8

Designing Databases to Support Software Applications

The previous chapters showed how to gather user requirements, build a database model, and normalize the database to improve its performance and robustness. Those chapters showed how to look at the database from the customers' perspective, from the end user's perspective, and from a database normalization perspective, but there's one other point of view that you should consider before you open your database product and start slapping tables together: the programmer's.

You may not be responsible for writing a program to work with a database. The database may not ever directly interact with a program (although that's rare). In any case, the techniques that you would use to make a database easier for a program to use often apply to other situations. Learning how to help a database support software applications can make the database easier to use in general.

In this chapter you learn:

- Steps you can take to make the database more efficient in practical use.
- Methods for making validation easier in the user interface.
- Ways to easily manage non-searchable data.

This chapter describes several things that you can do to make the database more program-friendly.

A few of these ideas (such as multi-tier architecture) have been covered in earlier chapters. They are repeated in brief here to tie them together with other programming-related topics, but you should refer to the original chapters for more detailed information.

Plan Ahead

Any complicated task benefits from prior planning, and software development is no exception. It has been shown that the longer an error remains in a project the longer it takes to fix it. Database

design occurs very early in the development process, so mistakes made here can be very costly. A badly designed database provides the perfect proving ground for the expression, ''Act in haste, repent at leisure.'' Do your work up front or be prepared to spend a lot of extra time fixing mistakes.

Practically all later development depends directly or indirectly on the database design. The database design acts like a building's foundation. If you build a weak foundation, the building on top of it will be wobbly and unsound. The Leaning Tower of Pisa is a beautiful result built on a weak foundation, but it's the result of luck more than planning and people have spent hundreds of years trying to keep it from falling down. If you try to build on a wobbly foundation, you're more likely to end up with a pile of broken rubble than an interesting building.

After you get some experience with database design, it's very tempting to just start cranking out table and field definitions without any prior planning, but that's almost always a mistake. Don't immediately start building a database or even creating a relational object model. At least sketch out an ER diagram to better understand the entities that the database must represent before you start building.

Document Everything

Write everything down. This helps prevent disagreements about who promised what to whom. (''But you promised that the database could look up a customer's credit rating and Amazon password.'')

Good documentation also keeps everyone on the same wavelength. If you have done a good job of writing requirements, use cases, database models, design specifications, and all of the other paperwork that describes the system, the developers can scurry off into their own little burrows and start working on their parts of the system without fear of building components that won't work together.

You can make programmers' lives a lot easier if you specify table and field definitions in great detail. Write down the fields that belong in each table. Also write down each field's properties: name, data type, length, whether it can be null, string format (such as ''mm/dd/yyyy'' or ''###-####''), allowed ranges (1–100), default values, and other more complex constraints.

Programmers will need this information to figure out how to build the user interface and the code that sits behind it (and the middle tiers if you use a multi-tier architecture). Make sure the information is correct and complete at the start so the programmers don't need to make a bunch of changes later.

For example, suppose Price must be greater than $1.00. The programmers get started and build a whole slew of screens that assume Price is greater than $1.00. Now it turns out that you meant Price must be *at least* $1.00 not *greater than* $1.00. This is a trivial change to the design and to the database but the programmers will need to go fix the whole bunch of screens that contain the incorrect assumption. (Actually, good programming practices will minimize the problem, but you can't assume everyone is a top-notch developer.)

After you have all of this information, don't just put it in the database and assume that everyone can get the information from there. Believe it or not, some developers don't know how to use every conceivable type of database product (MySQL, SQL Server, Access, Informix, Oracle, DB2, Paradox, Sybase, PostgreSQL, FoxPro — there are hundreds) so making them dig this information out of the database can be a time-consuming hassle. Besides, writing it all up gives you something to show management to prove that you're making progress.

Consider Multi-Tier Architecture

A multi-tier architecture can help isolate the database and user interface development so programmers and database developers can work independently. This approach can also make a database more flexible and amenable to change. Unless you're a project architect, you probably can't decide to use this kind of architecture by yourself but you can make sure it is considered. See Chapter 6 for more details about multi-tier architectures.

Convert Domains into Tables

It's easy enough to validate a field against its domain by using check constraints. For example, suppose you know that the Addresses table's State field must always hold one of the values CA, OR, or WA. You can verify that a field contains one of those values with a field-level check constraint. In Access, you could set the State field's Validation Rule property to:

```
='CA' Or ='OR' Or ='WA'
```

Other databases use different syntax.

Although this is simple and it's easy for you to change, it's not easily visible to programmers building the application. That means they need to write those values into the code. Later if you change the list, the programmers need to change the code.

Even worse, someone needs to remember that the code needs to be changed! It's fairly common to change one part of an application and forget to make a corresponding change elsewhere. Those kinds of mistakes can lead to some bugs that are very hard bugs.

A better approach is to move the domain information into a new table. Create a States table and put the values CA, OR, and WA in it. Then make a foreign key constraint that requires the Addresses table's States field to allow only values that are in the States table. Programmers can query the database at run time to learn what values are allowed and can then do things such as making a combo box that allows only those choices. Now if you need to change the allowed values, you only need to update the States table's data and the program automatically picks up the change.

Wherever possible, convert database structure into data so everyone can read it easily.

Try It Out — Lookup Tables

Okay, this is really easy but it's important so bear with me. Suppose your Phoenician restaurant offers delivery service for customers with ZIP Codes between 02154 and 02159. In Access, you could validate the customer's Zip field with the following field-level check constraint:

```
>='02154' And <='02159'
```

Unfortunately that constraint is hidden from the programmers building the user interface. These checks may also not be in a form that's easy for the program to understand. Most programmers don't know how to open Access, read a field's check constraints, and parse an expression such as this one to figure out what it does.

How could you make this condition easier for programmers to discover and use at run time?

How It Works

Though reading and parsing check constraints is hard, it's fairly easy for a program to read the values from a table. The answer is to make a DeliveryZips table that lists the ZIP Codes in the delivery area:

Zip
02154
02155
02156
02157
02158
02159

This seems less elegant than the field-level check constraint but it's a lot easier for the program to understand.

Keep Tables Focused

If you keep each table well-focused, there will be fewer chances for misunderstandings. Different developers will have an easier time keeping each table's purpose straight so different parts of the application will use the data in the same (intended) way.

Modern object-oriented programming languages also use objects and classes that are very similar to the objects and classes used in semantic object modeling and that are similar to the entities and entity sets used by entity-relationship diagrams. If you do a good job of modeling, and keep object and table definitions well-focused, those models practically write the code by themselves. There are still plenty of other things for the programmers to do but at least they'll be able to make programming object models that closely resemble the database structure.

Use Three Kinds of Tables

One tip that can help you keep tables focused is to note that there are three basic kinds of tables. The first kind stores information about objects of a particular type. These hold the bulk of the application's data.

The second kind of table represents a link between two or more objects. For example, the association tables described in Chapter 5 represent a link between two types of objects.

Figure 8-1 shows an ER diagram to model employees and projects. Each employee can be assigned to multiple projects and each project can involve multiple employees. The EmployeeRoles table provides a

link between the other two object tables. It also stores additional information about the link. In this case, it stores the role that an employee played on each project.

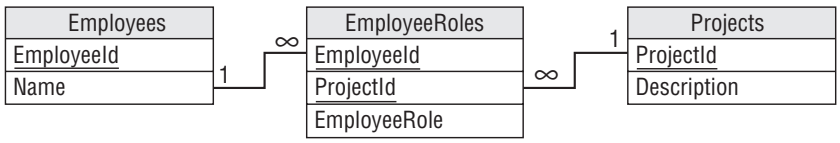

Figure 8-1

The third basic kind of table is a lookup table. These are tables created simply to use in foreign key constraints. For example, the States table described in the earlier section ''Convert Domains into Tables'' is a lookup table.

When you build a table, ask yourself whether it represents an object, a link, or a lookup. If you cannot decide or if the table represents more than one of those, the table's purpose may not be clearly defined.

Try It Out — Well-Focused Tables

Take a look at the following table of extra-terrestrial animals:

Animal	Size	Planet	PlanetaryMass
Hermaflamingo	Medium	Virgon 4	1.21
Shunkopotamus	Large	Dilbertopia	0.88
Mothalope	Medium	Xanth	0.01
Shunkopotamus	Large	Virgon 4	1.21
Platypus	Small	Australia	1.00

The table's primary key is the combination of Animal/Planet. (The PlanetaryMass field is measured in Earth masses.)

This isn't a very well-focused table.

