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.

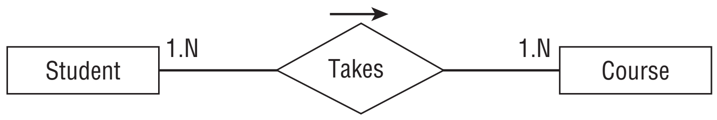

Figure 9-1

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.

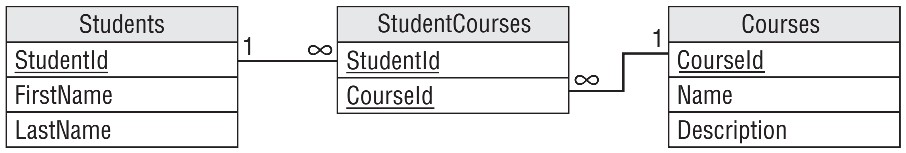

Figure 9-2

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.

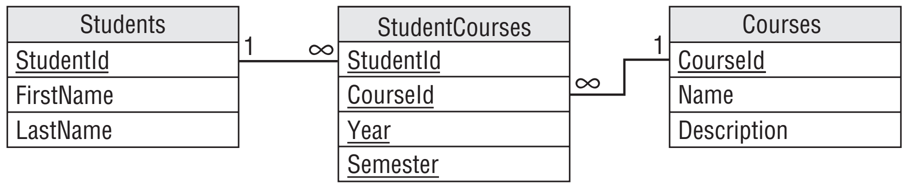

Figure 9-3

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.

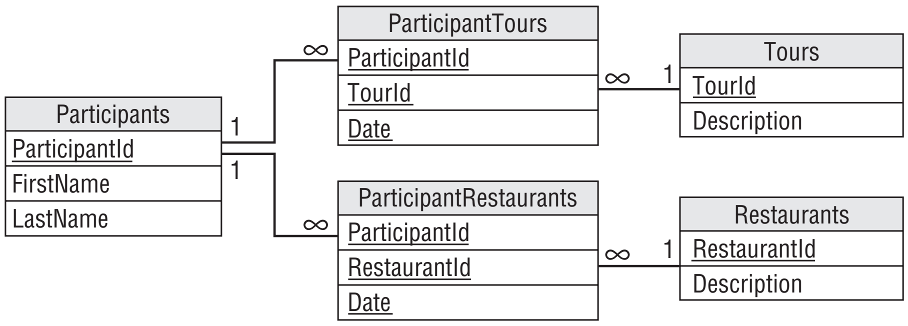

Figure 9-4

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.

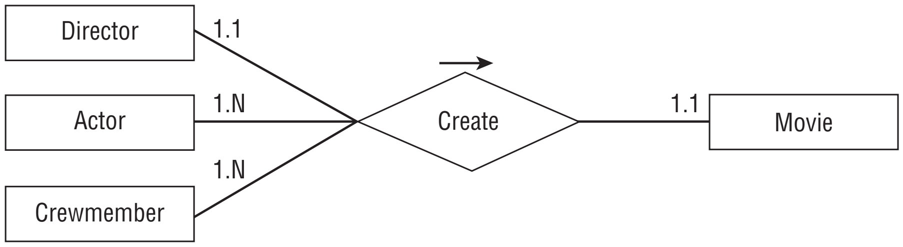

Figure 9-5

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.

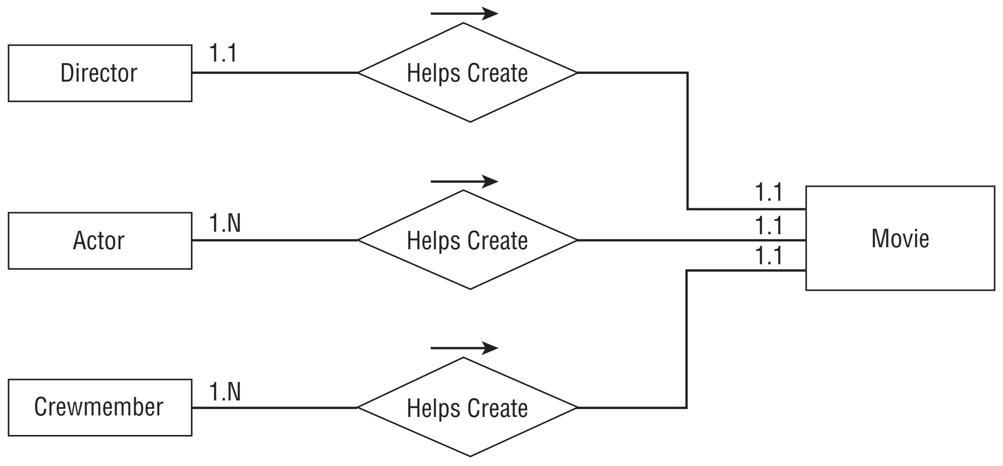

Figure 9-6

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

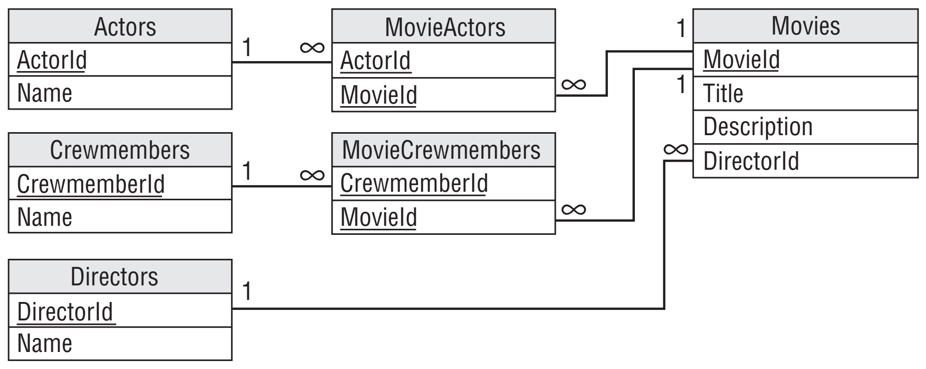

Figure 9-7

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.

