding a primary key column to each table, and placing only one value in each column, you are ensuring that the data conforms to the first normal form. As you can see in Figure 4-3, a third table (AuthorBook) is also being used. This table allows you to match the

IDs for authors and books in a way that supports books written by multiple authors, authors who have written multiple books, and multiple authors who have written multiple books. Had you tried to match the authors to their books in one of the two tables, the table would potentially fill with enormous amounts of redundant data, which would defeat one of the purposes of the relational database.

Another thing to notice is that a primary key has been defined on the AuthorBook table. The primary key is made up of two columns. (Both columns are shown in gray.) As a result, each set of values in the two columns must be unique. In other words, no two rows can contain the same AuthID *and* BookID values, although values can be repeated in individual columns. For example, the AuthID value of 1009 is repeated, but each instance of that value is associated with a different BookID value. Because of the primary key, no AuthID value can be associated with a BookID value more than once.

> *A primary key made up of more than one column is referred to as a* composite primary key.

Creating this table might appear as though all you've done is to create a lot more data than you initially had to start. The example here, however, represents only a small amount of data. The advantages of normalizing data are best seen when working with large quantities of data.

Second Normal Form

The second normal form builds on and expands the first normal form. To be in compliance with the second normal form, a table must meet the following requirements:

❑   The table must be in first normal form.

❑   All nonprimary key columns in the table must be dependent on the entire primary key.

Given that the first of these two rules is fairly obvious, take a look at the second one. As you'll recall from earlier in the chapter, a primary key is one or more columns in a table that uniquely identify each row so that no two rows, when taken as a whole, are identical. To illustrate how the second normal form works, first take a look at an example of a table that violates the second normal form. In the AuthorBook table in Figure 4-4, a primary key is defined on the AuthLN and BookTitle columns. Together, the values in these two columns must uniquely identify each row in the table.

AuthorBook

AuthFN	AuthMN	AuthLN	BookTitle	Copyright
Hunter	S.	Thompson	Hell's Angels	1966
Rainer	Maria	Rilke	Letters to a Young Poet	1934
John	Kennedy	Toole	A Confederacy of Dunces	1980
Annie	NULL	Proulx	Postcards	1992
Annie	NULL	Proulx	The Shipping News	1993
Nelson	NULL	Algren	Nonconformity	1996

Primary key

Figure 4-4

You can see how the primary key works in the fourth and fifth rows, which are related to the author Annie Proulx. Although both rows are concerned with the same author, they refer to different books. As a result, the values Proulx and Postcards identify one row, and the values Proulx and The Shipping News identify the second row. Although the values in either one of the individual primary key columns can be duplicated (in that column), the values in both columns, when taken as a whole, must be unique. This is another example of a composite primary key.

Now examine how this table applies to the second normal form. As previously stated, all nonprimary key columns in the table must be dependent on the entire primary key, which, in this case, is made up of the author's last name and the book title. Based on the way that the table is currently defined, the AuthFN and AuthMN columns are dependent on the AuthLN column, and the Copyright column is dependent on the BookTitle column. The AuthFN and AuthMN columns are not dependent on the BookTitle column, though, and the Copyright column is not dependent on the AuthLN column. As a result, the table violates the second normal form.

Another problem with the table is the columns used for the primary key. By defining the primary key in this way, you're assuming that two authors with the same last name won't write a book with the same title and that no one author will write two books with the same title. This assumption, though, might not necessarily be true. If two authors with the same last name write books with the same title, the primary key would prevent you from adding the second book to the table.

The most effective way to normalize the data in the AuthorBook table is to use the solution that you saw for the first normal form: Create a table for the authors and one for the books, add a primary key column to each table, and create a third table that matches up the identifiers for authors and books, as shown in Figure 4-5. For the Authors table, the primary key is the AuthID column, and for the Books, table, the primary key is the BookID column. Now the columns in each table are dependent on their respective primary keys, and no columns exist that are not dependent on the primary key.

Authors

AuthID	AuthFN	AuthMN	AuthLN
1006	Hunter	S.	Thompson
1007	Rainer	Maria	Rilke
1008	John	Kennedy	Toole
1009	Annie	NULL	Proulx
1010	Nelson	NULL	Algren

AuthorBook

AuthID	BookID
1006	14356
1007	12786
1008	17695
1009	19264
1009	19354
1010	16284

Books

BookID	BookTitle	Copyright
14356	Hell's Angels	1966
12786	Letters to a Young Poet	1934
17695	A Confederacy of Dunces	1980
19264	Postcards	1992
19354	The Shipping News	1993
16284	Nonconformity	1996

Figure 4-5

In addition, a primary key has been defined on the AuthID and BookID columns of the AuthorBook table. As a result, any of the primary key columns in a row, when taken as a whole, must be unique from all other rows. Because there are no other columns in this table, the issue of dependent columns is not a concern, so you can assume that this table also conforms to the second normal form.

Third Normal Form

As with the second normal form, the third normal form builds on and expands the previous normal form. To be in compliance with the third normal form, a table must meet the following requirements:

❑ The table must be in second normal form.

❑ All nonprimary key columns in the table must be dependent on the primary key and must be independent of each other.

If you take a look at Figure 4-6, you see an example of a table that violates the third normal form. Notice that a primary key is defined on the BookID column. For each book, there is a unique ID that identifies that book. No other book can have that ID; therefore, all characteristics related to that book are dependent on that ID. For example, the BookTitle and Copyright columns are clearly dependent on the primary key. For each book ID, there is a title and a copyright date.

To illustrate this better, take a look at the first row in the table. As you can see, the book is assigned a BookID value of 14356. The title for this ID is Hell's Angels, and the copyright is 1966. Once that ID is assigned to that title and copyright, that title and copyright become dependent on that ID. It identifies that title and copyright as part of a unique row. Despite their dependence on the primary key, the BookTitle and Copyright columns are independent from each other. In other words, you can include the BookTitle and the Copyright columns, but you don't necessarily need to include both because one isn't dependent on the other for their meaning. The ChineseSign column is very different from the BookTitle and Copyright columns. It provides the Chinese astrological year sign for the year that the book was copyrighted. The ChineseSign value has nothing to do with the BookID and is not related to the book itself. Instead, the ChineseSign column is totally dependent on the Copyright column. Without the Copyright column, the ChineseSign column would have no meaning. As a result, the ChineseSign column violates the third normal form.

Books

BookID	BookTitle	Copyright	ChineseSign
14356	Hell's Angels	1966	Horse
12786	Letters to a Young Poet	1934	Dog
17695	A Confederacy of Dunces	1980	Monkey
19264	Postcards	1992	Monkey
19354	The Shipping News	1993	Rooster
16284	Nonconformity	1996	Rat

Figure 4-6

To ensure that the data conforms to the third normal form, you should separate the data into two tables, one for books and one for Chinese astrological year, as shown in Figure 4-7. From there, you should assign a primary key to the Year column of the ChineseYears table. Because each year must be unique, it is a good candidate for a primary key column. You don't necessarily have to add a column to a table to use as a primary key if an existing column or columns will work.

By separating the data into two tables, each column is now dependent on its respective primary key, and no columns are dependent on nonkey columns.

Books

BookID	BookTitle	Copyright
14356	Hell's Angels	1966
12786	Letters to a Young Poet	1934
17695	A Confederacy of Dunces	1980
19264	Postcards	1992
19354	The Shipping News	1993
16284	Nonconformity	1996

ChineseYears

Year	Sign
1989	Snake
1990	Horse
1991	Goat
1992	Monkey
1993	Rooster
1994	Dog
1995	Pig
1996	Rat
1997	Ox
1998	Tiger
1999	Cat
2000	Dragon

Figure 4-7

By making certain that the data conforms to the third normal form, you're ensuring that it has been normalized according to all three normal forms. And although there are even more normal forms that you can conform to, for the most part, the first three normal forms meet most of your database design needs. If you plan to focus heavily on database design or plan to design complex databases, you're encouraged to research other references for more details about all normal forms and the relational model.

In the meantime, you can go a long way to achieving a normalized database by thinking in terms of separating data into *entities*, discrete categories of information. For example, books represent one entity; publishers represent another. If you keep in mind that, whenever designing a database, you want each table to represent a distinct entity, you go a long way in designing a database that achieves the third normal form.

Relationships

One of the defining characteristics of a relational database is the fact that various types of relationships exist between tables. These relationships allow the data in the tables to be associated with each other in meaningful ways that help ensure the integrity of normalized data. Because of these relationships, actions in one table cannot adversely affect data in another table.

For any relational database, there are three fundamental types of relationships that can exist between tables: one-to-one relationships, one-to-many relationships, and many-to-many relationships. This section takes a look at each of these relationships.

One-to-One Relationships

A one-to-one relationship can exist between any two tables in which a row in the first table can be related to only one row in the second table and a row in the second table can be related to only one row in the first table. The following example demonstrates how this works. In Figure 4-8, a one-to-one relationship exists between the Authors table and the AuthorsBios table. (The line that connects the tables represents the one-to-one relationship that exists between the tables.)