1. What ideas is this table is trying to capture?
2. What types of ideas are these (object, link, or lookup)?
3. Suggest a better design that keeps the table's separate purposes separate.

How It Works

1. What ideas is this table is trying to capture?

   This table is trying to capture three different ideas: information about the animals (Animal and Size), information about planets (Planet and PlanetaryMass), and the associations between the

animals and planets. The fact that this table holds information about three different concepts is a sign that it is not well-focused.

(Also note this table is not in Second Normal Form because it has non-key fields that do not depend on the entire key. Recall that the primary key is Animal/Planet. The Size field depends on Animal but not Planet, and the PlanetaryMass field depends on Planet but not Animal.)

2. What types of ideas are these (object, link, or lookup)?

   Information about the animals and information about the planets represent objects. Information about the associations between animals and planets is a link.

3. Suggest a better design that keeps the table's separate purposes separate.

   The key is to move the three kinds of information into three different tables. Figure 8-2 shows one relational design that separates the three sets of information into three tables.

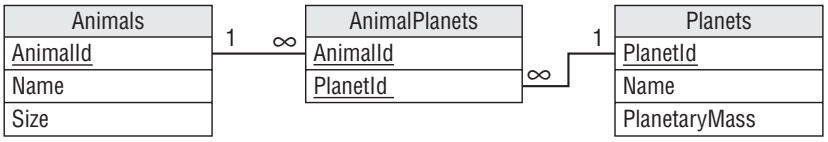

Figure 8-2

Figure 8-3 shows the tables holding their original data.

Animals

Size	Animal	AnimalId
Medium	Hermaflamingo	1
Large	Shunkopotamus	2
Medium	Mothalope	3
Small	Platypus	4

AnimalPlanets

AnimalId	PlanetId
1	101
2	102
3	103
2	101
4	104

Planets

PlanetId	HomePlanet	PlanetaryMass
101	Virgon 4	1.21
102	Dilbertopia	0.88
103	Xanth	0.01
104	Australia	1.00

Figure 8-3

Use Naming Conventions

Use consistent naming conventions when you name database objects. It doesn't matter too much what conventions you use as long as you use something consistent.

Some database developers prefix table names with tbl and field names with fld as in, ''The tblEmployees table contains the fldFirstName and fldLastName fields.'' For simple databases, I prefer to omit the prefixes because it's usually easy enough to tell which names represent objects (tables) and which represent attributes (fields).

Some developers also use a lnk prefix for tables that represent a link between objects as in, ''The lnkAnimalsPlanets table links tblAnimals and tblPlanets.'' These developers are also likely to use lu or lup as a prefix for lookup tables.

Some developers prefer to use UPPERCASE_LETTERS for table names and lowercase_letters for field names. Others use MixedCase for both.

Some prefer to make table names singular (the Employee table) and others prefer to make them plural (the Employees table).

As I said, it doesn't matter too much which of these conventions you use as long as you pick some conventions and stick to them.

However, there are three ''mandatory'' naming rules that you should follow or the other developers will snicker behind your back and openly mock you at parties.

First, don't use special characters in table names, field names, or anywhere else in database objects. For example, some databases (such as Access) allow you to include spaces in table and field names. For example, you can make a field named ''First Name.'' Those databases also provide some mechanism for making these weird names readable to the database. For example, in Access you need to use square brackets to surround a field with a name containing spaces as in ''[First Name].'' This produces hard-to-read expressions in check constraints and anywhere else you use the field. It also makes programmers using the field take similar annoying steps and that makes their code less readable, too.

Second, if two fields in different tables contain the same data, give them the same name. For example, suppose you use an ID field to link the Employees table to the EmployeePhones table. Don't call this linking field Id in the Employees table and EmpId in the EmployeePhones table. That's just asking for trouble. Call the field EmployeeId in both tables. (A corollary to this rule is that you cannot name an ID field something vague such as Id. It may make sense in the main table such as Employees but that name won't make sense in a related table such as EmployeePhones.)

The third mandatory naming rule is to use meaningful names. Don't abbreviate to the point of obscurity. It shouldn't take a team of National Security Agency cryptographers to decipher a table's field names. StudPrfCrs is much harder to read than StudentPreferredCourses. Don't be afraid to spell things out so everyone can understand them. (The exception here seems to be the military where everyone would understand ''SecInt visited NavSpecWarGru'' but saying ''the Secretary of the Interior visited the Naval Special War Group'' would brand you as an outsider.)

The section ''Poor Naming Standards'' in Chapter 10 has more to say about naming conventions and includes a few links that you can follow to learn about some specific standards that you can adopt if you like.

Allow Some Redundant Data

Chapter 7 explained that it is not always best to normalize a database as completely as possible. The higher forms of normalization usually spread data out into tables that are linked by their fields. When a program needs to display that data, it must reassemble all of that scattered data and that can take some extra time.

For example, if you allow customers to have any number of phone numbers, email addresses, postal addresses, and contacts, then what seems to the user like a simple customer record is actually spread across the Customers, CustomerPhones, CustomerEmails, CustomerAddresses, and CustomerContacts tables.

In some cases, it may be better to restrict the database's flexibility somewhat to gain speed and simplicity. For example, if you allow the customers to have only two phone numbers, one email address, and one contact, you cut the number of tables that make up the customer's information from five to two. The database won't be as infinitely flexible and it won't be quite as completely normalized, but it will be easier to use.

Usually it's also best not to store the same data in multiple ways because that can lead to modification anomalies. For example, you don't really need a Balance field in a customer's record if you can recalculate the balance by adding up all of the customer's credits and debits.

However, suppose you're running an Internet service that allows customers to download music so a typical customer makes dozens or even hundreds of purchases a month. After a year or two, adding up all of a customer's credits and debits could be time consuming. In this case, you might be better off adding a Balance field to the customer's record and exercising a little extra caution when updating the account.

Don't Squeeze in Everything

Just because you're using a database doesn't mean every piece of data that the system uses must be squeezed in there somewhere. Databases provide tools for storing and retrieving some strange pieces of data such as audio, video, images — just about anything you can cram into a computer file. That doesn't mean you should go crazy and store every file on your computer within the database.

For example, suppose an application must locate and play thousands of audio files. You could store all of them in the database or you could place the files in a directory tree somewhere and then store the locations of those files in the database. That makes the database simpler, smaller, and possibly more efficient because it doesn't need to store all of those files. It also makes it a lot easier to update the files. Instead of loading a new file into the database, you can simply replace the file on the disk.

This technique can also be useful for managing large amounts of shared data such as Web pages. You don't need to copy Wikipedia pages into your database (in fact, it would probably be a copyright violation). Instead you can store the URLs pointing to the pages you need. In this case you give up control of the data but you also don't have to store and maintain it yourself. If the data is updated, you'll see the new data the next time you visit a URL.

There are only a couple of drawbacks to this technique. First, you lose the ability to search inside any data that is not stored inside the database. I don't know of any databases that let you search inside video, audio, or jpeg data, however, so you probably shouldn't lose much sleep over giving up an ability that you don't have anyway. I wouldn't move textual data outside the database in this way, however, unless you're sure you'll never want to search inside it.

Second, you give up some of the security provided by the database. For example, you lose the database's record-locking features so you may have trouble allowing multiple concurrent users to update the data.

Try It Out First Normal Form

Suppose you're building a database of amusing commercials (see `www.veryfunnyads.com` and `giesbers.net/video` for some good ones). The Commercials table includes the fields Name, Product, Description, Length, Video (the commercial), and Still (a representative frame to remind you what the commercial is about).

Figure out which of these fields should remain in the database and which might be moved outside into the file system:

1. To figure out which fields should remain in the database, identify those that you might want to search. Include fields with simple values (such as numbers and short strings) that are easy to store in a database but that would not make a very big file on the disk.
2. To figure out which fields might be moved outside of the database into the file system, identify the fields that contain large chunks of non-searchable data.

How It Works

1. In this database, you might want to search the Name, Product, Description, and Length fields.
2. You cannot search the Video or Still fields whether you want to or not so you might as well move them into the file system. The Video field will contain particularly large amounts of data so moving it outside of the database might even make the database more efficient.

Summary

Though the focus of this book and your database design efforts is on databases, a database rarely lives in total isolation. Usually someone writes a program to interact with it. Often the database is just a backend for a complicated user interface.

To get the most out of your database, you need to consider it in its environment. In particular, you should think about the applications and programmers who will interact with it. Often a few relatively small changes to the design can make life easier for everyone who works with the database.

In this chapter you learned to:

- Plan ahead and document everything.
- Convert domains into tables to help user interface programmers and to make maintaining domain information easier.
- Keep tables well-focused and make each perform a single task.
- Use some redundancy and denormalized tables to improve performance.