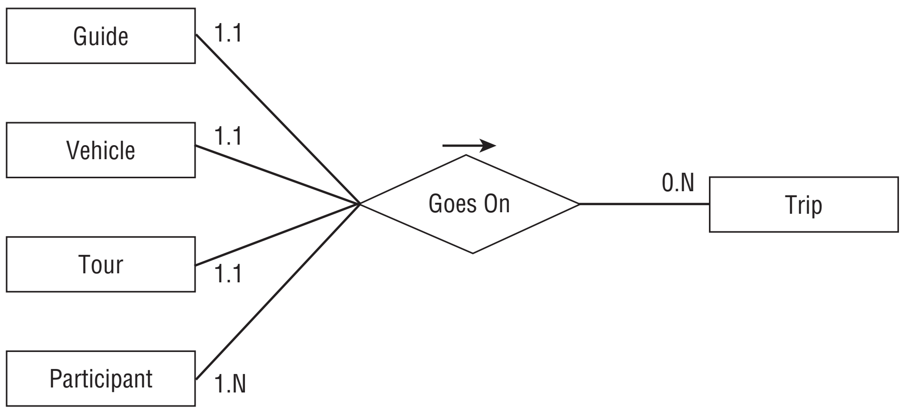

Figure 9-8

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.

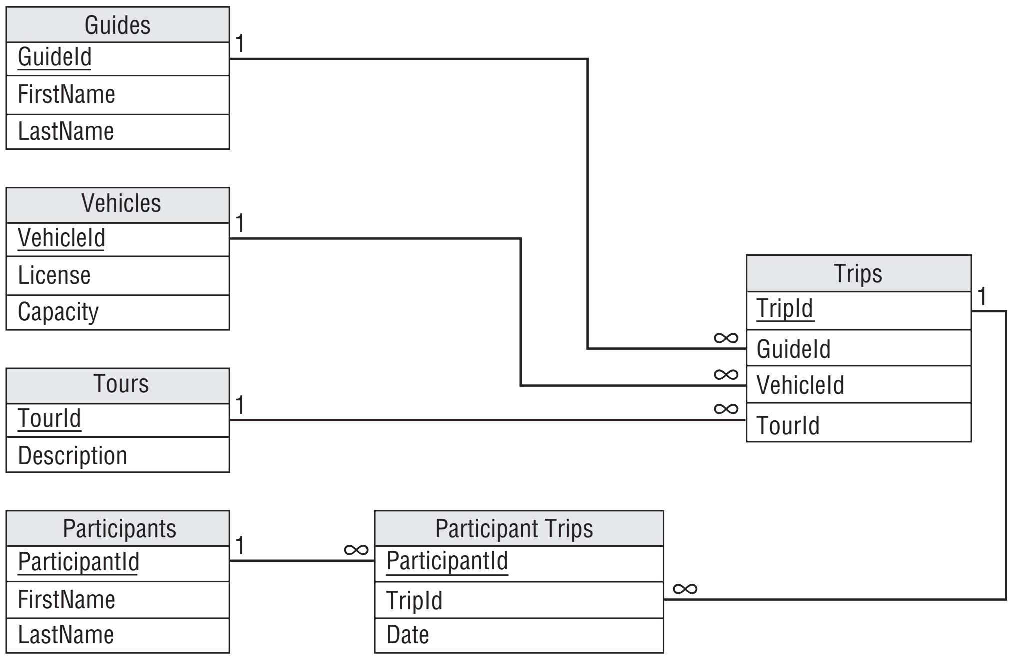

Figure 9-9

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.

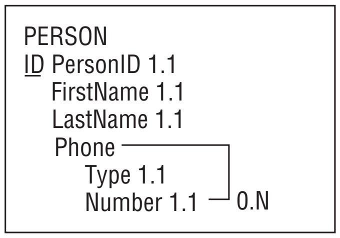

Figure 9-10

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.

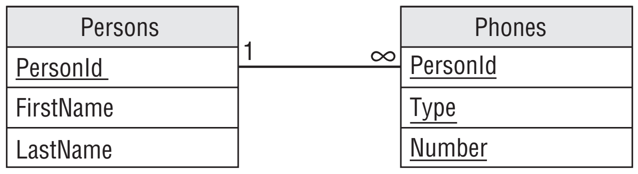

Figure 9-11

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.

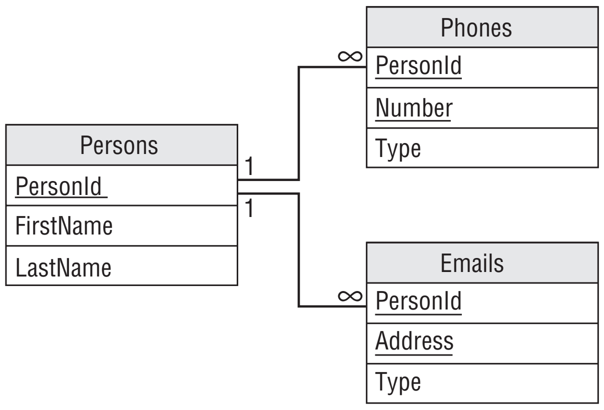

Figure 9-12

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.

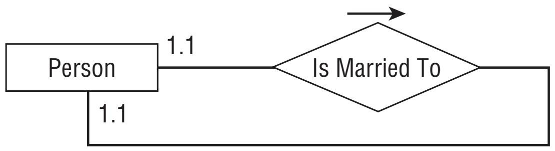

Figure 9-13

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

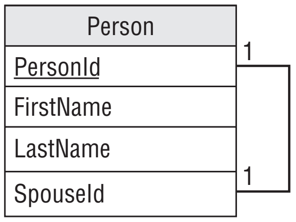

Figure 9-14

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.

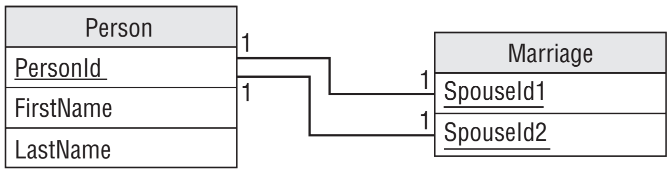

Figure 9-15

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.

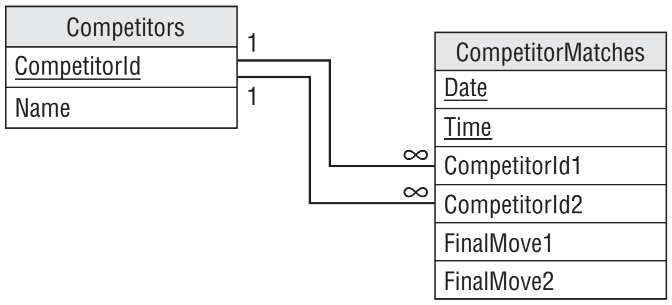

Figure 9-16

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.

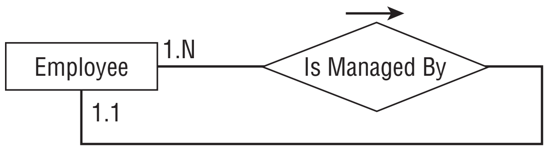

Figure 9-17

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

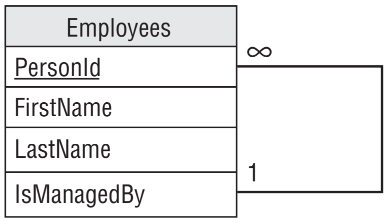

Figure 9-18

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.