Several different systems are used to represent the relationships between tables, all of which connect the tables with lines that have special notations at the ends of those lines. The examples in this book use a very basic system to represent the relationships.

Authors

AuthID	AuthFN	AuthMN	AuthLN
1001	Edith	NULL	Wharton
1002	Kate	NULL	Chopin
1003	T.	S.	Eliot
1004	Joyce	Carol	Oates
1005	Samuel	L.	Clemens

AuthorsBios

AuthID	Born	Died
1001	1862	1937
1002	1850	1904
1003	1888	1965
1004	1938	NULL
1005	1835	1910

Figure 4-8

Each table includes a primary key that is defined on the AuthID column. For any one row in the Authors table, there can be only one associated row in the AuthorsBios table, and for any one row in the AuthorsBios table, there can be only one associated row in the Authors table. For example, the Authors table includes a row for the author record that has an AuthID value of 1001 (Edith Wharton). As a result, the AuthorsBios table can contain only one row associated with author 1001. In other words, there can be only one biography for each author.

If you refer again to Figure 4-8, you'll see that the AuthorsBios table includes a row that contains an AuthID value of 1004. Because a one-to-one relationship exists between the two tables, only one record can exist in the Authors table for author 1004. As a result, only one author can be associated with that author biography.

Generally, one-to-one relationships are the least likely type of relationships to be implemented in a relational database; however, there are sometimes reasons to use them. For example, you might want to separate tables simply because one table would contain too much data, or perhaps you would want to separate data into different tables so you could set up one table with a higher level of security. Even so, most databases contain relatively few, if any, one-to-one relationships. The most common type of relationship you're likely to find is the one-to-many.

One-to-Many Relationships

As with one-to-one relationships, a one-to-many relationship can exist between any two tables in your database. A one-to-many relationship differs from a one-to-one relationship in that a row in the first table can be related to one or more rows in the second table, but a row in the second table can be related to only one row in the first table. Figure 4-9 illustrates how the one-to-many relationship works.

Authors

AuthID	AuthFN	AuthMN	AuthLN
1006	Hunter	S.	Thompson
1007	Rainer	Maria	Rilke
1008	John	Kennedy	Toole
1009	Annie	NULL	Proulx
1010	Nelson	NULL	Algren

AuthorBook

AuthID	BookID
1006	14356
1007	12786
1008	17695
1009	19264
1009	19354
1010	16284

Books

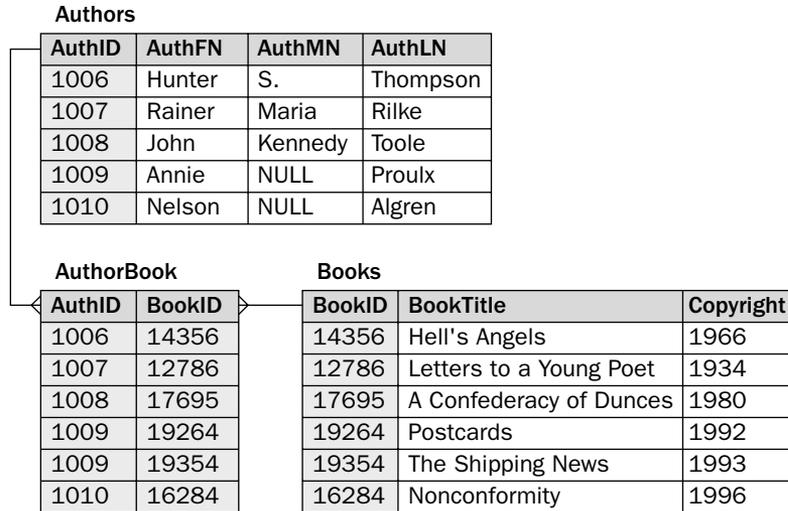| BookID	BookTitle	Copyright
14356	Hell's Angels	1966
12786	Letters to a Young Poet	1934
17695	A Confederacy of Dunces	1980
19264	Postcards	1992
19354	The Shipping News	1993
16284	Nonconformity	1996

Figure 4-9

As you can see in the figure, there are three tables: Authors, AuthorBook, and Books. Notice that the lines connecting the Authors table and the Books table to the AuthorBook table have three prongs on the AuthorBook side. This is sometimes referred to as a *crow's foot*. The three prongs represent the *many* side of the relationship. What this means is that, for every row in the Authors table, there can be one or more associated rows in the AuthorBook table, and for every row in the Books table, there can be one or more associated rows in the AuthorBook table. For every row in the AuthorBook table, however, there can be only one associated row in the Authors table or the Books table.

Notice that each table includes an identifying column designated as the primary key. In the Authors table, the primary key is the AuthID column, and in the Books table, the primary key is the BookID column. For the AuthorBook table, the primary key is defined on both columns. This is another example of a composite primary key.

If you take a look at the AuthorBook table, notice that the first column is AuthID. The column contains AuthID values from the Authors table. This is how the one-to-many relationship is implemented, by referencing the primary key of the *one* side of the relationship in a column on the *many* side. The same thing is true for the Books table. The BookID column in the AuthorBook table contains the BookID values from the Books table. For example, the first row in the AuthorBook table contains an AuthID value of 1006 and a BookID value of 14356. This indicates that author 1006 wrote book 14356. If you now refer to the Authors table, notice that author 1006 is Hunter S. Thompson. If you refer to the Books table, you'll see that book 14356 is *Hell's Angels.* If an author has written more than one book, the AuthorBook table contains more than one row for that author. If a book is written by more than one author, the AuthorBook table contains more than one row for that book.

A one-to-many relationship is probably the most common type of relationship you'll see in your databases. (This would also include the many-to-one relationship, which is simply a reversing of the order in which the tables are physically represented.) The next section deals with the many-to-many relationship.

Many-to-Many Relationships

A many-to-many relationship can exist between any two tables in which a row in the first table can be related to one or more rows in the second table, but a row in the second table can be related to one or more rows in the first table. Take a look at an example to help illustrate how this relationship works. In Figure 4-10, you can see three tables: Authors, AuthorBook, and Books. Authors and Books are connected by a dotted line that represents the many-to-many relationship. There are three prongs on each end. For any one author in the Authors table, there can be one or more associated books. For any one book in the Books table, there can be one or more authors. For example, author 1009 — Annie Proulx — is the author of books 19264 and 19354 — Postcards and The Shipping News, respectively — and authors 1011 and 1012 — Black Elk and John G. Neihardt, respectively — are the authors of Black Elk Speaks.

Authors

AuthID	AuthFN	AuthMN	AuthLN
1006	Hunter	S.	Thompson
1007	Rainer	Maria	Rilke
1008	John	Kennedy	Toole
1009	Annie	NULL	Proulx
1010	Nelson	NULL	Algren

AuthorBook

AuthID	BookID
1006	14356
1007	12786
1008	17695
1009	19264
1009	19354
1010	16284

Books

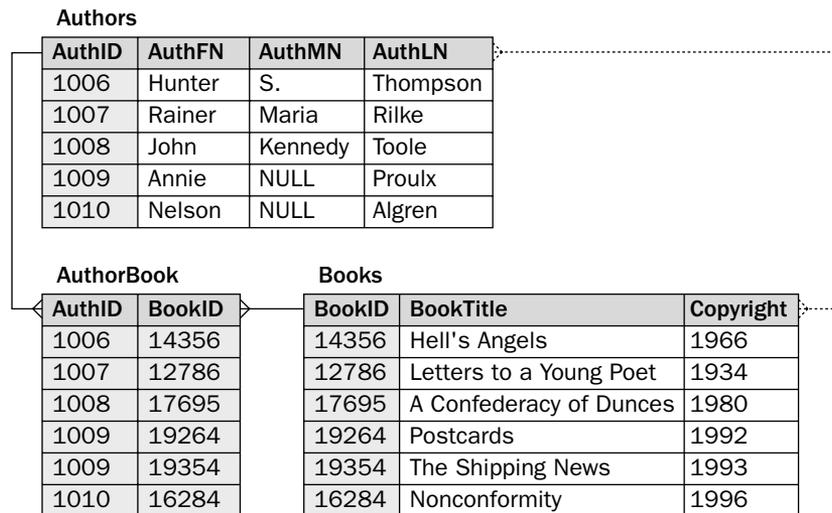| BookID	BookTitle	Copyright
14356	Hell's Angels	1966
12786	Letters to a Young Poet	1934
17695	A Confederacy of Dunces	1980
19264	Postcards	1992
19354	The Shipping News	1993
16284	Nonconformity	1996

Figure 4-10

Also notice in the Authors and Books tables that there is no reference in the Authors table to the books that the authors have written, and there is no reference in the Books table to the authors that have written the books. When a many-to-many relationship exists, it is implemented by adding a third table between these two tables that matches the primary key values of one table to the primary key values of the second table. You saw examples of this during the discussion of normalizing data earlier in this chapter. The many-to-many relationship is logical and is not physically implemented, which is why a dotted line represents the relationship in Figure 4-10. In the next section, "Creating a Data Model," you learn how identifying many-to-many relationships is part of the database design process.