The last several chapters dealt with database design techniques and considerations. Those chapters explained general techniques for building a data model and then modifying it to make it more efficient.

The next chapter switches from general discussion to more specific techniques. It summarizes some of the methods described in the previous chapters and explains some common database design patterns that you may find useful in providing specific data features.

Before you move on to Chapter 9, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. Suppose you sell ocean cruises. Customers first decide what kind of Ship they want to travel on: Luxury Liner, Schooner, or Tuna Boat. Depending on that choice, they may select different classes of cabins. Luxury Liners provide 1st through 5th class. Schooners have 1st and 2nd class (basically a single or a double), and Tuna Boats have a single class (which they playfully call 1st Class) where you share a single large bunkroom with the rest of the crew.

   You could validate the Trips record's Ship and Cabin fields by using a table-level check constraint but, because you're a team player, you would rather build a foreign key constraint so the user interface can read the allowed values from the tables.

   Build such a table and display its data. Explain how this table will be used in the foreign key constraint.

2. Figure 8-4 shows a relational diagram showing the relationships between students, the classes they are taking, and the departments that hold the classes. For each table in this diagram, tell which of the three types of table it is: object, link, or lookup.

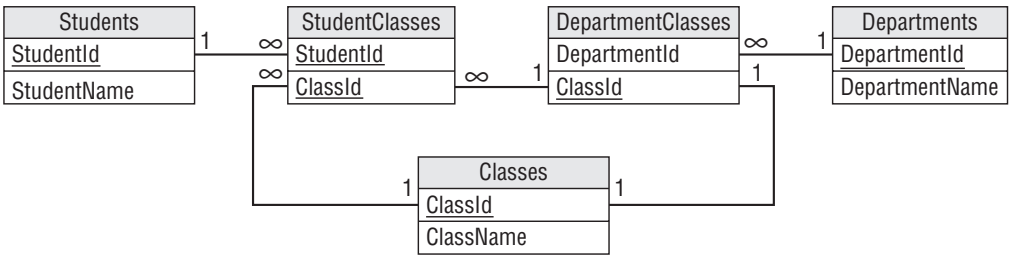

Figure 8-4

3. The following table stores information about checkers matches. Explain why it lacks focus and how you would fix it.

Player1	Player1Rank	Player2	Player2Rank	MatchTime
Smith	10	Jones	3	1:00
Marks	9	Lars	4	1:00
Aft	8	Cook	5	2:00
Mauren	7	Juno	6	2:00

4. Assume you have a large database that tracks how closely airplanes are to their scheduled departure and landing times. It tracks these values by plane (which is associated with a particular airline) and airport. It also records the weather at the starting and landing airports.

   Which of the following values should you store in a redundant variable and which should you calculate as needed?

   a. Average minutes late for an airline at a particular airport.
   b. Average minutes late for all airlines at a particular airport.
   c. Average minutes late for an airline across the country.
   d. Average minutes late for all airlines across the entire country.

   Assume that you need these numbers quickly several times per day.

9

Common Design Patterns

The previous chapters described general techniques for building database designs. For example, Chapter 5 explained how to build semantic object models and entity-relationship diagrams for a database, and how to convert those models into relational designs. Chapter 7 explained how to transform those designs to normalize the database.

This chapter takes a different approach. It focuses on data design scenarios and describes methods for building them in a relational model.

In this chapter you learn techniques for:

- Providing different kinds of associations between objects.
- Storing data hierarchies and networks.
- Handling time-related data.
- Logging user actions.

This chapter does not provide designs for specific situations such as order tracking or employee payroll. Appendix B, ''Sample Database Designs,'' contains those sorts of examples.

This chapter focuses on a more detailed level to give you the techniques you need to build the pieces that make up a design. You can use these techniques as the beginning of a database design toolbox that you can apply to your problems.

The following sections group these patterns into three broad categories: associations, temporal data, and logging and locking.

Associations

Association patterns represent relationships among various data objects. For example, an association can represent the relationship between a rugby team and its opponents during matches.

The following sections describe different kinds of associations.

Many-to-Many Associations

It's easy to represent a many-to-many association in an ER diagram. For example, a Student can be enrolled in many Courses and a Course includes many Students, so there is a many-to-many relationship between Students and Courses. Figure 9-1 shows an ER diagram modeling this situation.

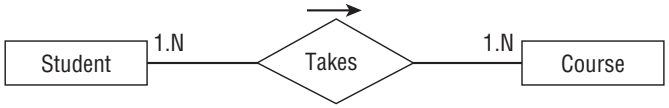

Figure 9-1

Unfortunately relational databases cannot handle many-to-many relationships directly. To build this kind of relationship in a relational database, you need to add an association table to represent the relationship between students and courses. Simply create a table called StudentCourses and give it fields StudentId and CourseId. Figure 9-2 shows this structure.

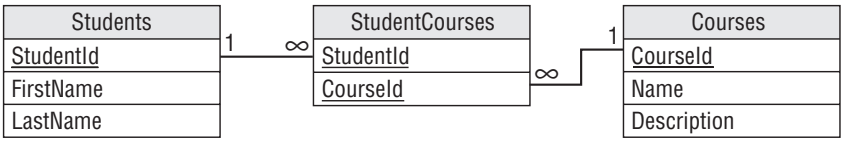

Figure 9-2

To list all of the courses for a particular student, find the StudentCourses records with that Student Id. Then use each of those records' CourseId values to find the corresponding Courses records.

To list all of the students enrolled in a particular course, find the StudentCourses records with that CourseId. Then use each of those records' StudentId values to find the corresponding Students records.

Multiple Many-to-Many Associations

Sometimes a many-to-many relationship contains extra associated data. For example, the previous section explained how to track students and their current course enrollments. Suppose you also want to track student enrollments over time. In other words, you want to know each student's enrollments for each year and semester. In this case, you really need to make multiple many-to-many associations between students and courses. You need whole sets of these associations to handle each school semester.

Fortunately this requires only a small change to the previous solution. The StudentCourses table shown in Figure 9-2 can already represent the relationship of students to courses. The only thing missing is a way to add more records to this table to store information for different years and semesters.

The solution is to add Year and Semester fields to the StudentCourses table. Figure 9-3 shows the new model.

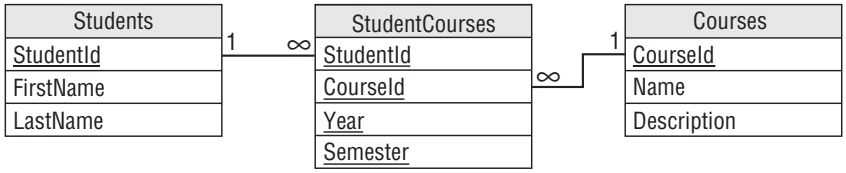

Figure 9-3

Now the StudentCourses table can store multiple sets of records representing different years and semesters.

If you need to store extra information about each semester, you could make a new Semesters table to hold that information. Then you could add the Year and Semester fields to this new table and use them as a foreign key in the StudentCourses table.

Try It Out — Many-to-Many Relations

Suppose you're coordinating a week-long tour called ''Junk Yards of the Napa Valley.'' Each day, the participants can sign up for several tours of different junk yards. They can also sign up for dinner at a fine restaurant or winery.

Build a relational model to record this information.

1. Build Participants, Tours, and Restaurants tables.
2. Study the relationships between Participants and Tours, and between Participants and Restaurants. Determine whether they are many-to-many or some other kind of relationship.
3. Build a relationship table to represent each many-to-many relationship. Be sure to include enough fields to distinguish among similar combinations of the involved tables. (For example, Bill really liked the trip to Annette's Scrap and Salvage on Tuesday so he took that tour again on Thursday.) Your model needs a ParticipantTours table and a ParticipantRestaurants table. To distinguish among repeats such as Bill's, add a Date field to each table.

How It Works

1. There's no real trick in Step 1. Just be sure to give each table an ID field so it's easy to refer to its records.
2. To understand Step 2, remember that each participant can go on many tours and each tour can have many participants so the Participants/Tours relationship is many-to-many. During the week, each participant can eat at several restaurants and each restaurant can feed many participants, so the Participants/Restaurants relationship is also many-to-many.
3. To model the two many-to-many relationships, the model needs a ParticipantTours table and a ParticipantRestaurants table. To distinguish among repeats (a customer takes the same tour twice or visits the same restaurant twice), add a Date field to each table.

Figure 9-4 shows the final relational model.