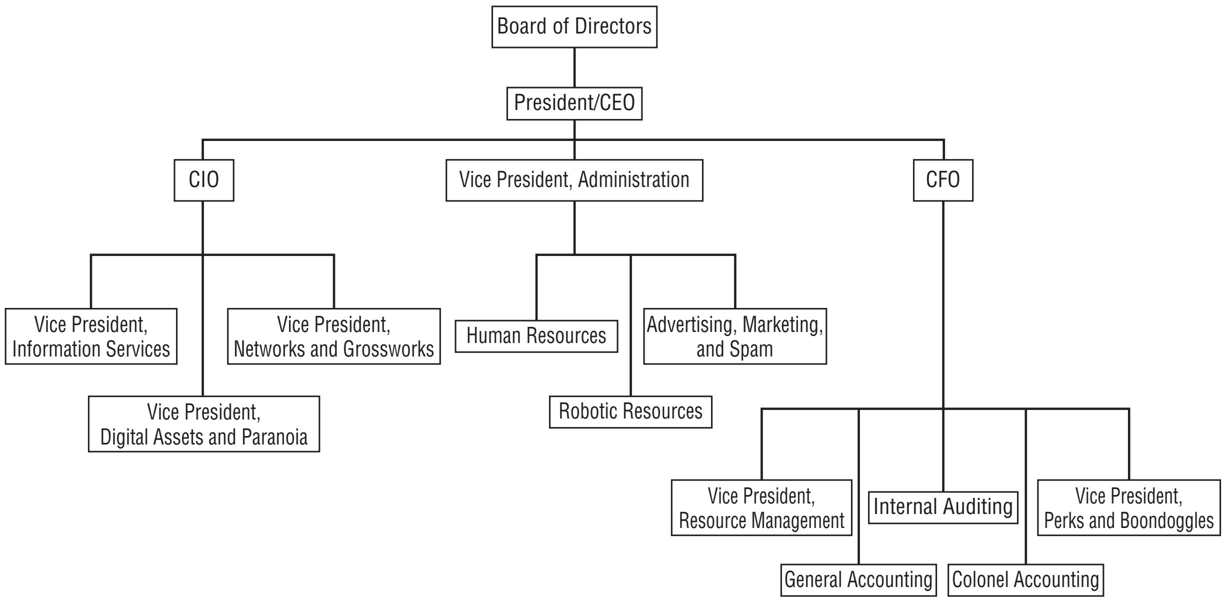
Board of Directors
President/CEO
CIO
Vice President, Administration
CFO
Vice President, Information Services
Vice President, Networks and Grossworks
Vice President, Digital Assets and Paranoia
Human Resources
Advertising, Marketing, and Spam
Robotic Resources
Vice President, Resource Management
Internal Auditing
Vice President, Perks and Boondoggles
General Accounting
Colonel Accounting

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.

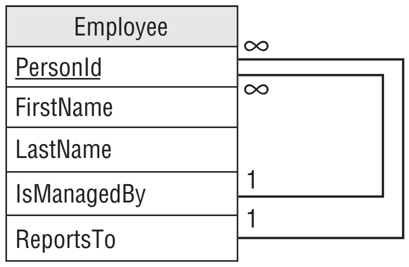

Figure 9-20

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.

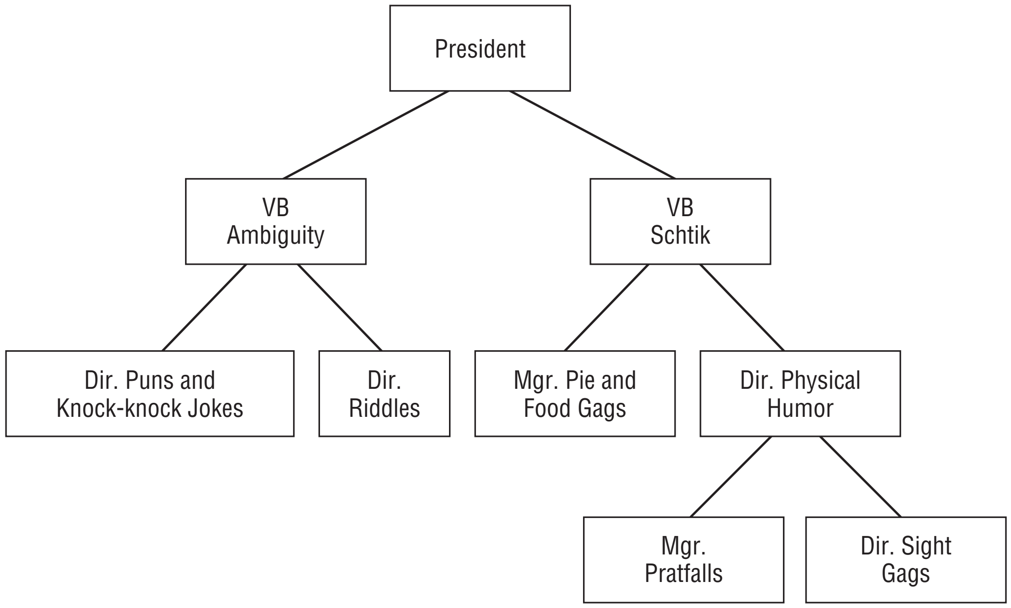

Figure 9-21

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.

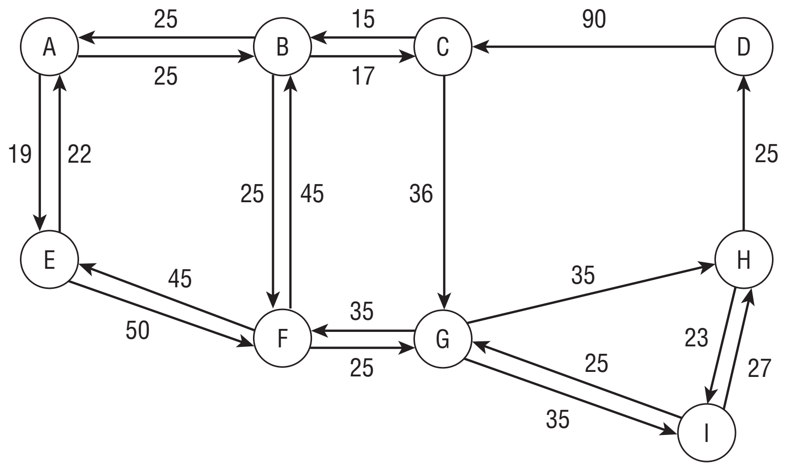

Figure 9-22

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.