Creating a Data Model

As you have seen in this chapter, the relational model is based on tables that contain normalized data and on the meaningful relationships between those tables. Specifically, a table is made up of columns and rows that form a table-like structure. The rows are identified through the use of primary keys, and the data is normalized based on the rules of the normal forms. From these concepts, you should be able to design a database that adheres to the standards of the relational model.

An important part of the database design process is the development of a data model. A *data model* is a physical representation of the components that make up the database as well as the relationships between those components. A data model for a relational database should show all of the following information:

❑   The tables that make up the database

❑   The columns that make up each table

❑   The data type that defines each column

❑   The primary key that identifies each row

❑   The relationships that exist between tables

Ultimately, a data model should be concerned with how the represented database is implemented in a particular RDBMS. For this reason, some database designers will develop a logical data model and, from that, design a physical data model. A logical data model is concerned with data storage in its purest sense, adhering strictly to the rules of normalization and the relational model, while a physical data model provides a representation of how that database will actually be implemented in a particular RDBMS. The logical model is indifferent to *how* the database will be implemented. The physical model is specific to a particular implementation.

For the purposes of this book, you need to be concerned with only one data model. Because the goal is to design a database specific to MySQL, it is not necessary to create two models.

As part of the data modeling process, you must identify the data type for each column. Because you do not learn about data types until Chapter 5, the types are provided for you. In addition to identifying data types, you need to identify the foreign keys. A *foreign key* is one or more columns in a table whose values match the values in one or more columns in another table. The values in the foreign key of the first table usually come from the primary key in the second table. Use a foreign key when a relationship exists between two tables. The foreign key allows you to associate data between tables. As you get deeper into the topic of data modeling, you get a better sense of how foreign keys work.

Now take a look at an example to help illustrate how to use a data model to design a database. Figure 4-11 shows a model that contains five tables: Authors, AuthorBook, Publishers, BookPublisher, and Books. Each table is represented by a rectangle, with the name of the table on top. In each rectangle there is a list of columns — along with their data types — included in that particular table.

Now take a closer look at one of the tables. The Books table includes four columns: BookID, BookTitle, Copyright, and PubID. Each column is listed with the data type that defines the type of data that is permitted in the column. For example, the BookTitle column is defined with the VARCHAR(60) data type. Data types are assigned to columns based on the business rules defined for that database and the restrictions of data types in MySQL. Chapter 5 discusses data types in far greater detail. For now, all you need to know is that a particular data type has been assigned to a specific column. Once you're more familiar with data types, you can assign them yourself.

Returning to the Books table in Figure 4-11, another aspect to notice is that a line separates the BookID column from the other columns. In a data model such as this one, any column listed above the line participates in the primary key. As a result, the primary key in the Books table is made up of the BookID column and no other columns.
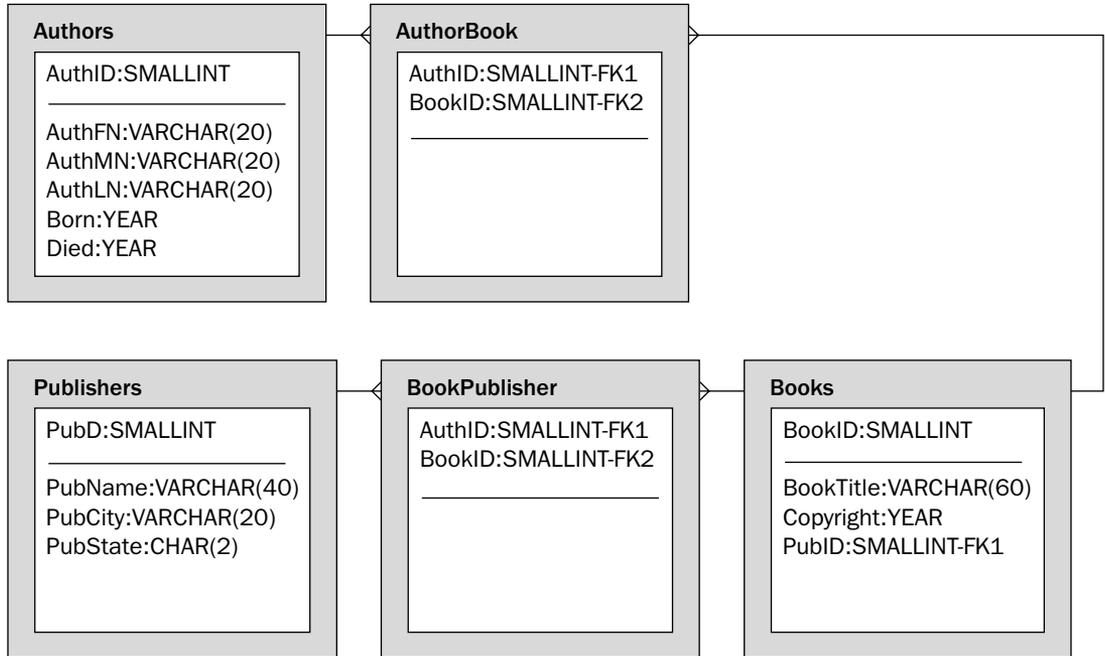
Figure 4-11

One other aspect to notice in the Books table is the PubID column, which is followed by FK1. *FK* is an acronym for foreign key. As a foreign key, the PubID column contains values from the associated data in the PubID column in the Publishers table. The foreign key exists because a relationship exists between the two tables. The relationship is designated by the line that connects the two tables and the three prongs at one end, indicating that the tables are participating in a one-to-many relationship. As a result, a publisher listed in the Publishers table can publish one or more books, but only one publisher can publish a book, and that publisher is identified by the value in the foreign key column (PubID).

Also notice in Figure 4-11 that a number that follows the FK designates each foreign key table. For example, the AuthorBook table includes an FK1 and an FK2 because there are two foreign keys. The numbers are used because some tables might include multiple foreign keys in which one or more of those keys are made up of more than one column. If this were to occur, it could become confusing as to which columns participate in which foreign keys. By numbering the foreign keys, you can avoid this confusion.

The foreign keys in the AuthorBook table participate in one-to-many relationships with the Authors table and the Books table. All relationships in a data model are indicated by the connecting lines, with three prongs used on the many end. In the case of this data model, three one-to-many relationships exist in all.

As you can see, a data model includes a number of elements that you can use to build your database. The model acts as a blueprint from which you can create tables, define columns, and establish relationships. The modeling method used here is only one type of method available to database designers, and each method uses its own approach for identifying elements in the database. The basic objects, as they're represented here, are fairly similar among the various methods, and ultimately, the goal of any data model should be to represent the final database design clearly and concisely, as it will be implemented in your RDBMS.

Regardless of which data modeling system you use, there are generally four steps that you should follow when developing a data model:

❑ Identifying the potential entities that will exist in the database

❑ Normalizing the data design to support data storage in the identified entities

❑ Identifying the relationships that exist between tables

❑ Refining the data model to ensure full normalization

Whenever you're developing a data model, you're generally following the business rules that have been defined for a particular project. Return once more to Figure 4-11 for an example of how a business rule might work. Suppose that, as part of the project development process, one of the business rules states that an author can write one or more books, a book can be written by one or more authors, and one or more authors can write one or more books. To ensure that you meet this business rule, your database design must correctly reflect the relationship between authors and books.

Although the process of developing business rules is beyond the scope of this book, you should still be aware that, at the root of any database design, there are requirements that specify the storage data needs that the database should support. For that reason, whenever you're designing a database, you should take into account the requirements defined by your business rules and be certain that your database design correctly incorporates these requirements. Now take a closer look at each step involved in the data modeling process.

Identifying Entities

Early in the design process, you want to identify the entities and attributes that ultimately create your data model. An *entity* is an object that represents a set of related data. For example, authors and books are each considered an entity. You can think of identifying entities as the first step in identifying the tables that will be created in your database. Each entity is associated with a set of attributes. An *attribute* is an object that describes the entity. An attribute can be thought of as a property of the entity. For example, an author's first name is an attribute of the author's entity. You can think of identifying attributes as the first step in identifying the columns that make up your tables.

The first step in developing a data model is to identify the objects (entities and attributes) that represent the type of data stored in your database. The purpose of this step is to name any types of information, categories, or actions that require the storage of data in the database. Initially, you don't need to be concerned with tables or columns or how to group data together. You want only to identify the type of data that you need to store.

Take a look at an example in order to illustrate how this works. Suppose that you're developing a database for a bookstore. You need to design that database based on the following business rules:

❑ The database must store information about the authors of the books sold by the bookstore. The information should include each author's first, middle, and last names, the author's year of birth, and if applicable, the author's year of death.

❑ The database must store information about the books sold by the bookstore. The information should include the title for each book, the year the book was copyrighted, the publisher who published the book, and the city and state in which the publisher is located.

❏ One author can write one or more books, one or more authors can write one book, and one or more authors can write one or more books.