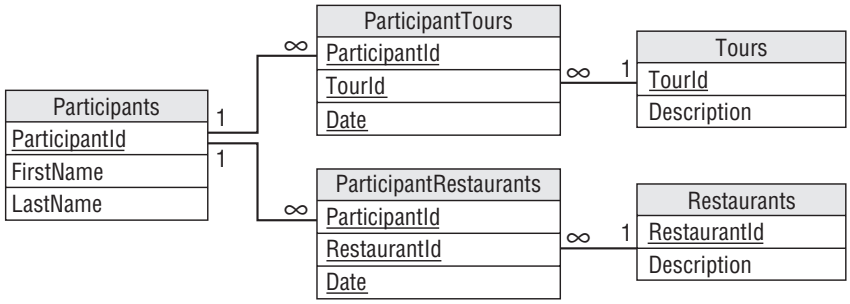

Figure 9-4

Multiple-Object Associations

A multiple-object association is one where many different kinds of objects are collectively associated to each other. For example, making a movie requires a whole horde of people including a director, a bunch of actors, and a huge number of crew members. You could model the situation with the ER diagram shown in Figure 9-5.

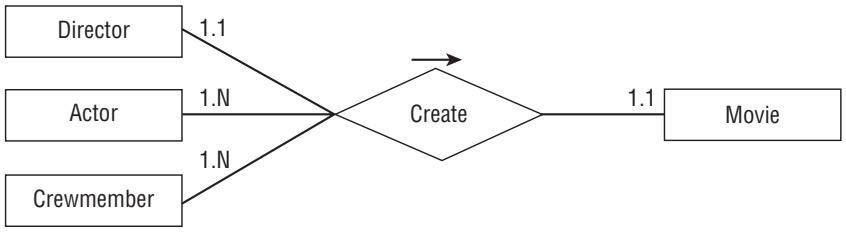

Figure 9-5

If this collection of people always worked as a team, this situation would be easy to implement in a relational model. You would assign all of the people a TeamId and then build a Movies table with a TeamId field to tell who worked on that movie.

Unfortunately, this idea doesn't quite work because all of these people can work on any number of movies in any combination.

You can solve this problem by thinking of the complex multi-object relationship as a combination of simpler relationships. In this case, you can model the situation as a one-to-one Director/Movie relationship, a many-to-many Actor/Movie relationship, and a many-to-many Crewmember/Movie relationship.

Figure 9-6 shows the new ER diagram.

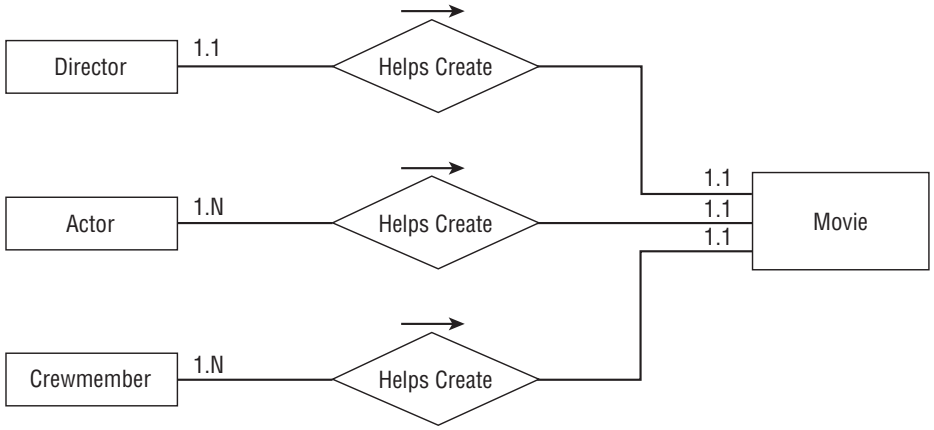

Figure 9-6

You can convert this simpler diagram into a relational model as shown in Figure 9-7.

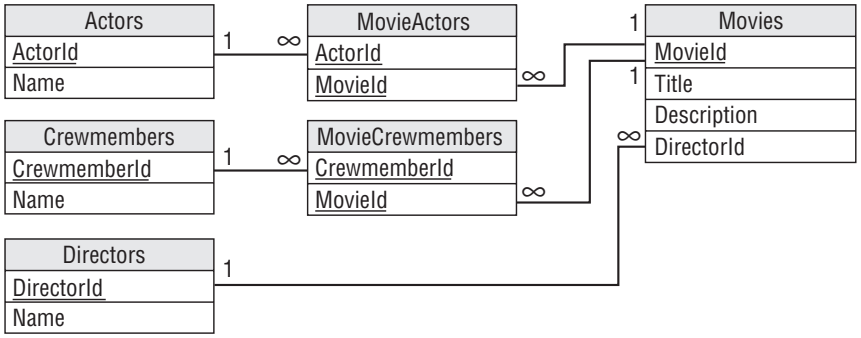

Figure 9-7

Notice that this model uses two association tables to represent the two many-to-many relationships. The relationship between Directors and Movies doesn't require an association table because this is a simpler one-to-one relationship.

Try It Out — Building Multiple-Object Associations

Consider another aspect of the "Junk Yards of the Napa Valley" tours. You have multiple tour guides and multiple vehicles. A Trip represents a specific instance of a tour by a guide, vehicle, and a group of participants.

Build a relational model to hold this data.

1. Build Guides, Vehicles, Tours, Participants, and Trips tables.
2. Study the relationships between Trips and Guides, Vehicles, Tours, and Participants. Determine whether they are many-to-many or some other kind of relationship.
3. Build an ER diagram to show these relationships.
4. Build a relationship table to represent each many-to-many relationship.
5. Draw the relational model.

How It Works

1. There's no real trick in this. Just be sure to give each table an ID field so it's easy to refer to its records.
2. Each guide can lead several trips but each trip has a single guide, so Guides/Trips is a one-to-many relationship.

   Each vehicle can go on many trips but each trip has a single vehicle, so Vehicles/Trips is a one-to-many relationship.

   A tour represents a destination. A destination can be the target of many trips but each trip visits only one destination, so Tours/Trips is a one-to-many relationship.

   Finally, each participant can go on many trips and each trip can have many participants, so Participants/Trips is a many-to-many relationship.
3. Figure 9-8 shows these relationships in an ER diagram.

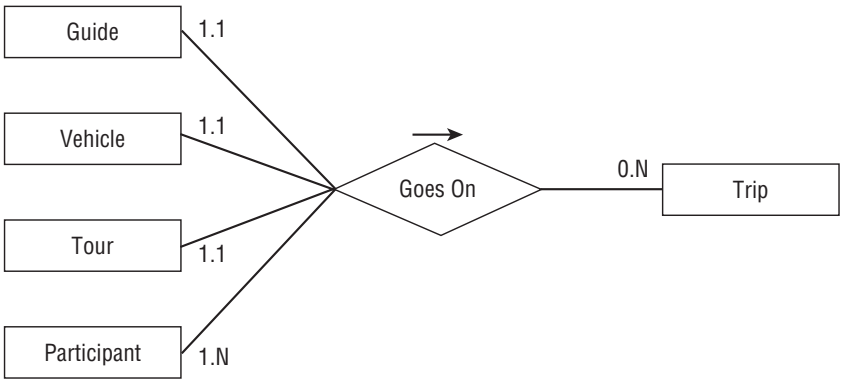

Figure 9-8

4. This model has only one many-to-many relationship: Participants/Trips. To handle it, the model needs a ParticipantsTrips table.

   Figure 9-9 shows the final relational model.

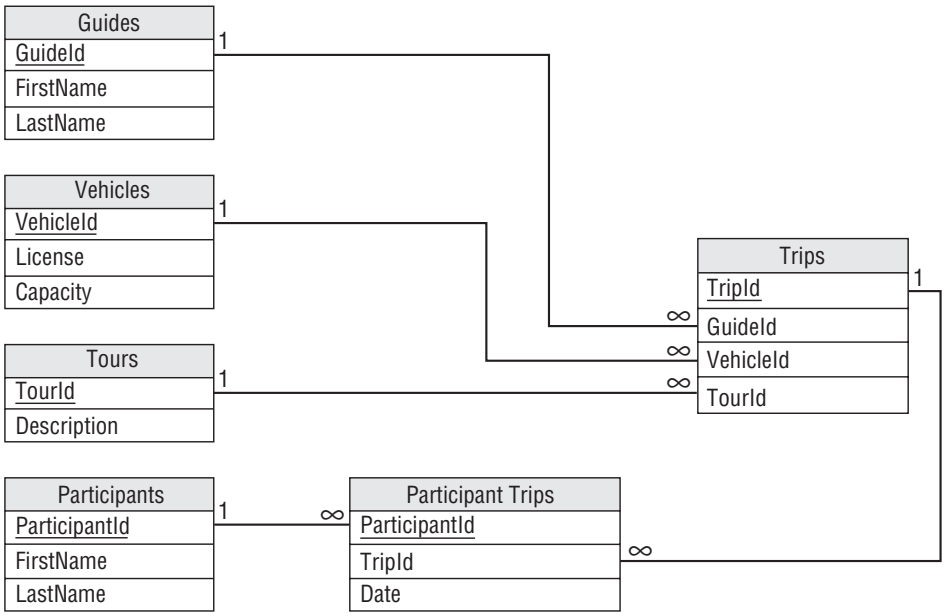

Figure 9-9

Repeated Attribute Associations

Some entities have multiple fields that represent either the same kind of data or a very similar kind of data. For example, it is common for orders and other documents to allow you to specify a daytime phone number and an evening phone number. Other contact-related records may allow you to specify even more phone numbers for such things as cell phone, FAX, pager, and others.