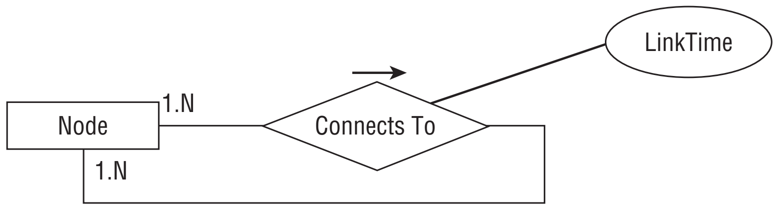

Figure 9-23

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.

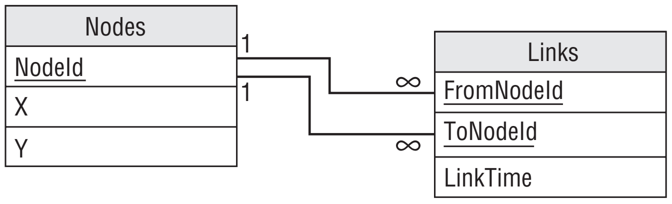

Figure 9-24

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.

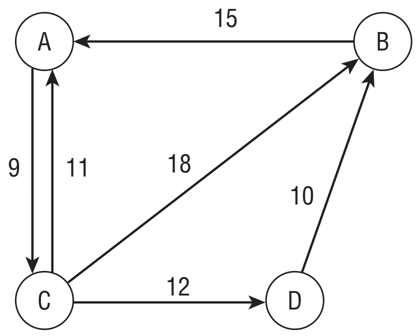

Figure 9-25

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.

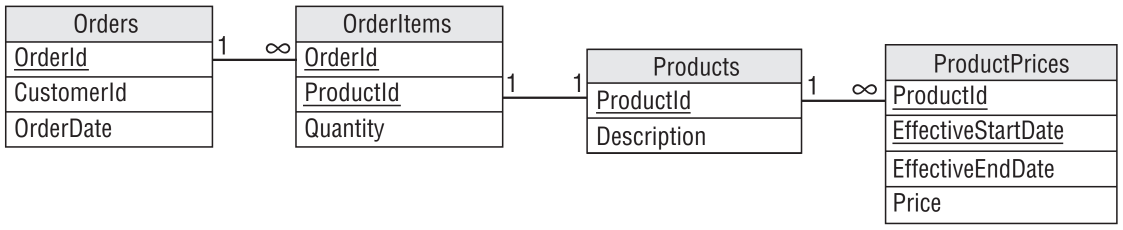

Figure 9-26

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.

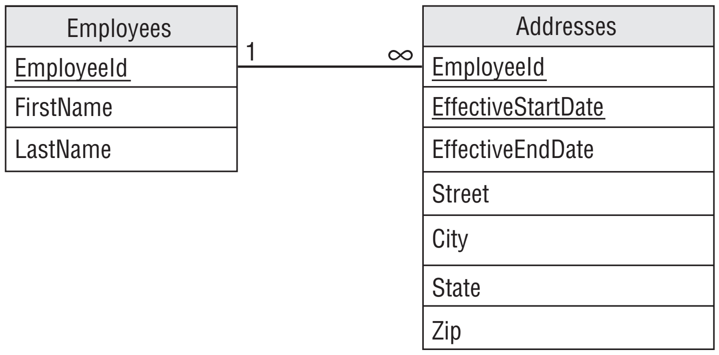

Figure 9-27

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.

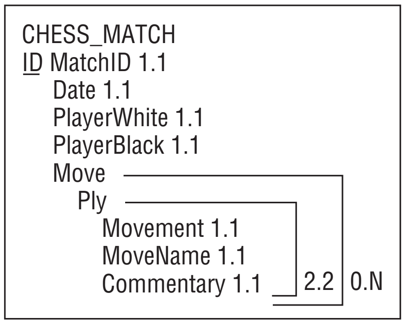

Figure 9-28

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.

10

Common Design Pitfalls

Chapter 9 described some common patterns that you may want to use while designing a database. This chapter takes an opposite approach: it describes some common pitfalls that you don't want to fall into while designing a database. If you see one of these situations starting to sprout in your design, stop and rethink the situation so you can avoid a potential problem as soon as possible.

In this chapter you learn to avoid problems with:

- ❑ Normalization and denormalization.
- ❑ Lack of planning and standards.
- ❑ Mishmash and catchall tables.
- ❑ Performance anxiety.

The following sections describe some of the most common and troublesome problems that can infect a database design.

Lack of Preparation

I've nagged about this in earlier chapters but it's time to nag again. Database design is often one of the first steps in development. It's only natural for developers to want to rush ahead and get some serious coding done. That gives them something to show management to prove that they aren't *only* playing computerized mahjong and reading friends' MySpace pages. It's also more fun than working on plans, designs, use cases, documentation, and all the other things that you need to do before you can roll up your sleeves and get to work.

Before you start cranking out tables and code, you need to do your homework. Some of the things you need to do before you start wiring up the database include:

- ❑ Make sure you understand the problem.
- ❑ Write requirements documents to state the problem.
- ❑ Build use cases to see if you have solved the problem.

- ❑ Design a solution.
- ❑ Test the design to see if it satisfies the use cases.
- ❑ Document everything.

Remember, the time you spend on up-front design almost always pays dividends down the road.

Poor Documentation

This is part of preparation but is so important and under-appreciated that deserves its own section. Many developers think of documentation as busy work or a chore to keep managers who have no real talents off their backs while they build elegant data structures of intricate beauty and complexity.

I confess that occasionally that's a handy attitude, but the real purpose of documentation is to keep everyone on the project focused on the same goals. The documentation should tell people where the project is headed. It should spell out the project's design decisions so everyone knows how the pieces will fit together.

If the documentation is weak, different people will make different and often contradicting assumptions. Eventually those assumptions will collide and you'll have to resolve the conflict. That will require developers to go back and fix work that they made under the wrong assumptions. That leads to more work, more errors, and copious bickering over whose fault it was.

The real fault was poor documentation.

Poor Naming Standards

In a sense, naming standards are part of documentation. When done properly, an object's name should give you a lot of information about the object. For example, if I tell you to build an Employees table, you probably know a lot about that table without even being told. You know that it will need name, address, phone, email, and Social Security Number information (in the United States, at least). In most companies, it will also need an employee ID, hire date, title, department, salary, and payroll information (deductions, bank account for automatic deposit, and so forth). Somehow it should probably link to a manager and possibly to projects. You got all of that from the single word ''Employees.''

Now suppose I told you to build a People table but I really want to use the table to hold employee data. You'd probably only put about half of the necessary fields in this table. You'd get the name and address stuff right, but you'd completely miss the business-related fields.