❏ One publisher can publish one or more books, and one or more publishers can publish one book.

From this information, you should be able to create a list of objects that require data storage. For example, the author's first name is an example of one of the objects that exists in this scenario. The first step in creating a data model is to record each object as it appears in the business rules. Figure 4-12 provides an example of how you might list the objects described in these sample business rules.
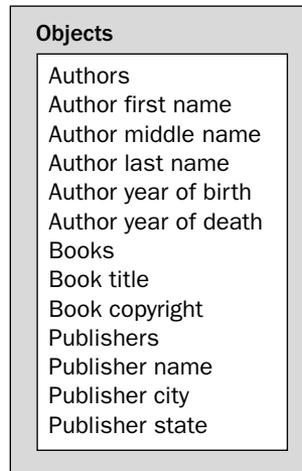
Objects

Authors
Author first name
Author middle name
Author last name
Author year of birth
Author year of death
Books
Book title
Book copyright
Publishers
Publisher name
Publisher city
Publisher state

Figure 4-12

As you can see, any object included in the business rules is listed here. From this information, you can begin to identify the entities and attributes and group them together into logical categories. For example, from the list of objects shown here, you might determine that there are three primary categories of information (entities): data related to authors, data related to books, and data related to publishers, as shown in Figure 4-13.
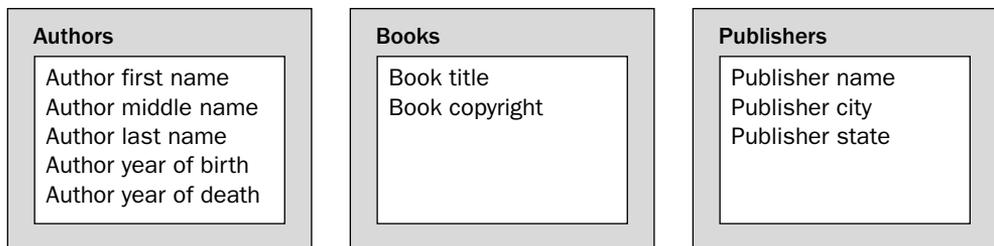
Authors

Author first name
Author middle name
Author last name
Author year of birth
Author year of death

Books

Book title
Book copyright

Publishers

Publisher name
Publisher city
Publisher state

Figure 4-13

By separating the object into logical categories, you provide yourself with a foundation from which you can begin to identify the tables in your database. At this point, the data model is far from complete, and you need to continue to modify it as you progress through the data modeling process. Categorizing the

entities in this way provides you with a foundation to begin normalizing the data structure, which is the next step in the data modeling process.

Normalizing Data

Once you define and categorize the primary entities and attributes in your data model, you can begin normalizing that structure, which results in the initial tables that make up your database. As you apply the rules of normalization to the structure, you identify tables, define the columns in those tables, assign data types to the columns, and establish a primary key.

Returning to Figure 4-13, you can see that there are three distinct entities: Authors, Books, and Publishers. The next step is to begin applying the rules of normalization to the entities. You've already started the process by organizing the objects into entities and their related attributes. As a result, you have a good start in identifying at least some of the tables needed in your database. Initially, your database includes the Authors, Books, and Publishers tables. From here, you can identify primary keys for each table, keeping in mind that all non-primary key columns must be dependent on the primary key columns and independent of each other.

For each table, you must determine whether to use existing columns for your primary key or to add one or more columns. For example, in the Authors table, you might consider using the author's first, middle, and last names as a composite primary key. The problem with this is that names can be duplicated, so they are seldom good candidates to use as primary keys. In addition, because you often refer to a primary key through a foreign key, you want to keep your primary keys as short as possible to minimize data redundancy and reduce storage needs. As a result, you're usually better off adding a column to the table that identifies each row and can be configured as the primary key, as shown in Figure 4-14.
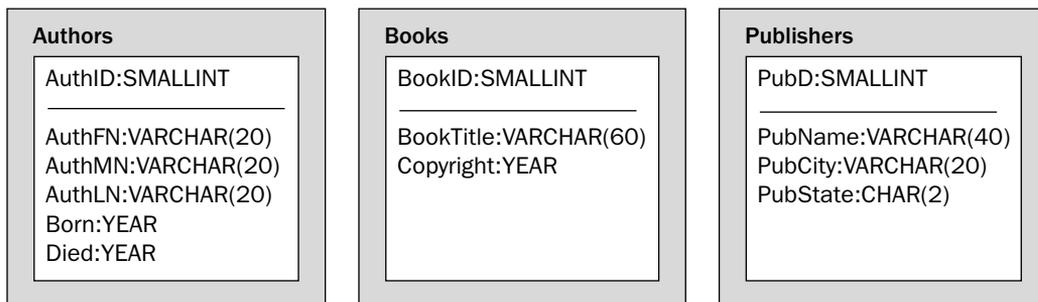
Authors	Books	Publishers
AuthID:SMALLINT	BookID:SMALLINT	PubD:SMALLINT
AuthFN:VARCHAR(20) AuthMN:VARCHAR(20) AuthLN:VARCHAR(20) Born:YEAR Died:YEAR	BookTitle:VARCHAR(60) Copyright:YEAR	PubName:VARCHAR(40) PubCity:VARCHAR(20) PubState:CHAR(2)

Figure 4-14

As you can see, the Authors table now has a primary key — AuthID — which is assigned the SMALLINT data type. (You learn about data types in greater detail in Chapter 5.) The other columns have been given names to represent the attributes, and they have also been assigned data types. When you begin assigning names to columns, you want to ensure that you use a system that is consistent throughout your database. For example, if you use mixed case to name your database objects, as is the case here, you should use that convention throughout. An alternative would be to use underscores to separate elements in a name, as in author_fname. Again, whichever method you use, you should remain consistent. In addition, regardless of the method you use, you must adhere to the following requirements to create objects in MySQL:

❏ Names can contain any alphanumeric characters that are included in the default character set.

❏ Names can include underscores (_) and dollar signs ($).

❏ Names can start with any acceptable character, including digits.

❏ Names *cannot* be made up entirely of digits.

❏ Names *cannot* include a period (.).

❏ Names *cannot* include an operating system's pathname separator, such as a backslash (\) or forward slash (/).

You should keep your naming conventions as simple as possible. In addition, object names should clearly reflect the meaning of that object. For example, you wouldn't want to assign arbitrary names and numbers to columns (such as Col1, Col2, Col3, etc.) because they provide no indication of what content that column might contain. Basically, you want to make sure that you use a logical naming structure when assigning names to the objects in your database so developers and administrators can easily identify their meaning.

Return to the bookstore database example to continue the normalization process. If you refer back to Figure 4-13, you can see that the second entity is Books, which contains book-related attributes. In this case, neither column would make a good candidate as a primary key. Even taken together they cannot ensure uniqueness. As a result, the best solution for a primary key is to add a column that uniquely identifies each row. The same is true for the Publishers table, which requires an additional column for a primary key. The additional column provides not only an easy way to ensure uniqueness in the table, but also an efficient way to reference a row in the table from another table, without having to repeat a lot of data.

If you return to Figure 4-14, notice that all three tables now have primary keys and column names with data types. In addition, all columns are dependent on their respective primary keys, and no columns are dependent on nonprimary key columns.

Identifying Relationships

The next step in creating a data model is to identify the relationships that exist between the tables. This step is usually a straightforward process of looking at each set of two tables, determining whether a relationship exists, and determining the type of relationship. Figure 4-15 demonstrates how this works.
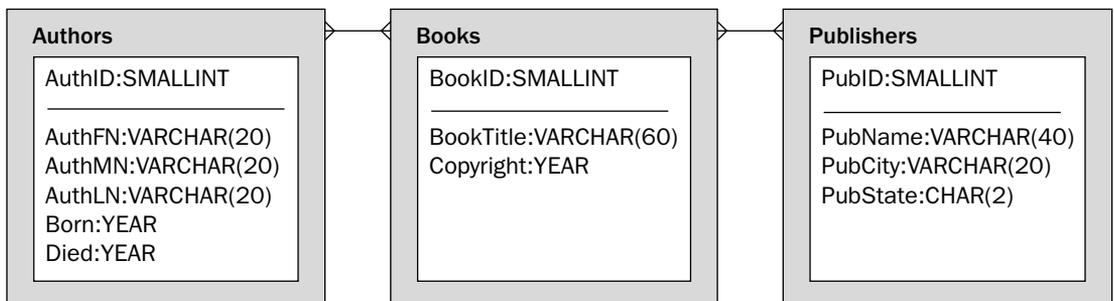

Authors

AuthID:SMALLINT
_____

AuthFN:VARCHAR(20)
AuthMN:VARCHAR(20)
AuthLN:VARCHAR(20)
Born:YEAR
Died:YEAR

Books

BookID:SMALLINT
_____

BookTitle:VARCHAR(60)
Copyright:YEAR

Publishers

PubID:SMALLINT
_____

PubName:VARCHAR(40)
PubCity:VARCHAR(20)
PubState:CHAR(2)

Figure 4-15

Because your data model contains three tables, three possible relationships can exist among them:

❑   Authors/Books

❑   Authors/Publishers

❑   Books/Publishers

First take a look at the Authors/Books table set. As you'll recall from the business rules outlined for this example, one author can write one or more books, one or more authors can write one book, and one or more authors can write one or more books. Not only does this imply that a relationship exists between authors and their books; it is a many-to-many relationship. As a result, you must add the appropriate line (with a three-pronged end) between the Authors and Books tables.