Figure 9-10 shows a semantic object model for a `PERSON` class that allows any number of `Phone` attributes.

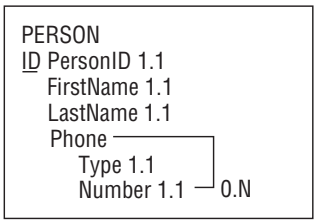

Figure 9-10

To allow any number of a repeated attributes in a relational model, build a new table to contain the repeated values. Use the original table's primary key to link the new records back to the original table.

Figure 9-11 shows how to do this for the PERSON class shown in Figure 9-10.

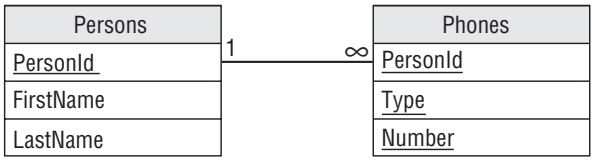

Figure 9-11

Because the Phones table's primary key includes all of the table's fields, the combination of PersonId/Type/Number must be unique. That means a particular person can only use a phone number for a particular purpose once. That makes sense. It would be silly to list the same phone number as a work number twice for the same person. However, a person could have the same number for multiple purposes (daytime and evening number are the same cell phone) or have multiple phone numbers for the same purpose (office and receptionist numbers for work phone).

You can use the primary keys and other keys to enforce other kinds of uniqueness. For example, to prevent someone from using the same number for different purposes, make PersonId/Number a unique key. To prevent someone from providing more than one number for the same purpose (for example, two cell phone numbers), make PersonId/Type a unique key.

For another example, suppose you want to add multiple email addresses to the Persons table. Allow each person to have any number of phone numbers and email addresses of any type, but don't allow duplicate phone numbers or email addresses. (For example, you cannot use the same phone number for Home and Work numbers.)

Just as you created a Phones table, you would create an Emails table with Type and Address fields, plus a PersonId field to link it back to the Persons table. To prevent an email address from being duplicated for a particular person, include those fields in the table's primary key. Figure 9-12 shows the new relational model.

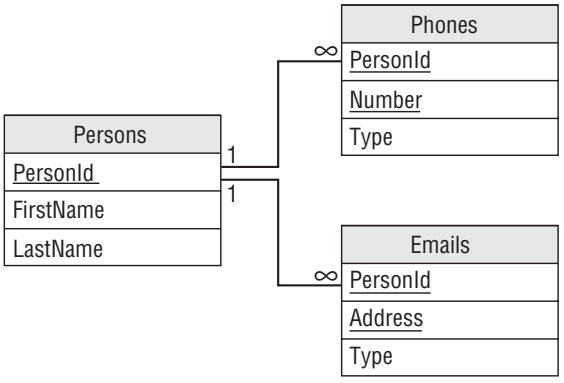

Figure 9-12

Reflexive Associations

A *relflexive* or *recursive* association is one in which an object refers to an object of the same class. You can use recursive associations to model a variety of different situations ranging from simple one-to-one relationships to complicated networks of association.

The following sections describe different kinds of reflexive associations.

One-to-One Reflexive Associations

As you can probably guess, in a one-to-one reflexive association an object refers to another single object of the same class. For example, consider the `Person` class's Spouse field. A Person can be married to exactly one other person (at least in much of the world) so this is a one-to-one relationship. Figure 9-13 shows an ER diagram representing this relationship.

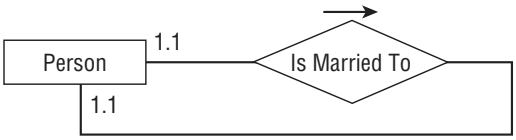

Figure 9-13

Figure 9-14 shows a relational model for this relationship.

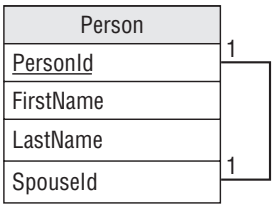

Figure 9-14

Unfortunately this design does not require that two spouses be married to each other. For example, Ann could be married to Bob and Bob could be married to Cindy. That might make an interesting television show, but it would make a confusing database.

Another approach would be to create a Marriage table to represent a marriage. That table would give the IDs of the spouses. Figure 9-15 shows this design.

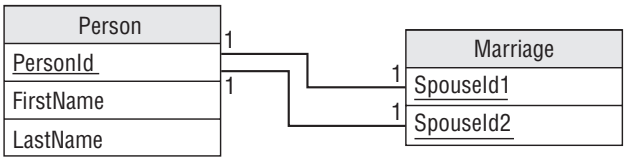

Figure 9-15

In this design the Person table refers to itself indirectly.

For another example, suppose you're making a database that tracks competitive rock-paper-scissors matches (see `http://www.usarps.com`). You need to associate multiple competitors with each other to show who faced off in the big arena. You also want to record who won and what the winning moves were.

You would start by making a Competitors table with fields Name and CompetitorId.

Next you would make a CompetitorMatches table to link Competitors. This table would contain CompetitorId1 and CompetitorId2 fields, and corresponding FinalMove1 and FinalMove2 fields to record the contestants' final moves. To distinguish among different matches between the same two competitors, the table would also include Date and Time fields.

Figure 9-16 shows the relational model.

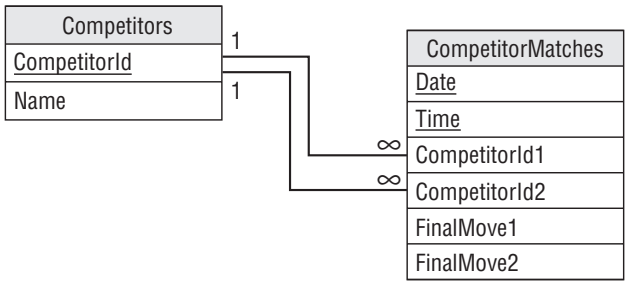

Figure 9-16

One-to-Many Reflexive Associations

Typically employees have managers. Each employee is managed by one manager and a manager can manage any number of employees, so there is a one-to-many relationship between managers and employees.

But a manager is just another employee, so this actually defines a one-to-many relationship between employees and employees. Figure 9-17 shows an ER diagram for this situation.

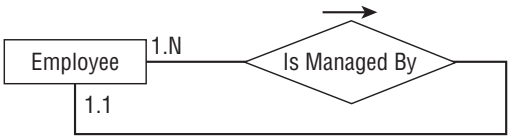

Figure 9-17

Figure 9-18 shows a relational model that handles this situation.

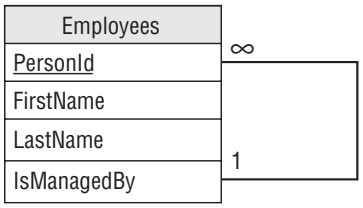

Figure 9-18

Hierarchical Data

Hierarchical data takes the form of tree-like structures. Every object in the hierarchy has a ''parent'' object of the same type. For example, a corporate organizational chart is a hierarchical data structure that shows which employee reports to which other employee. Figure 9-19 shows the org chart for a fictional company.