The problems become worse when you start working with multiple related tables and fields. For example, suppose you use employee IDs to link a bunch of tables together but one table calls the linking field EmpNo, another calls it EmployeeId, and a third calls it Purchaser.

This may seem like a small inconvenience and in isolation it is, but together a lot of little inconveniences can add up to a real headache. Inconsistent naming makes developers think harder about names than the things they represent and that makes them less productive and less accurate.

I have worked on projects where poor naming conventions made small changes take days instead of hours because developers had to jump back and forth through the code to figure out what was happening. Inconsistent naming by itself is unlikely to sink a project, but it is enough to nudge an already leaky ship toward the rocky shoals.

Write down names for fields that will be used in more than one table and stick to them so the same concept gets the same name everywhere. Then use those names consistently. Consistency is more important than following particular arcane formulae to generate names, although I will mention two useful conventions for naming database objects such as tables and fields.

First, don't use keywords such as `TABLE`, `DROP`, and `INDEX`. Though these may make sense for your application and they may be allowed by the database, they can make programming confusing. If one of these words really fits well for your project, try adding something to make it even more descriptive. For example, if your database will hold seating assignments and it really makes sense to have a field named Table, try naming it TableNumber or AssignedTable instead.

Second, don't put special characters such as spaces in table or field names even if the database allows it. Although there are ways to use these sorts of names, it makes working with the database a lot more confusing and remember, the point of good naming conventions is to reduce confusion.

For some more information on naming conventions, see some of these links:

- `http://en.wikipedia.org/wiki/Identifier_naming_convention`
- `http://standards.iso.org/ittf/PubliclyAvailableStandards/c035347_ISO_IEC_11179-5_2005(E).zip`
- `http://www.gorillatraining.com/en-us/library/ms229002.aspx`
- `http://vyaskn.tripod.com/object_naming.htm`

Picking good names for tables is like a vocabulary test. You need to think of a word or short phrase that sums up as many of the features in the related items as possible so someone else who looks at your table's name will immediately understand the characteristics that you're trying to record. The following table shows some examples.

A table that holds:	**Should be called:**
Magazines, newspapers, and comic books	Periodicals
Things that your company sells, including physical items and services	Products
People who work in your restaurant, including servers, bussers, cooks, and greeters	Employees
Things that cost you money, such as groceries, gasoline, and fencing lessons	Expenses
Things that you pay for but that are for work purposes, such as stationery, stamps, and phone calls	BusinessExpenses

Thinking Too Small

Too often developers design a perfectly reasonable database only to discover during the final stages of the project that it cannot handle the load that's dumped on it. Make some calculations, estimate the database's storage and transaction loads, calculate the likely network traffic, and then multiply by five. For some applications, such as online Web applications that can have enormous spikes in load over just a few hours, you might want to multiply by ten or more.

Be sure you use a realistic model of the users' computers and networks. It's fairly common in software development to give the programmers building a system great big, shiny, powerful computers so they can be more productive. (It takes a lot of horsepower to play those interactive role-playing games quickly so you can get back to work.)

Unfortunately, customers often cannot afford to buy new computers for every user. (Five developers times $3,000 is $15,000. That's not exactly pocket money, but it's nothing compared to $2,000 times 200 users for a total of $400,000.) Make sure your calculations are based on the hardware that the users will really have, not on the dream machine that you are using.

If you don't think your architecture can handle that load, you should probably rethink things a bit. You may be able to buy a more powerful server, buy more disk space, move to a faster network, or split the data across multiple servers. If those tricks don't work, you might need to consider a three-tier architecture with different middle-tier objects running on separate computers. You might need to think about moving some of the more intense calculations out of database code and moving them into code running on separate servers. You might need to redesign the database to use turnkey records. You might even need to split the database into disjoint pieces that can run in different computers.

Solving these problems may be difficult, but you should at least plan ahead and be prepared to face them. A sure way to ruin customer goodwill is to get the customers all excited, release the database, and then tell them they can't use it for four months while you rethink its performance problems.

Not Planning for Change

As you design the database, look for places that might need to change in the future. You don't need to build features that may never be needed, but you don't want to narrow the design so those features cannot be implemented later.

In particular, look for exceptions in the data. Customers often think in terms of paper forms, and those are easy to modify. It's easy to cross out headings and scribble in the margins of a paper form. It's a lot harder to do that in a computerized system.

Whenever you see two or more things that have a lot in common, ask the customers if those are enough or whether you'll sometimes need to add more. Listen for words such as ''except,'' ''sometimes,'' and ''usually.'' Those words often hint at changes yet to come.

For example, suppose a customer says, ''This field holds the renter's front binding tension, unless he's goofy-footed.'' Here the word ''unless'' tells you that this one field may not be good enough to hold all of the data. You'd better find out what ''goofy-footed'' means and change the database accordingly.

For another example, suppose the customer says, ''The order form must hold two addresses, one for shipping and one for billing. Unless, of course, we're billing a split order.'' This says that two address fields (or groups of fields) isn't enough. At this point you probably need to pull the address data into a new table so you can accommodate any number of addresses, including the ones the customer hasn't remembered yet.

For a third example, suppose you're building a coaching tool for youth soccer teams. Figure 10-1 shows your initial design.

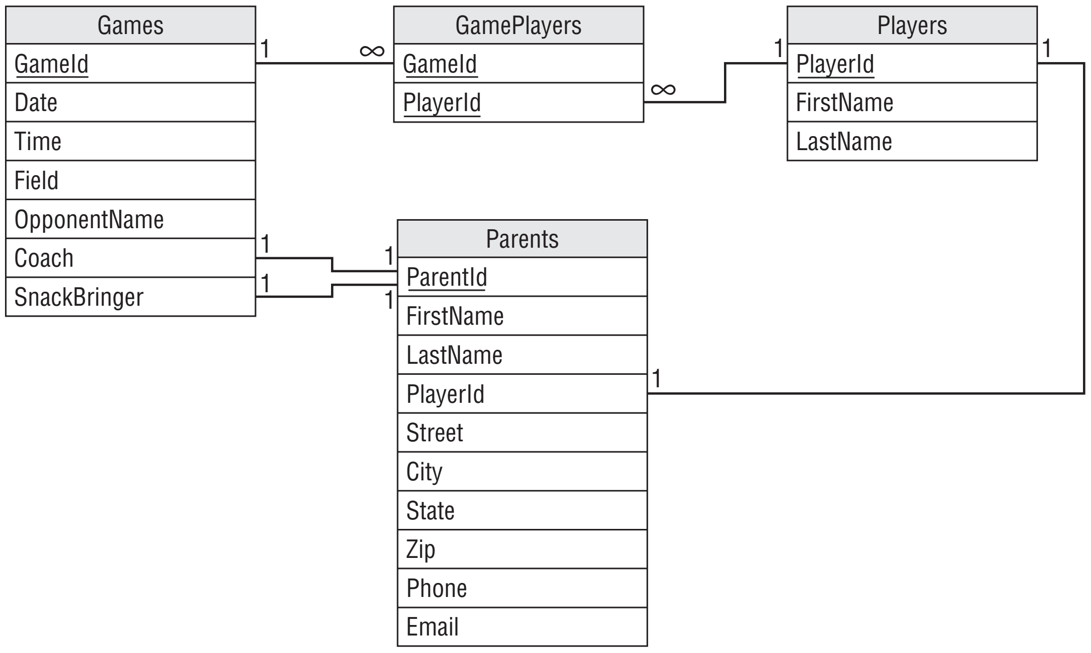

Figure 10-1

Let's review this design and identify any pieces that seem like they might change later.