Moving on to the Authors/Publishers book set, you can find no relationship between these two tables. Nothing in the business rules implies a relationship between the two tables, and no entities suggest such a relationship. (And certainly in the real world, the question of any sort of relationship existing between authors and their publishers is one that is constantly open to debate.) If the business rules had included information that suggested a relationship between authors and publishers, it would have to be included in the data model.

The business rules, though, do imply a relationship between the Books and Publishers tables. According to the rules, one publisher can publish one or more books, and one or more publishers can publish a book. This clearly indicates that a many-to-many relationship exists between the Books table and the Publishers table, as shown in Figure 4-15. As a result, you must add the appropriate relationship line to your data model.

Refining the Data Model

Once you identify the relationships that exist between the tables, you can refine your data model as necessary. You might discover at this point that an entity isn't properly represented or that the data hasn't been fully normalized. This is a good time to review your business rules to ensure that your data model complies with all the specifications.

One way in which you're likely to need to refine your data model is to address any many-to-many relationships that you identified in the last stage of the data modeling process. In MySQL, and in most RDBMSs for that matter, a many-to-many relationship is implemented by adding a third table between the two tables in the relationship. The third table, referred to as a *junction table,* acts as a bridge between the two tables to support the many-to-many relationship. If you refer to Figure 4-11, you can see how the addition of the AuthorBook table to the model bridges the Authors and Books tables and how the addition of the BookPublisher table bridges the Books and Publishers tables. When you add a junction table, the many-to-many relationship is implemented as two one-to-many relationships. For example, a one-to-many relationship now exists between the Authors and AuthorBook table and the Books and AuthorBook table.

Junction tables of this nature usually include, at the very least, the primary key values from their respective tables. As a result, the AuthorBook table includes two foreign keys: one on the AuthID column and one on the BookID column. In addition, together these two columns form a primary key, which enforces the uniqueness of each row in that table. Through these two columns, one author can be associated with multiple books, multiple authors can be associated with one book, and multiple authors can be associated

with multiple books. The BookPublisher table works the same way as the AuthorBook table. The BookPublisher table includes two foreign keys: one on the PubID column and one on the BookID column. Together these two columns form the primary key. Once you add the junction table and indicate which columns are foreign keys, you must properly show the two one-to-many relationships that result from adding the table. Be sure to add the correct relationship lines to your data model between the Authors and AuthorBook table and the Books and AuthorBook table. Once again, refer to Figure 4-11, which shows the final data model.

After adding any necessary junction tables and relationship lines, you should review the data model once more to ensure that your changes didn't affect any of the other tables adversely or that more changes aren't necessary. Once you're satisfied that this stage is complete, your data model is ready for you to use to begin creating your database.

Designing the DVDRentals Database

Now that you have an idea of how to create a data model, it's time to try it out for yourself. The following four Try It Out sections walk you through the steps necessary to create your own data model. The model that you design in these exercises is used in Chapter 5 to create the DVDRentals database. All subsequent exercises through the rest of the book are based on that database, including the applications that you develop in Chapters 17–20.

The data model that you design here is created for a fictional store that rents DVDs to its customers. The database tracks the inventory of DVDs, provides information about the DVDs, records rental transactions, and stores the names of the store's customers and employees. To create the data model, you need to use the following business rules:

❏ The database stores information about the DVDs available for rent. For each DVD, the information includes the DVD name, the number of disks included with the set, the year the DVD was released, the movie type (for example, Action), the studio that owns the movie rights (such as Columbia Pictures), the movie's rating (PG and so forth), the DVD format (Widescreen, for example), and the availability status (Checked Out). Multiple copies of the same DVD are treated as individual products.

❏ The database should store the names of actors, directors, producers, executive producers, co-producers, assistant producers, screenwriters, and composers who participated in making the movies available to rent. The information includes the participants' full names and the role or roles that they played in making the movie.

❏ The database should include the full names of the customers who rent DVDs and the employees who work at the store. Customer records should be distinguishable from employee records.

❏ The database should include information about each DVD rental transaction. The information includes the customer who rented the DVD, the employee who ran the transaction, the DVD that was rented, the date of the rental, the date that the DVD is due back, and the date that the DVD is actually returned. Each DVD rental should be recorded as an individual transaction. Every transaction is part of exactly one order. One or more transactions are treated as a single order under the following conditions: (1) the transaction or transactions are for a single customer checking out one or more DVDs at the same time and (2) the transaction or transactions are being run by a single employee.

The business rules provided here are not meant to be an exhaustive listing of all the specifications that would be required to create a database, particularly if those specifications were to include a front-end application that would be interfacing with the database. The business rules shown here, however, are enough to get you started in creating your data model. Later in the book, as you add other elements to your database, the necessary business rules are provided.

To perform the exercises in the Try It Out sections, you need only a paper and pencil. If you have a draw program or a data modeling program, feel free to use that, although it isn't necessary.

Try It Out   Identifying Entities

As you learned earlier in the chapter, the first step in creating a data model is to identify the possible objects (entities and attributes) to include in that model. Refer to the preceding business rules to perform the following steps:

1.   Identify the possible objects listed in the first business rule. On a piece of paper, draw a rectangle for this business rule. Label the rectangle as "DVDs for rent." In the rectangle, list the potential objects found in that business rule.

2.   Repeat the process for the second business rule. Label the rectangle "Movie participants," and include any objects that you identify in that business rule. This one might be trickier because it covers the roles in the movie (such as producer or actor) and the participants' full names. Remember, though, that the actors, directors, producers, and so forth, are the participants.

3.   Create two rectangles for the third business rule, label one "Customers" and the other "Employees," and add the necessary objects.

4.   For the fourth business rule, create a rectangle and label it "Transactions/orders." Include in the rectangle each object that makes up a transaction. You should now have five rectangles with objects in each one. Your diagram should look similar to Figure 4-16.
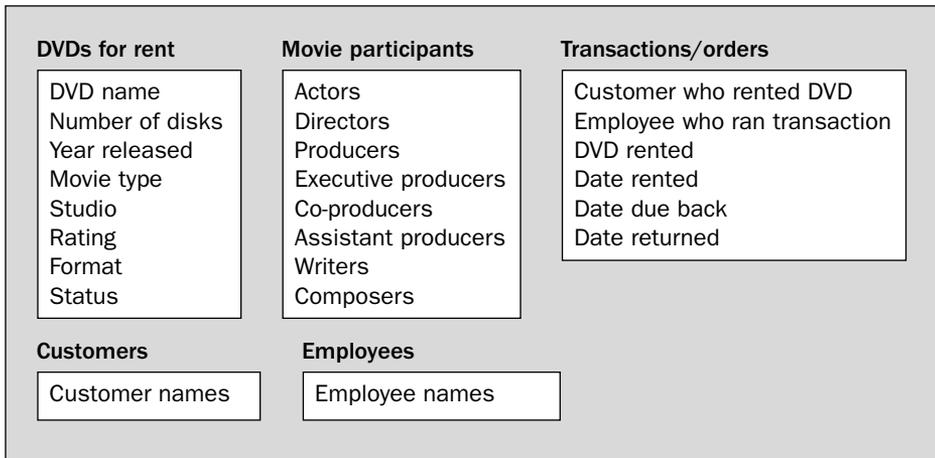

Figure 4-16

5. The next step is to organize the objects into entities. You've already done some of the work simply by listing the objects. For example, the DVDs for rent group of objects can be left together as you have them, although you can simplify the category name to "DVDs." The Transactions/ orders group of entities also provides you with a natural category. You probably want to separate the customers and employees into separate categories and specify that an attribute be created for each part of the name. You should separate employees and customers into two entities so that you can easily distinguish between the two. In addition, this approach allows you to treat each category as a separate entity, which could be a factor in securing the database.

Another reason for separating employees from customers is that, if this were actually a database that would be implemented in a production environment, there would probably be more attributes for each entity. For example, you would probably want to know the customers' home addresses and phone numbers, or you might want to include your employees' social security numbers. For the purposes of demonstrating how to model a database, the information you've included here should be enough.