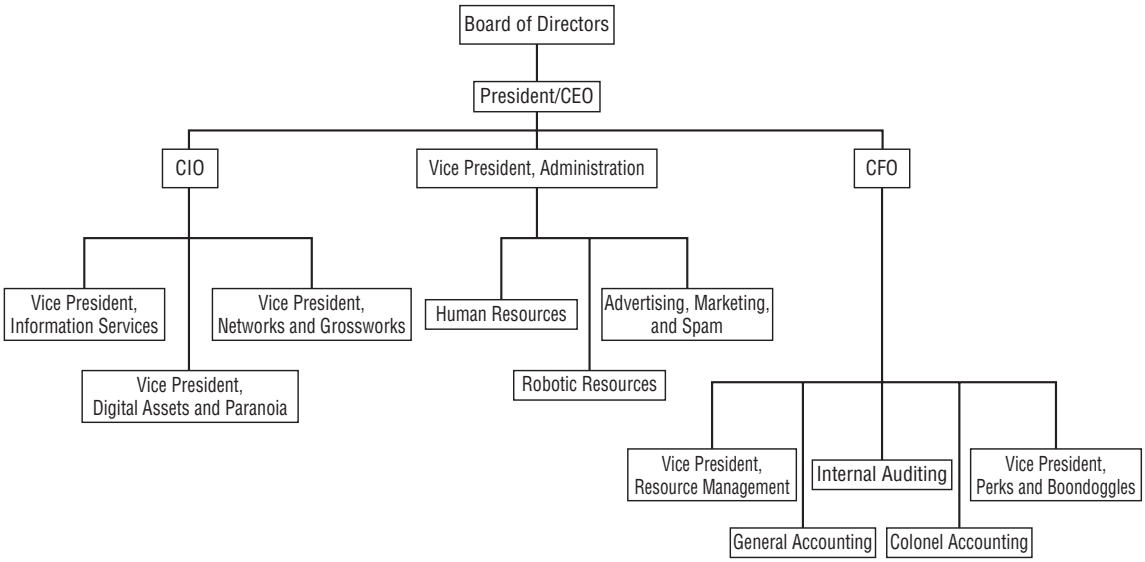

Figure 9-19

Hierarchical data actually is an instance of a one-to-many reflexive association as described in the previous section. Generally people think of the ''Is Managed By'' relationship as being relatively flat, so managers supervise front-line employees but no one needs to manage the managers. In that case, the hierarchy is very short.

An org chart represents the infinitesimally different concept of ''Reports To.'' I guess this is more palatable to managers who don't mind reporting to someone even if they don't need help managing their own work. (Although I've known a few managers who could have used some serious help in that respect.)

The ''Reports To'' hierarchy may be much deeper (physically, not necessarily intellectually) than the ''Manages'' hierarchy but you can still model it in the same way. Figure 9-20 shows an `Employee` class that can model both hierarchies simultaneously.

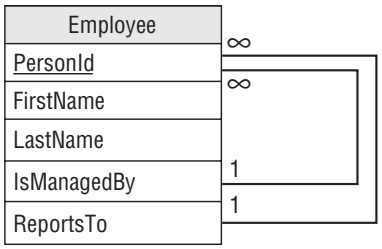

Figure 9-20

Notice that the relationships used to implement a hierarchy are ''upward-pointing.'' In other words, each object contains a reference to an object higher up in the hierarchy. This is necessary because each object has a single ''parent'' in the hierarchy but may have many ''children.'' Though you can list an object's parent in a single field, you cannot list all of its children in a single field.

Try It Out Working with Hierarchical Data

The following table contains information about a corporate org chart.

PersonId	Title	ReportsTo
1	Mgr. Pie and Food Gags	9
2	Dir. Puns and Knock-Knock Jokes	6
3	Dir. Physical Humor	9
4	Mgr. Pratfalls	3
5	President	null
6	VP Ambiguity	5
7	Dir. Riddles	6
8	Dir. Sight Gags	3
9	VP Schtik	5

Use this data to reconstruct the org chart graphically.

1. Find the record that represents the root node.
2. For each node on the bottom level of the tree so far (initially this is just the root node), find all of the records that have that node as a parent (ReportsTo). Attach them below their parent. For example, the first time around you would find the people who report to President. Their records have ReportsTo equal to President's PersonId: 5. These people are VP Ambiguity and VP Schtik. Attach them below President.
3. Repeat until you have processed every record.

How It Works

1. The root node is the one that has no parent. In the table, it's the one where ReportsTo is null: President.
2. Draw the root node.
   a. Find the people who report to President. Their records have ReportsTo equal to President's PersonId: 5. These people are VP Ambiguity and VP Schtik. Attach them below President.
   b. Find the people who report to VP Ambiguity. They are Dir. Puns and Knock-Knock Jokes and Dir. Riddles. Draw them below VP Ambiguity.
   c. Find the people who report to VP Schtik. They are Mgr. Pie and Food Gags and Dir. Physical Humor. Draw them below VP Schtik.
   d. Find the people who report to Dir. Puns and Knock-Knock Jokes. There are none so Dir. Puns and Knock-Knock Jokes is a leaf node.
   e. Find the people who report to Dir. Riddles. There are none so Dir. Riddles is a leaf node.
   f. Find the people who report to Mgr. Pie and Food Gags. There are none so Mgr. Pie and Food Gags is a leaf node.
   g. Find the people who report to Dir. Physical Humor. They are Mgr. Pratfalls and Dir. Sight Gags. Draw them below Dir. Physical Humor.
   h. Find the people who report to Mgr. Pratfalls. There are none so Mgr. Pratfalls is a leaf node.
   i. Find the people who report to Dir. Sight Gags. There are none so Dir. Sight Gags is a leaf node.

   At this point, every record is represented on the tree so we're done.
3. Figure 9-21 shows the finished org chart.

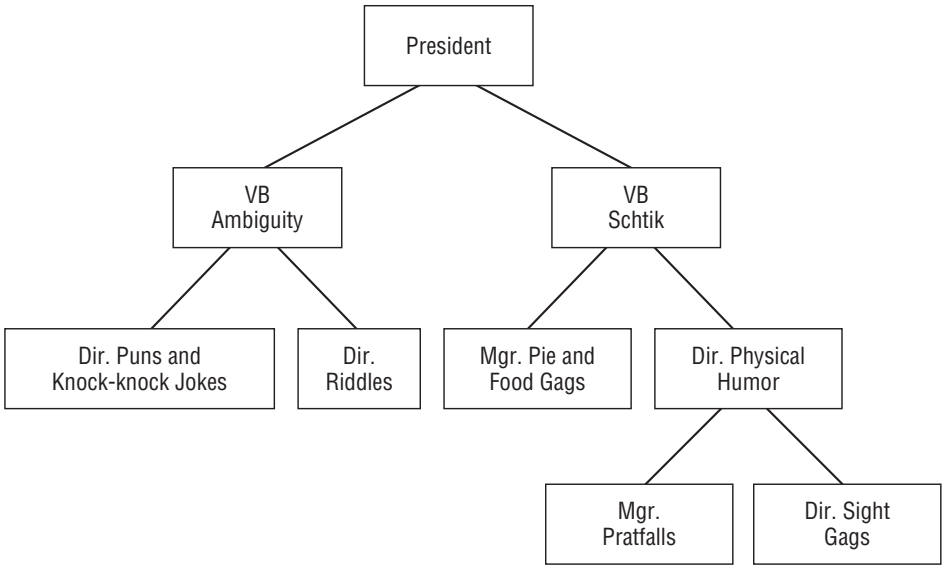

Figure 9-21

Network Data

A network contains objects that are linked in an arbitrary fashion. References in one object point to one or more other objects.

For example, Figure 9-22 shows a street network. Each circle represents a node in the network. An arrow represents a link between two nodes. The numbers give the approximate driving time across a link. Notice the one-way streets with arrows pointing in only one direction.

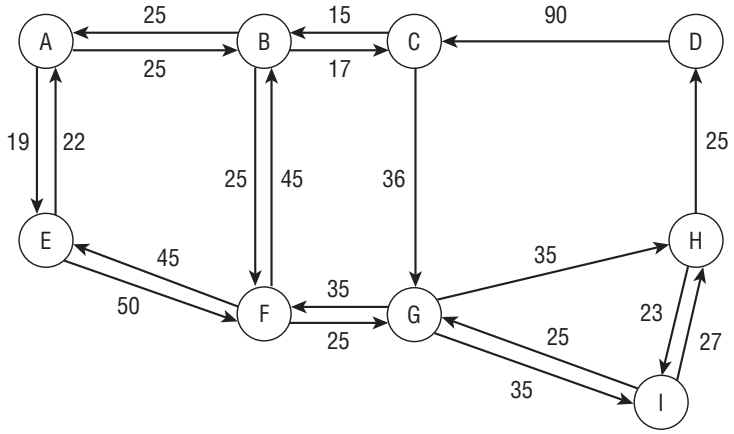

Figure 9-22

An object cannot use simple fields to refer to an arbitrary number of other objects, so this situation requires an intermediate table describing the links between objects.