It's often useful to look at fields that allow a single value and ask yourself if they might need to change later. In this design, there are several such fields. It's also particularly useful to look at one-to-one relationships and this design contains some of those, too.

First, do we need to change the one-to-many relationship between Games and GamePlayers to a many-to-many relationship? Probably not. If the group of players is the same in any two games, that's more or less coincidence rather than a more formal arrangement such as an ''A team'' and a ''B team.'' (Although I've known some frighteningly serious youth soccer coaches. Seriously scary individuals with clipboards yelling at the tops of their lungs at four-year-olds.)

The many-to-one relationship between GamePlayers and Players, together with the Games/GamePlayers relationship, helps model the many-to-many relationship between Games and Players, so it probably shouldn't change.

What about the one-to-one relationship between Parents and Players? This link implies that a parent can have only one player and that may not be a good assumption. What if a parent has two players on the same team? For the younger age brackets, you won't see players of different ages on the same team, but you will find twins on the same team.

This relationship also implies that each player has a single parent (which is unlikely until cloning techniques become more practical). You could add information about a second parent (in fact, that's a very common approach) but if a player's parents are separated and remarried, you might need up to four parents and sometimes you might need to contact any subset of them to figure out if a player will be at a game. It might make the most sense to just allow a player to have any number of parents and not ask too many questions. (And don't even think about requiring players to have the same last names as their parents! The combinations, including hyphenated last names, are too numerous to contemplate.)

The first one-to-one relationship between Games and Parents means that one parent will be coach for that game. Is that a reasonable assumption? Will you ever need to worry about multiple coaches or assistant coaches? For a youth league, it's probably good enough to assume there is only one main coach and not worry about any others, so I wouldn't change that. But this is a good question to ask.

By far the most important piece of information in this database tells who is bringing a game's snacks. The second one-to-one relationship between Games and Parents means one parent will bring snacks for the game. Is that a reasonable assumption? In my experience, there has always been only one snack-bringer per game. As in the case with the coach, you can probably at least assume there is a main snack-bringer and if someone else wants to bring extra cupcakes for a player's birthday we just won't worry about it. But again, a good question.

One final place to look for these kinds of changes is in the fields within a table. In this design, the field that begs for multiple values is Phone. Lots of people have multiple phones and sometimes you may need to call several to track someone down. (Such as the all-important snack bringer!) I would split the Phone field off into a new table to allow parents to have any number of phone numbers.

Figure 10-2 shows the new and improved design.

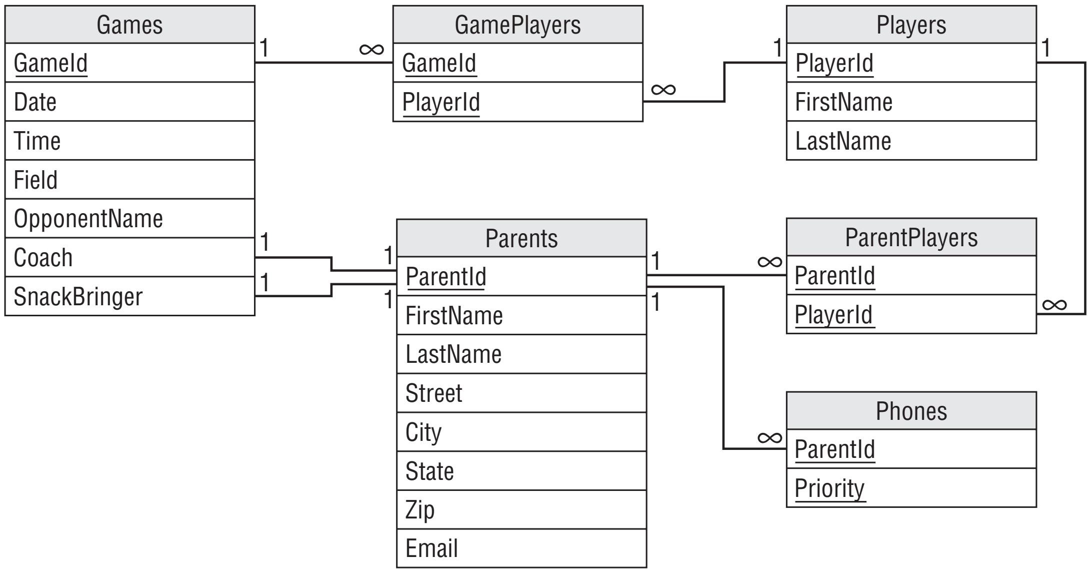

Figure 10-2

Again, you don't need to use psychic powers to build all of the features that the customer will need in the next 15 years, but keep your eyes open during requirements gathering and try not to close off opportunities for later change.

Too Much Normalization

Taken to extremes, too much normalization can lead to a database that scatters related data all over the place for little additional benefit. It can make the design confusing and can slow performance.

When you normalize, think about what a change will cost and what benefits it will provide. Think about how the data will be accessed. If data is only read and written through stored procedures or middle-tier code, that code can help play a role in keeping the data consistent and may allow you to get away with slightly less normalization in the database's tables. Putting every table in Fifth Normal Form or Domain/Key Normal Form isn't always necessary to keep the data safe.

I once worked on a project where a certain database developer (who coincidentally had just taken a class in database normalization) wanted to split every data value out into a separate table. For example, a customer record would contain little more than a CustomerId. Then a Values table would hold the actual data in its three fields Id, ValueName, and ValueData. To look up a customer's name, you would search the Values table for a record with Id equal to the customer's ID and ValueName equal to ''Name.'' In some bizarre otherworldly sense, this table is very normalized and it lets you do some amazing things. For example, you could decide to add a new EarSize field to the customer data without changing the tables at all. However, that design doesn't reflect the structure of the data so it would be next to impossible to use.