6. The last objects you should categorize are those in the Movie participants group. This group is a little different from the others because you're working with a list of participant types, and for each of those types, you must include the full names of the participants. One way to approach this would be to create an entity for each participant type. Because different people can participate in different ways in one or more movies, though, you could be creating a situation in which your database includes a great deal of redundant data. Another approach would be to create one entity that includes the different parts of each participant's name and to include an attribute that identifies the role that the participant plays. Figure 4-17 demonstrates how this would work.
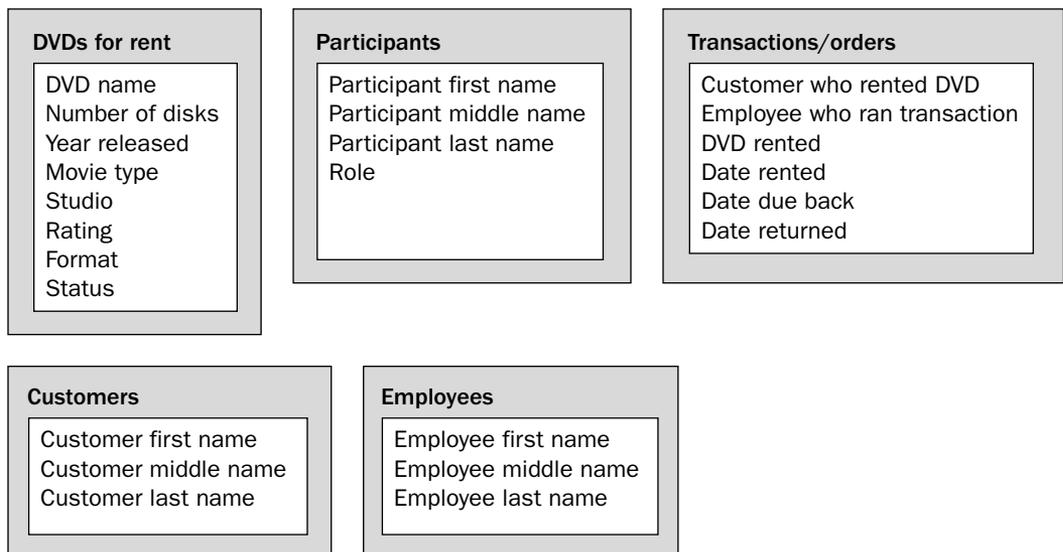
DVDs for rent	Participants	Transactions/orders
DVD name Number of disks Year released Movie type Studio Rating Format Status	Participant first name Participant middle name Participant last name Role	Customer who rented DVD Employee who ran transaction DVD rented Date rented Date due back Date returned

Customers	Employees
Customer first name Customer middle name Customer last name	Employee first name Employee middle name Employee last name

Figure 4-17

Notice that only one entity exists for participants and that it includes the name attributes and a role attribute. Now a participant's name needs to be listed only once in the database.

How It Works

In this exercise, you identified the potential entities and attributes that help form the foundation for your data model. First, you organized your objects according to the business rules in which they appeared. From there, you separated the objects into logical categories that grouped together entities and attributes in a meaningful way. At the same time, you divided the name attributes into first, middle, and last names, and you organized the participant data into a consolidated format that will help reduce data redundancy. From this organization, you can begin to normalize the data, which is the subject of the following Try It Out.

Try It Out   Normalizing Data

Once you identify and categorize the entities and attributes for your database, you can begin to normalize the data structure and define the tables and columns. Usually, the best way to do this is to work with one entity at a time. The following steps help you normalize the data structure that you created in the previous exercise:

1. Start with the DVDs entity. This entity already provides a solid foundation for a table because it clearly groups together similar attributes. The first step should be to define a primary key for the table. Because there can be multiple copies of the same DVD, and because different DVDs can share the same name, there is no column or columns that you can easily use as a primary key. As a result, the easiest solution is to add a column that uniquely identifies each row in the table. That way, each DVD has a unique ID.

2. At this point, you could leave the table as is and simply assign names and data types to the columns. (Note that, for the purposes of this exercise, the data types are provided for you.) When you have repeating values, such as status and format, an effective way to control the values that are permitted in a column is to create a table specifically for the values. That way you can add and modify permitted values without affecting the column definitions themselves. These types of tables are often referred to as *lookup tables.* Using lookup tables tends to decrease the amount of redundant data because the repeated value is often smaller than the actual value. For example, suppose you use an ID of s1 to identify the status value of Checked Out. Instead of repeating Checked Out for every row that this status applies to, the repeated value is s1, which requires less storage space and is easier to process.

   In the case of the DVDs s, you would probably want to create lookup tables for the movie type, studio, rating, format, and status attributes. Each of these tables would contain a primary key column and a column that describes the option available to the DVDs table. You would then add a column to the DVDs table that references the primary key in the lookup table. Also, be sure to name all your columns. This exercise uses a mixed case convention to name data objects. For example, a column named DVDName represents the DVD name attribute.

3. Next, take a look at the Participants entity. Notice that it includes an entry for the participant's role. This is another good candidate for a lookup table. Again, you should include a column that acts as the primary key and one that lists the role's name. Once you create a Roles table, you can add a column for the role ID to the Participants table. In addition, that table should include a primary key column that uniquely identifies each participant. Also, be sure to assign column names to the entities.

4. For both the Employees entity and the Customers entity, add a primary key column and assign a column name to the other attributes.

5. The next step is to separate the Transactions/orders entity into two tables because transactions are subsets of orders. For every order, there can be one or more transactions. In addition, each order must have one customer and one employee. As a result, you should create an Orders table.

The table should include a column that references the customer ID and a column that references the employee ID. The table should also include a primary key column.

Because you're tracking the customer and the employee at the order level, you do not need to include them at the transaction level. As a result, your Transactions table does not need to reference the employee or customer, but it does need to reference the order. In addition, it must reference the DVD that is being rented. Finally, you need to add a column to the Transactions table that acts as the primary key. Your data model should now be similar to the one found in Figure 4-18.
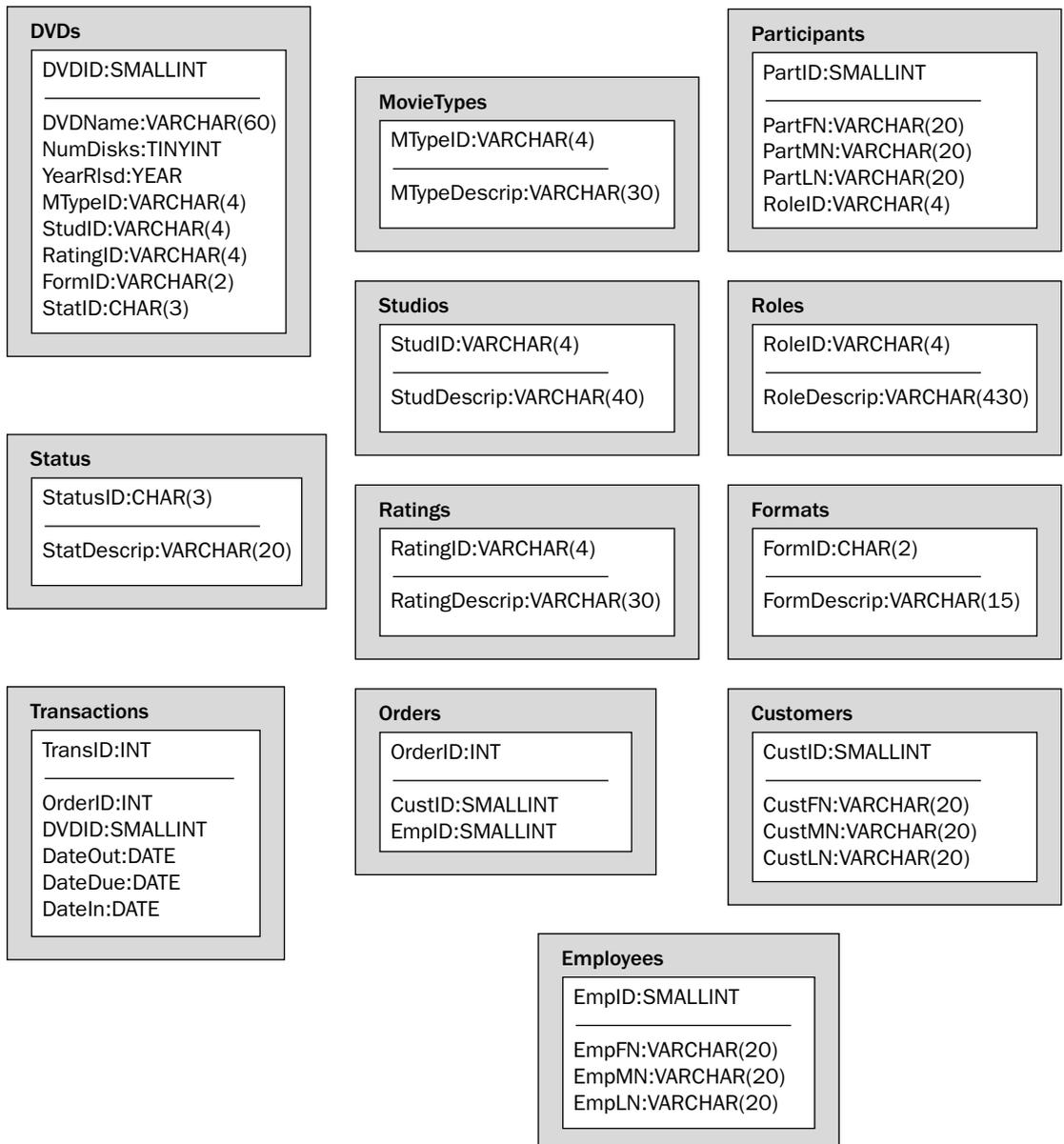
DVDs