Figure 9-23 shows an ER diagram describing the network's Node object. Notice that the Connects To relationship has a `LinkTime` attribute that stores the time needed to cross the link.

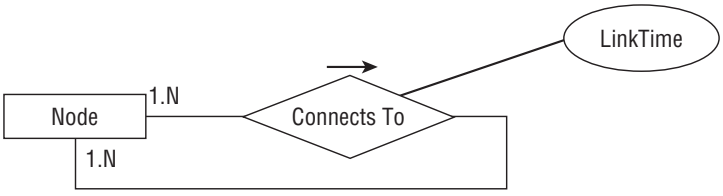

Figure 9-23

The section ''Many-to-Many Associations'' earlier in this chapter showed how to build a relational model for many-to-many relationships. That method just needs a small twist to make it work for a many-to-many reflexive relationship.

Instead of creating two tables to represent the related objects, just create a single Nodes table. Then create an intermediary table to represent the association between two nodes. That object represents the network link and holds the LinkTime data.

Figure 9-24 shows this design. In addition to a NodeId, the Nodes table contains X and Y coordinates for drawing the node.

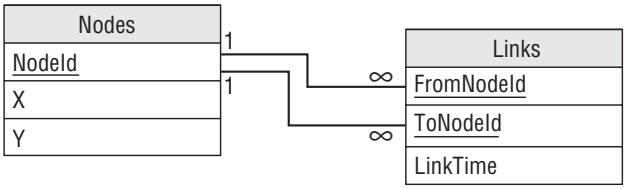

Figure 9-24

Note that you need to use some care when you try to use this data to build a network in a program. One natural approach is to start with a node, follow its links to new nodes, and then repeat the process, following those nodes' links.

Unfortunately if the network contains loops, the program will start running around in circles like a dog chasing its tail and it will never finish.

A better approach is to select all of the Nodes records and make program objects to represent them. Then select all of the Links records. For each Links record, find the objects representing the ''from'' node and the ''to'' node and connect them. This method is fast, requires only two queries, and best of all, eventually stops.

For a concrete example, consider the small network shown in Figure 9-25. The numbers next to links show the links' times.

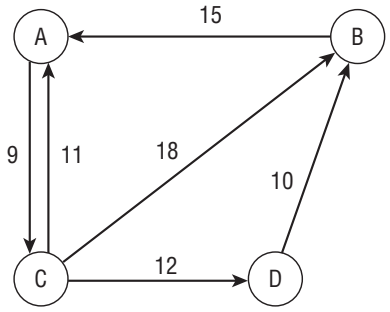

Figure 9-25

Start by making a Nodes table with fields NodeId, X, and Y. The following table shows the Nodes table's values. The X and Y fields are blank here because we're not really going to draw the network, but a real program would fill them in.

NodeId	X	Y
A		
B		
C		
D		

Next make a Links table with fields FromNode and ToNode, plus a LinkTime field. Looking at Figure 9-25, you see which nodes are connected to which others and what their LinkTime values should be. The following table shows the Links table's data.

FromNode	ToNode	LinkTime
A	C	9
B	A	15
C	A	11
C	B	18
C	D	12
D	B	10

Temporal Data

As its name implies, temporal data involves time. For example, suppose you sell produce and the prices vary greatly from month to month (tomatoes are expensive in the winter, while pumpkins are practically worthless on November 1). To keep a complete record of your sales, you not only need to track orders but also the prices at the time each order was placed.

The following sections describe a few time-related database design issues.

(For some more in-depth discussion of some of these issues, you can download the free eBook ''Developing Time-Oriented Database Applications in SQL'' at `http://www.cs.arizona.edu/people/rts/tdbbook.pdf`.)

Effective Dates

One simple way to track an object that changes over time is to add fields to the object giving its valid dates. Those fields give the object's effective or valid dates.

Figure 9-26 shows a relational model for temporal produce orders or orders for any other products with prices that change over time.

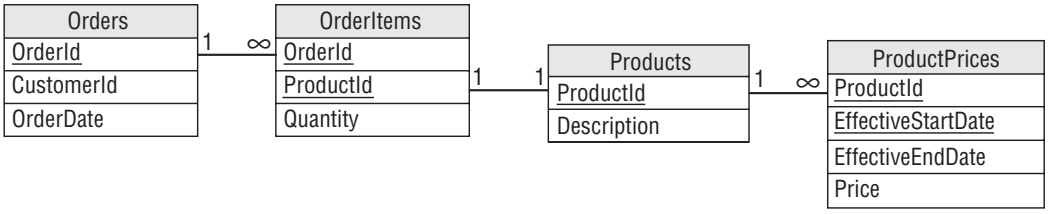

Figure 9-26

The Orders table contains an OrderId field and a Date, in addition to other order information such as CustomerId. The OrderId field provides the link to the OrderItems table.

Each OrderItems record represents one line item in an order. Its ProductId field provides a link to the Products table, which describes the product purchased on this line item. The Quantity field tells the number of items purchased.

The ProductPrices table has a ProductId field that refers back to the Products table. The Price field gives the product's price. The EffectiveStartDate and EffectiveEndDate fields tell when that price was in effect.

To reproduce an order, you would follow these steps:

1. Look up the order in the Orders table and get its OrderId. Record the OrderDate.
2. Find the OrderItems records with that OrderId. For each of those records, record the Quantity and ProductId. Then:
    a. Find the Products record with that ProductId. Use this record to get the item's description.
    b. Find the ProductPrices record with that ProductId and where EffectiveStartDate < = OrderDate < = EffectiveEndDate. Use this record to get the item's price at the time the order was placed.

The result is a *snapshot* of how the order looked at the time it was placed. By digging through all of these tables, you should be able to reproduce every order as it appeared when it was entered into the system.

Suppose you want to store one address for each employee but you want to track addresses over time. You don't want to track any other employee data temporally.

To build a relational model to hold this information, start by creating a basic Employees table that holds EmployeeId, FirstName, LastName, and other fields as usual.

Next design an Addresses table to hold the employee addresses. Create Street, City, State, and Zip fields as usual. Include an EmployeeId field to link back to the Employees record and EffectiveStartDate and EffectiveEndDate fields to track temporal data.

Figure 9-27 shows the resulting relational model.

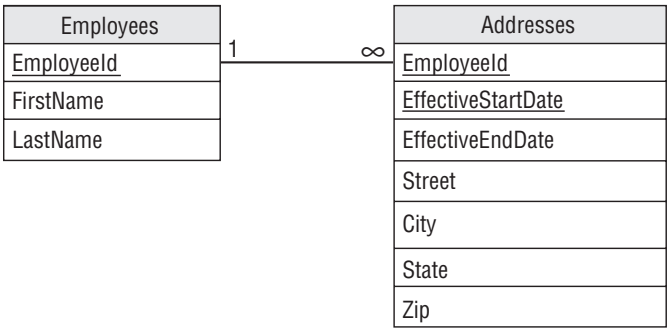

Figure 9-27

Deleted Objects

When you delete a record, the information that the record used to hold is gone forever. If you delete an employee's records, you lose all of the information about that employee including the fact that he was fired for selling your company's secrets to the competition. Because the employee's records were deleted, he could potentially get a job in another part of the company and resume spying with no one the wiser.

Similarly when you modify a record, its previous values are lost. Sometimes that doesn't matter but other times it might be worthwhile to keep the old values around for historical purposes. For example, it might be nice to know that an employee's salary was increased by only 0.25% last year so you might consider a bigger increase this year.

One way to keep all of this data is to never, ever delete or modify records. Instead you use effective dates to ''end'' the old record. If you're modifying the record rather than deleting it, you would then create a new record with effective dates starting now.

For example, suppose you hired Hubert Phreen on 4/1/2006 for a salary of $45,000. On his first anniversary, you increased his salary to $46,000 and on his second you increased it to $53,000. He then grew spoiled and lazy so he hasn't gotten a raise since. The following table shows the records this scenario would generate in the EmployeeSalaries table. Using this data, you can tell what Hubert's current salary is and what it was at any past point in time.

Employee	Salary	EffectiveStartDate	EffectiveEndDate
Hubert Phreen	$45,000	4/1/2006	4/1/2007
Hubert Phreen	$46,000	4/1/2007	4/1/2008
Hubert Phreen	$53,000	4/1/2008	1/1/3000

(To really be correct, you need to make one of the effective dates be exclusive. For example, you might decide that a record is valid starting on the effective start date up through but not including the effective end date. For example, Hubert's salary was $46,000 from April 1, 2007 through March 31, 2008. Then on April 1, 2008 his salary increased to $53,000.)