Insufficient Normalization

Though too much normalization can make the database slower than necessary, poor performance is rarely the reason a software project fails. Much more common reasons for failure are designs that are too complex and confusing to build, and designs that don't do what they're supposed to do. A database that doesn't ensure the data's integrity definitely doesn't do what it's supposed to do.

Normalization is one of the most powerful tools you have for protecting the data against errors. If the database refuses to allow you to make a mistake, you won't have trouble with bad data later. Adding an extra level of indirection to gather data from a separate table adds only milliseconds to most queries. It's very hard to justify allowing inconsistent data to enter the database to save one or two seconds per user per day.

This doesn't mean you need to put every table in Fifth Normal Form, but there's no excuse for tables that are not in at least Third Normal Form. It's way too easy to normalize tables to that level for anyone to claim that it's not necessary.

If the code needs to parse data from a single field (Hobbies = ''sail boarding, skydiving, knitting''), break it into multiple fields or split its values into a new table. If a table contains fields with very similar names (JanPayment, FebPayment, MarPayment), pull the data into a new table. If two rows might contain identical values, figure out what makes them logically different and add that explicitly to the table so you can make a primary key. If some fields' values don't depend on the entire key, consider spreading the record across multiple tables.

Insufficient Testing

This problem is closely related to ''Thinking too Small'' and ''Too Much Normalization.'' Some developers perform little or no testing before releasing a database into the wild. They run through a few tests to check correctness (the better ones go through all of the use cases) and assume that everything will work in the field. Then when customers try to use it under realistic conditions, the whole thing falls apart. They discover bugs that the testers missed and the performance is unacceptable.

Be sure to test the database and any attached applications thoroughly. Fully testing every nook and cranny of a system takes a lot of work, but it's necessary. You need to be sure you exercise every piece of code, every table, and every constraint. You also need to perform load testing to see if the database can handle the expected load.

If you don't find all of the bugs and bottlenecks, sooner or later the user will, guaranteed!

It's also pretty safe to assume that every non-trivial application contains at least some bugs no matter how much testing you perform. In that case, you cannot find every bug so you may be temped not to even try. The goal isn't really to catch every bug, but to find enough of them and to find the most likely to occur so the probability of the users finding a bug is very small. The bugs will still be hiding in there, but if you only get one or two user complaints per year, you're doing pretty well.

Performance Anxiety

Many developers focus so heavily on performance that they needlessly complicate things. They make a simple solution complicated and harder to build and maintain all in the name of speed. They denormalize tables to avoid using any more tables than necessary and they build business rules into the database so they don't need to use stored procedures or other code to implement them separately.

Modern hardware and software is pretty fast, however. Often these CPU-pinching measures save only milliseconds on a one-second query. Think hard about whether a convoluted design will really save all that much time before you make things so complicated that you can't build, debug, and maintain the application. If you're not sure, either make some tests and find out or go with the simpler version and change it later if absolutely necessary. Usually performance is acceptable, but a database that contains contradictory data is not.

I once worked on a huge database application where a simple change to the data might require five or more minutes of recalculation. After about three days digging through horribly convoluted code and database structure, I found the problem. The original developers had used a bunch of tricks to perform calculations in some sneaky ways to save a little bit of time here and there. Then they had done something really silly that made them perform the same calculations again and again more than a hundred thousand times. They were so busy worrying about tripping over the blades of grass that they wandered blindly into a patch of poison ivy. I managed to speed things up a little, but a lot of their time-saving tricks were so buried in the underlying design that there wasn't much we could do without a total rewrite.

The moral is, you don't need to be stupid about design and ignore obvious chances to improve performance, but don't be so focused on the little things that they cloud the grander design.

First, make it work. Then make it work fast.

Mishmash Tables

Sometimes it's tempting to build tables that contain unrelated values. The classic example is a DomainValues table that contains allowed values for fields in tables scattered throughout the database. For example, suppose the State, Brand, and Medium fields take values from lists. State can take values CA, CT, NV, and so forth; Brand can take values MixAll, Thumb Master, and Pheidaux; and Medium can take values Oil, Acrylic, Pastel, and Crayon. You could build a DomainValues table with fields TableName, FieldName, and Value. Then it would hold records such as TableName = Artwork, FieldName = Medium, Value = Crayon. You would use this magic table to validate foreign keys in all of the other tables.

This approach will work, but it's more of a headache than it's worth. The table is filled with unrelated values and that can be confusing. It might seem that having one table rather than several would simplify the database design, but this single table does so many things that it can be hard to keep track of them all. Just think about drawing the database design's ER diagram with this single table connected to dozens of other tables.

Tying this table to a whole bunch of others can make it a chokepoint for the entire system. It can also lead to unnecessary redundancy if multiple tables contain fields that have the same domains.

It's better to use separate tables for each of the domains that you need. In this example, just build separate States, Brands, and Media tables. Though this requires more tables, the pieces of the design are simpler, smaller, and easier to understand.

Remember the rule that one table should do one clearly defined thing and nothing else. Although this kind of mishmash table has an easily defined purpose, it does not do just one thing.

Try It Out Mishmash Bash

Consider the following mishmash DomainValues table.

Table	Field	Value
Customers	State	CO
Customers	State	KS
Customers	State	WY
Employees	State	CO
Employees	State	KS
Employees	State	WY
OrderItems	Size	Large

Table	**Field**	**Value**
OrderItems	Size	Medium
OrderItems	Size	Small
Orders	ShippingMethod	Overnight
Orders	ShippingMethod	Priority
Orders	ShippingMethod	Snail Mail

How could you avoid building this mishmash table? What tables would use the new domain tables? To find out:

1. Figure out which records represent similar items.
2. Move similar records into smaller validation tables.
3. Define foreign keys to use the new tables for validation.

How It Works

1. Figure out which records represent similar items.

   This table contains domain values for four tables (Employees, Customers, OrderItems, and Orders) so you might think that you need four domain tables. However, the table really only holds domain values for three kinds of fields: states, sizes, and shipping methods. Those define the groups of similar items so you only need three new domain tables.

2. Move similar records into smaller validation tables.

   The States table contains the following data:

State
CO
KS
WY

   The Size table contains the following data:

Size
Large
Medium
Small

Finally, the ShippingMethod table contains the following data:

ShippingMethod
Overnight
Priority
Snail Mail

3. Define foreign keys to use the new tables for validation.

   Instead of making every table validate domains against the mishmash table, the database now uses the following foreign key constraints:

   - Customers.State = States.State
   - Employees.State = States.State
   - OrderItems.Size = Sizes.Size
   - Orders.ShippingMethod = ShippingMethods.ShippingMethod

Not Enforcing Constraints

When you design a table, you should write down the domains and other constraints for every field. Most database designers can handle that, but it's surprising how often those restrictions don't make it into the actual database.