DVDID:SMALLINT

DVDName:VARCHAR(60)
NumDisks:TINYINT
YearRlsd:YEAR
MTypeID:VARCHAR(4)
StudID:VARCHAR(4)
RatingID:VARCHAR(4)
FormID:VARCHAR(2)
StatID:CHAR(3)

MovieTypes

MTypeID:VARCHAR(4)

MTypeDescrip:VARCHAR(30)

Participants

PartID:SMALLINT

PartFN:VARCHAR(20)
PartMN:VARCHAR(20)
PartLN:VARCHAR(20)
RoleID:VARCHAR(4)

Studios

StudID:VARCHAR(4)

StudDescrip:VARCHAR(40)

Roles

RoleID:VARCHAR(4)

RoleDescrip:VARCHAR(430)

Status

StatusID:CHAR(3)

StatDescrip:VARCHAR(20)

Ratings

RatingID:VARCHAR(4)

RatingDescrip:VARCHAR(30)

Formats

FormID:CHAR(2)

FormDescrip:VARCHAR(15)

Transactions

TransID:INT

OrderID:INT
DVDID:SMALLINT
DateOut:DATE
DateDue:DATE
DateIn:DATE

Orders

OrderID:INT

CustID:SMALLINT
EmpID:SMALLINT

Customers

CustID:SMALLINT

CustFN:VARCHAR(20)
CustMN:VARCHAR(20)
CustLN:VARCHAR(20)

Employees

EmpID:SMALLINT

EmpFN:VARCHAR(20)
EmpMN:VARCHAR(20)
EmpLN:VARCHAR(20)

Figure 4-18

How It Works

As part of the data modeling process, you identified the tables included in the database and you listed the columns in each table. Normally, you would also assign data types to the columns, but for now, these were assigned for you. In determining how to set up the tables in your data model, you had to apply the rules of normalization to the entities and their attributes. For example, by adding a primary key to each table, you provided a way to uniquely identify each row so that the tables would conform to the normal forms. You also took steps to minimize redundant data by creating lookup tables that placed repeated values in separate tables. By the time you finished this exercise, the data structure should have been fairly normalized, and you're now ready to define the relationships between the tables.

Try It Out   Identifying Relationships

Now that you've identified the tables and their columns, you can begin defining the relationships between the tables. In most cases, it should be fairly obvious from the business rules and table structures where relationships exist and the type of relationships those are, although it's always a good idea to review each pair of tables to verify whether a relationship exists between those tables. For the purposes of brevity, this exercise focuses only on those sets of tables between which a relationship exists.

To identify the relationships between the tables in your data model, take the following steps:

1. Start with the DVDs table again, and compare it to other tables. You know from the previous exercise that lookup tables were created for several of the entities in this table. These include the MovieTypes, Studios, Ratings, Formats, and Status tables. Because of the nature of these tables and their origin with the DVDs table, you can assume that a one-to-many relationship exists between the DVDs table and the other five tables, with the DVDs table being the *many* side of the relationship. For example, the Studios table includes a list of the studios that own the rights to the movies. A studio can be associated with one or more movies; however, a movie can be associated with only one studio. As a result, a one-to-many relationship exists between those two tables.

    To indicate these relationships on your data model, use a line to connect each lookup table to the DVDs table, using the three-prong end on the DVDs side. In addition, add an FK followed by a number for each column in the DVDs table that references one of the lookup tables.

2. Now look at the DVDs and Participants tables. As would be expected with an inventory of movies, each movie can include one or more participants, each participant can participate in one or more movies, and multiple participants can participate in multiple movies. As a result, a many-to-many relationship exists between those two tables.

    Draw the relationship on your data model. Be sure to use the three-prong end on both sides.

3. Another table paired with the DVDs table is the Transactions table. Because each transaction must list the DVD that's being rented, you can assume that a relationship exists between these two tables. For every DVD there can be one or more transactions. For every transaction, there can be only one DVD. As a result, a one-to-many relationship exists between these tables, with the Transactions table on the *many* side of the relationship.

    Draw the relationship on your data model. Remember to add the FK reference to the referencing column in the Transactions table.

4. The next pair of tables to examine are the Participants and Roles tables. As you recall, each participant can play multiple roles, each role can be played by multiple participants, and multiple participants can play multiple roles, so a many-to-many relationship exists between these two tables.

Draw the relationship on your data model. Use the three-prong end on both sides.

5. Next look at the Transactions and Orders tables. Every order can contain one or more transactions, but each transaction can be part of only one order, so a one-to-many relationship exists between these two tables, with the Transactions table being the *many* side of the relationship.

Draw the relationship on your data model.

6. If you return to the Orders table, notice that it references the customer ID and employee ID, which means that a relationship exists between the Orders table and the Customers table as well as the Orders table and the Employees table. For every order, there can be only one employee and one customer, although each customer and employee can participate in multiple orders. As a result, the Orders table is on the *many* side of a one-to-many relationship with each of these other two tables.

For each of the relationships, you draw the correct relationship line on your data model, including the three-prong ends, where appropriate. Be sure to include the FK reference next to the referencing columns. Your data model should now be similar to the one shown in Figure 4-19.

How It Works

Once you normalized your data structure, you were able to determine the relationships that exist between the tables. As you discovered, there are a number of one-to-many relationships and two many-to-many relationships. In the model, every table participates in some type of relationship. The relationships provide a way to minimize data errors and data redundancy. As you may have noticed, no one-to-one relationships exist. These types of relationships are used more infrequently than the others, although they do have their benefits.

The final step in creating your data model is to refine your design. This includes not only making final adjustments to the model, but also addressing any many-to-many relationships.

Try It Out   Refining the Data Model

To finalize your data model, take the following steps:

1. Review the data model for inconsistencies or data that is not fully normalized. Once satisfied with the model, you can move on to the next step.

2. The next step is to address the many-to-many relationships. As you recall, these relationships are implemented through the use of a junction table to create one-to-many relationships. The Participants table is part of two different many-to-many relationships, which presents you with a unique situation.

To address these relationships, the best place to start is to define the situation: For each movie, there can be one or more participants who are playing one or more roles. In addition, each participant can play more than one role in one or more movies. As a result, movies are associated with participants, and participants are associated with roles. To address this situation, you can create one junction table that is related to all three tables: Roles, Participants, and DVDs. In each case, the junction table participates in a one-to-many relationship with the other tables, with the junction table on the *many* side of each relationship.
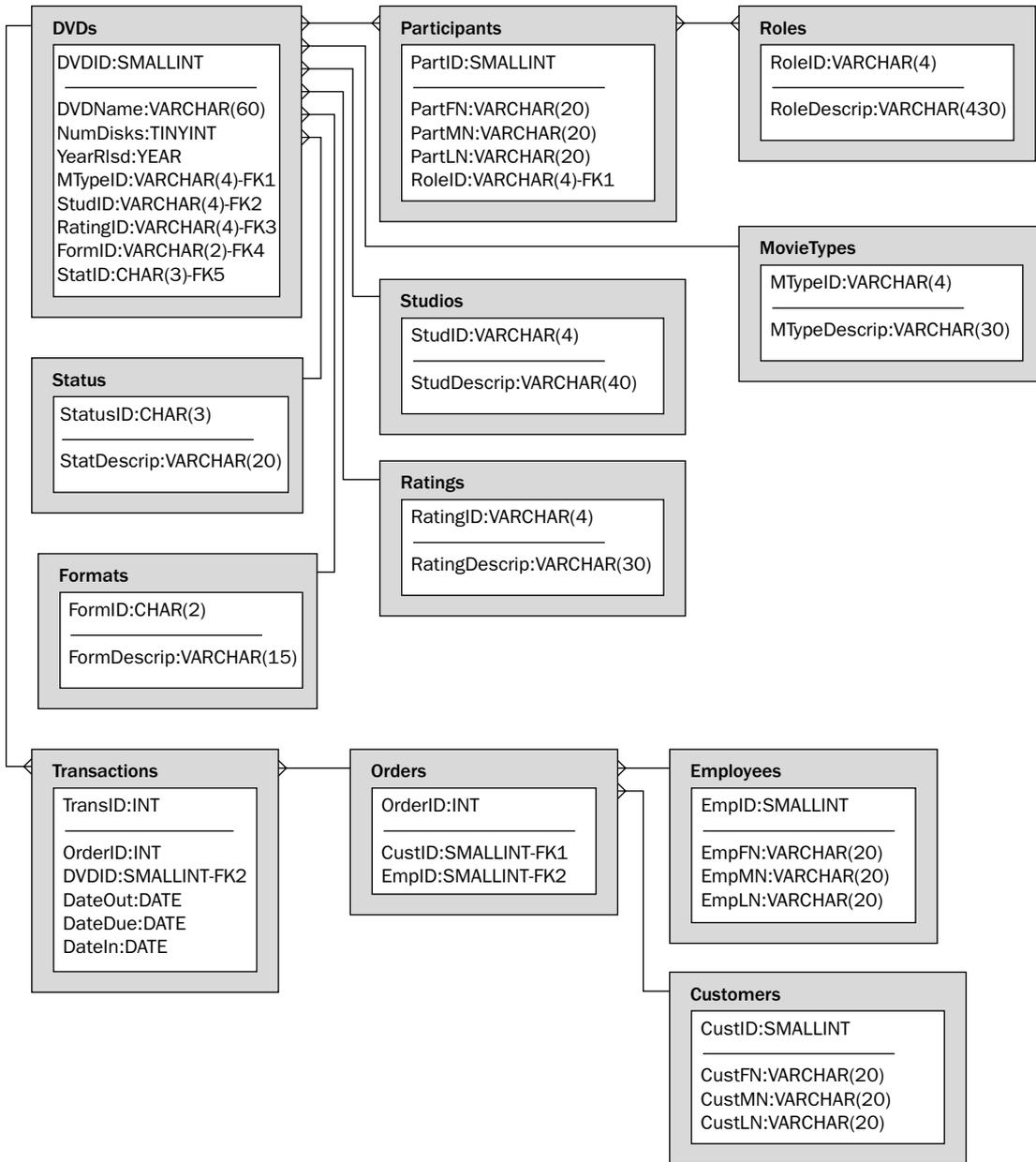
Figure 4-19