Deciding What to Temporalize

If you decide to use effective dates instead of deleting or modifying records, you will end up with a bigger database. Depending on how often these sorts of changes occur, it might be a *much* bigger database.

Disk space is relatively inexpensive these days (as little as $0.22 or so per GB and dropping every year) so that may not be a big issue. If the database is really huge, however, you may want to be selective in what tables you make temporal.

For example, in the model shown in Figure 9-26, only the ProductPrices table has effective dates. That would make sense for that example if you don't allow changes to orders after they are created.

That greatly reduces the amount of data that you will have to duplicate to record changes.

Before you rush out and add effective dates to everything in sight, carefully consider what data is worth saving in this manner.

Be sure to decide which tables to make temporal as early as possible because retrofitting effective date fields can be very difficult, particularly for any programs that access the data. Any queries that request data from tables with effective dates must be parameterized to get the right data. If you add effective date fields after you start development, you need to modify all of those queries and that gives you extra chances to make mistakes and insert bugs in the system.

This is definitely a case where you want thorough planning before you start to build.

Logging and Locking

Two techniques that I've found useful in a number of database applications are audit trails and turnkey records. Audit trails let you log changes to key pieces of data. Turnkey records let you easily control access to groups of related records.

Audit Trails

Many databases contain sensitive data, and it is important to make sure that the data is safe at all times. Though you cannot always prevent a user from incorrectly viewing or modifying the data, you can make a record showing who made a modification. Later, if it turns out that the change was unauthorized, you can hunt down the perpetrator and wreak a terrible vengeance.

> *For example, in 2007 State Department contractors ''inappropriately reviewed'' the passport files of then Senator and presidential candidate Barack Obama. The offenders were probably just curious, but it violated the department's privacy rules so two people were fired and one reprimanded.*

One way to provide a record of significant actions is to make an *audit trail* table. This table has fields Action, Employee, and Date to record what was done, who did it, and when it happened. For some applications, this information can be non-specific, for example, recording only that a record was modified and by whom. In other applications, it might record the fields that were modified and the old and new values.

A similar technique works well with the effective dates described in the section ''Temporal Data'' earlier in this chapter. If you never delete or update records, you can add a CreatedBy field to a table that you

want to audit and fill in the name of the user who created the record. Later if someone modifies the record, you will be able to see who made the modification in the new version of the record.

You may still want a separate AuditEvents table, however, to record actions other than creating, deleting, and modifying records. For example, you might want to keep track of who views records (as in the State Department passport case), generates or prints reports, sends emails, or prints letters. You might even want to record user log in and log out times.

Turnkey Records

When a user needs to modify a record, the database locks that record so other users can't see an inconsistent view of the data. This prevents others from seeing a half-completed operation.

Relational databases also provide transactions that allow one user to perform a series of actions atomically — as if they were a single operation. This effectively locks all of the records involved in those actions until the transaction is complete.

Though these features work well, their record-locking behaviors can lead to a couple of problems.

First, most databases won't tell you who has a record locked. If someone is in the middle of editing a record in the Employees table, you won't be able to edit that record. Unfortunately the database also won't tell you that Frank has the record locked so you can't go down the hall and ask him to release it. Or worse, you'll discover that Frank left his computer locked and went home for an early weekend so you're stuck until Monday. In that case, it will require database administrator powers and an act of Congress to unlock the record so you can get some work done.

A second issue is that a complicated series of locks adds to the database's load.

One technique I've found useful for addressing these problems is to use turnkey records to control access to a group of tables.

Suppose normalization has spread a work order's data across several tables holding basic information, addresses, phone numbers, email addresses, and other stuff. Now suppose the system is designed to assign work orders to users for processing. It would be nice to lock a work order's data while a user is working on it so others can't blunder in and make conflicting changes. Unfortunately, it's wasteful to lock all of those records.

To use a turnkey record, add a LockedBy field to a table that is central to the work order. This is probably the table that contains the work order's basic information.

Now to ''reserve'' the work order for use by a particular user, the program sets this record's LockedBy field to the user's name. That token means that this user has permission to mess with all of the work order's records in all of its tables without actually locking anything. Because the user's name is in the database, other parts of the program can tell that the record is locked and by whom. The program can even allow an administrator with appropriate privileges to clear that field so you can fix the work order after Frank has gone home.

The one drawback to this method is that if Frank's computer crashes while he has the work order reserved, then it remains locked. To recover, you'll need to add an option in the program to clear those sorts of zombie reservations.

A similar technique gives most of the same benefits while removing the problems that come with a LockedBy field. Suppose you assign each work order to a particular person who then works on it, and no one else ever works on that order.

To handle this case, add an AssignedTo field to the order. Some agent (either human or automated) sets the AssignedTo field and after that the field doesn't need to be changed. In that case, if Frank's computer crashes, his record is still assigned to him after he reboots. Because Frank is still the one who should work on the job, you don't need to clear this field. (Although in practice there will always be a situation where someone needs to foist a job off on someone else for some weird reason, so you should allow some way for an administrator to step in and fix it if necessary.)

Summary

This chapter described some common patterns that you can use to solve particular database design problems. For example, if you need to build a database that includes many-to-many relationships, you can use the pattern described in the section ''Many-to-Many Associations'' to implement that part of your relational database design.

In this chapter you learned to model:

- Many-to-many relationships and multiple-object associations
- Repeated attribute associations
- Reflexive or recursive associations
- Temporal data
- Logging and locking

The next chapter does the opposite of this one. It describes common mistakes people make when designing databases and explains ways to avoid those mistakes.

Before you move on to Chapter 10, however, use the following exercises to test your understanding of the material covered in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. Parcheesi is a board game for two to four players. Make an ER diagram to record information about Parcheesi matches.
2. Build a relational model to record information about Parcheesi matches. Be sure to include a way to tell who finished first through fourth.
3. Chess enthusiasts often like to walk through the moves of past matches to study how the play developed. They even give names to the most famous of these matches. For example, the ''Immortal Game'' was played on June 21, 1851 by Adolf Anderssen and Lionel Kieseritzky (see `http://en.wikipedia.org/wiki/Immortal_game`).

   The following text shows the first six moves in the Immortal Game:

   ```
   e4 e5 f4 exf4 Bc4 Qh4+?! Kf1 b5?! Bxb5 Nf6 Nf3 Qh6
   ```

(If someone showed me this string and I wasn't thinking about chess at the time I'm not sure whether I would guess it was an assembly program, encrypted data, or some new variant of Leet. See `http://en.wikipedia.org/wiki/Leet`.)

Of course, a database shouldn't store multiple pieces of information in a single field, so the stream of move data should be broken up and stored in separate fields. In chess terms, a *ply* is when one player moves a piece and a *move* is when both players complete a ply.

Figure 9-28 shows a semantic object model for a `CHESS_MATCH` class that stores the move information as a series of `Move` attributes, each containing two `Ply` attributes. The Movement field holds the actual move information (Qh4+?!) and MoveName is something like ''The Sierpinski Gambit'' or ''The Hilbert Defense.'' Commentary is where everyone else pretends they know what the player had in mind when he made the move.

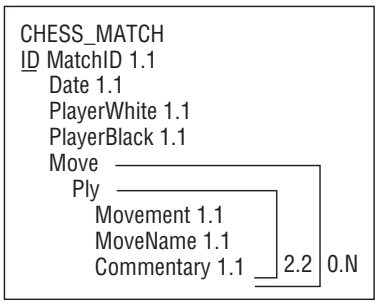

Figure 9-28

Draw an ER diagram to show the relationships between the `Player`, `Match`, `Move`, and `Ply` entity sets.

4. Build a relational model to represent chess relationships by using the tables Players, Matches, Moves, and Plies. How can you represent the one-to-two relationship between Moves and Plies within the tables' fields? How would you implement this in the database?
5. Consider the relational model you built for Exercise 4. The Moves table doesn't contain any data of its own except for MoveNumber. Build a new relational model that eliminates the Moves table. (Hint: collapse its data into the Plies table.) How does the new model affect the one-to-two relationship between Moves and Plies?
6. Suppose you are modeling a network of pipes and you want to record each pipe's diameter. Design a relational model that can hold this kind of pipe network data.
7. Suppose you run a wine and cheese shop. Wine seems to get more expensive the longer it sits on your shelves, but most cheeses don't last forever. Build a relational model to store cheese inventory information that includes the date by which it must be sold. Assume that each batch of cheese gets a separate sell-by date.
8. Modify the design you made for Exercise 7 assuming each *type* of cheese has the same shelf-life.