On your data model, add a junction table named DVDParticipant. Include a column that references the DVDs table, a column that references the Participants table, and a column that references the Roles table. Mark each column with an FK reference, followed by a number. Define

the primary key on all three columns. Also, be sure to mark the one-to-many relationship on your model, and remove the RoleID column from the Participants table. The column is no longer needed in the Participants table, because it is included in the DVDParticipant table.

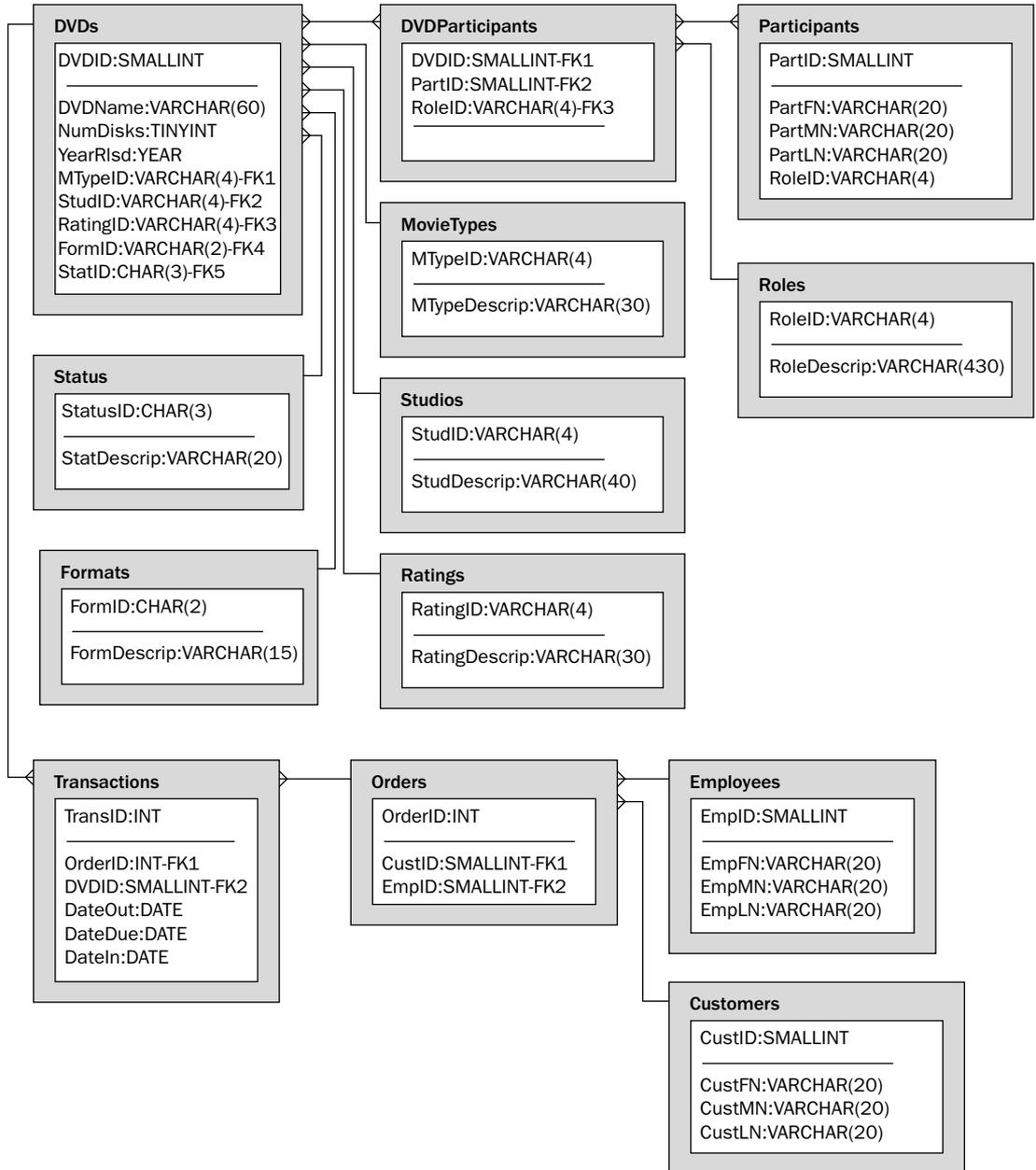Your data model should now look like the model illustrated in Figure 4-20.

Figure 4-20

How It Works

In this exercise, you took the final step in completing your data model by adding a junction table that bridged the many-to-many relationship between the DVDs and Participants table and the many-to-many relationship between the Participants and the Roles tables. The junction table — DVDParticipant — allows you to associate a DVD with a participant and a role. For example, one row in the junction table could show that Sydney Pollack was the producer of the movie *Out of Africa,* and another row in that table could show that Sydney Pollack was the director of that movie. By using the junction table, multiple participants are able to participate in multiple movies and do so in multiple roles. At the same time, the movies are listed only once (in the DVDs table), the participants are listed only once (in the Participants table), and roles are listed only once (in the Roles table).

When you have a table like DVDParticipant, which is on the many side of three different one-to-many relationships, you have what is referred to as a *ternary* relationship. Although ternary relationships are commonly implemented in a database, it's generally a good idea to work around them if possible. This sometimes means adding tables and creating more junction tables so that those junction tables are involved in no more than two one-to-many relationships. In cases like the DVDParticipant table, where there are no easy ways to work around the ternary relationship, this is a fairly efficient solution, particularly when the junction table is made up only of foreign key values.

Once you create the junction table, add the necessary FK references, draw the one-to-many relationships, and remove the RoleID column from the Participants table, you're ready to start building the DVDRentals database. Remember to hang on to your data model because you'll need to refer to it as you're developing your database.

Summary

As you worked your way through this chapter, you learned how to structure a relational database to support your data storage needs. The process of defining tables and their columns, ensuring that the data structure adheres to the rules of normalization, and identifying the relationships between tables allows you to create a data model that helps to ensure the integrity of the data being stored. In addition, a properly designed database helps to reduce redundant data and improve the reliability of that data. To allow you to create an effective data model, you were provided with the background information and skills necessary to perform the following tasks:

❑   Understand the relational model and the components of the model, including tables, columns, rows, and primary keys.

❑   Normalize data according to the first three normal forms.

❑   Define the relationships between tables, including one-to-one, one-to-many, and many-to-many relationships.

❑   Perform the steps necessary to create a data model, including identifying the potential entities to include in a database, normalizing the data design, identifying the relationships between tables, and refining your data model.

By creating a data model, you provided yourself with a blueprint for developing your database. From the model, you can create the necessary tables, include the appropriate column definitions in those tables, assign data types, define primary keys, and establish relationships between tables. From there,

you can create additional constraints in the database that meet any additional specifications outlined in your business requirements. Once you've created your database, you can add, retrieve, and manipulate data. Chapter 5 shows you how to assign data types and how to create and manage databases, tables, and indexes.

Exercises

The following questions are provided as a way for you to better acquaint yourself with the material covered in this chapter. Be sure to work through each exercise carefully. To view the answers to these questions, see Appendix A.

1. What are the components that make up a table in the relational model?

2. What requirements must a relation meet to be in compliance to the first normal form?

3. How does a one-to-many relationship differ from a many-to-many relationship?

4. You are creating a data model for a MySQL database. You identify the entities that you found in the business rules. You then group those entities into categories of related data. What step should you take next?

5. How are many-to-many relationships implemented in MySQL?