When you start building the database, go through the list of field constraints and check them off as you implement them. Often these are as simple as making a field required or writing field-level check constraints for a field.

You can rely on middle-tier objects and user interface code to enforce some of these, but the database is the final authority, so why not take advantage of its capabilities? You might want to allow the middle-tier to verify that a flyball team's number of dogs is no more than 3 because it's a possibly changing business rule. It seems unlikely that a team would ever include –1 dogs, however, so let the database enforce that rule at least.

Databases can also often verify field formats. For example, some databases can verify that a phone number string has the format ###-###-####. You may want the user interface to validate this type of format, too, but there's no reason not to let the database do whatever it can to ensure that garbage doesn't slip into the data.

The database is pretty good at enforcing these simple rules. Let it do its job so it can feel appreciated.

Obsession with IDs

ID numbers are nice. They are relatively small and efficient, and it's easy to ensure that they are always unique. However, they don't have any real meaning. You can probably recite your name, address, and phone number easily enough, but do you remember your employee ID, utility company account number, and driver's license number? Unless you have a better memory than mine (which is likely, I have a pretty poor memory) or someone took a shortcut when they defined their keys (in some states, your driver's license number is the same as your Social Security Number), you probably don't remember all of these.

It's okay to have some tables with keys that are not artificial IDs, particularly if the data provides a nice unique key readymade for you. Books have ISBNs (International Standard Book Numbers) that uniquely identify them so, if you're tracking books, use ISBN instead of creating a new meaningless number. Products often have SKU (Stock-Keeping Unit) or product numbers that are just as useful as an artificial number. Even keys that include multiple fields can be perfectly fine and give acceptable performance.

Three obvious times when you really do need to create an artificial primary key are:

- You might need to change the value of the natural key (you shouldn't change primary key values and some databases won't even let you).
- The natural key doesn't guarantee uniqueness.
- Adding an automatically generated surrogate key makes integration with other systems easier.

Before you create a new key field, ask yourself whether it's really necessary.

Try It Out IDs Undone

Consider the following tables.

- Customers with fields FirstName, LastName, Street, and so on.
- ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId.
- Books with fields Title, Author, Year, Pages, Price, and Publisher.
- InventoryItems with fields Name, Vendor, Description, and QuantityInStock.
- WeatherReadings with fields Date, Time, Temperature, and Humidity.

To figure out which of these probably needs an artificial primary key:

1. Look for a natural key.
2. Decide whether you will ever need to change the key's value.
3. Decide whether the key guarantees uniqueness.
4. Use the results from steps 1 through 4 to decide which tables need artificial IDs.

How It Works

1. Look for a natural key.
    - ❑ **Customers with fields FirstName, LastName, Street, and so on:** FirstName/LastName is a natural key.
    - ❑ **ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:** Date/WhitePlayerId /BlackPlayerId is a natural key.
    - ❑ **Books with fields Title, Author, Year, Pages, Price, and Publisher:** Title/Author is a natural key.
    - ❑ **InventoryItems with fields Name, Vendor, Description, and QuantityInStock:** Name/Vendor is a natural key.
    - ❑ **WeatherReadings with fields Date, Time, Temperature, and Humidity:** Date/Time is a natural key.
2. Decide whether you will ever need to change the key's value.
    - ❑ **Customers with fields FirstName, LastName, Street, and so on:** Depending on your application, you may need to change FirstName or LastName values. For example, when some friends of mine got married they decided to change both of their last names to something completely different. This means FirstName/LastName may not make a good primary key.
    - ❑ **ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:** The application should never need to change Date/WhitePlayerId/BlackPlayerId after the data for a match is entered.
    - ❑ **Books with fields Title, Author, Year, Pages, Price, and Publisher:** The application should never need to change Title/Author after the data for a book is entered.
    - ❑ **InventoryItems with fields Name, Vendor, Description, and QuantityInStock:** There's some chance that a vendor will rename a product. In that case, Name/Vendor might not make a good primary key, although you could also treat the renamed product as a new product instead.
    - ❑ **WeatherReadings with fields Date, Time, Temperature, and Humidity:** The application should never need to change Date/Time after the weather data for a reading is entered.
3. Decide whether the key guarantees uniqueness.
    - ❑ **Customers with fields FirstName, LastName, Street, and so on:** FirstName/LastName is probably not enough to guarantee uniqueness. There are just too many John Smiths and Maria Garcias out there.
    - ❑ **ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:** Unless you allow two players to play more than one match at the same time (which is possible), Date/WhitePlayerId/BlackPlayerId is unique.

- **Books with fields Title, Author, Year, Pages, Price, and Publisher:** It's extremely unlikely that the same author (or two authors with the same name) would publish two books with the same title but Title/Author does not absolutely guarantee uniqueness. You could make the key more unique by adding Year, but it's still not an absolute guarantee.
- **InventoryItems with fields Name, Vendor, Description, and QuantityInStock:** It's very unlikely that a particular vendor will have more than one product with the same name (you might not want to do business with such a confused vendor) so Name/Vendor guarantees uniqueness.
- **WeatherReadings with fields Date, Time, Temperature, and Humidity:** Unless you take more than one reading at the same time, Date/Time guarantees uniqueness.

4. Use the results from steps 1 through 4 to decide which tables need artificial IDs.
    - **Customers with fields FirstName, LastName, Street, and so on:** A customer's First-Name/LastName value may change and these fields don't guarantee uniqueness, so this table really needs an ID field.
    - **ChessMatches with fields Date, WhitePlayerId, and BlackPlayerId:** Date/WhitePlayerId/BlackPlayerId is a fine primary key.
    - **Books with fields Title, Author, Year, Pages, Price, and Publisher:** It is unlikely that you would have trouble with Title/Author/Year as a primary key but it's easy enough to use the book's ISBN code to remove all doubt. Adding that data will make it a lot easier to integrate with other systems, such as online bookstores.
    - **InventoryItems with fields Name, Vendor, Description, and QuantityInStock:** Name/Vendor is unlikely to cause trouble but there is a chance. Most products have part numbers, SKUs, UPCs (Universal Product Codes — those values that go with the barcode), or other ID values that make excellent primary keys, so I would add one of those.
    - **WeatherReadings with fields Date, Time, Temperature, and Humidity:** The combination Date/Time guarantees uniqueness and you shouldn't need to change those values later, so this table shouldn't need an artificial key.

Not Defining Natural Keys

Closely related to Obsession with IDs is not defining natural keys. A natural key is a key that you might actually use to search the data. If a table is only there to provide detail for another table, then an ID makes a reasonable link between the two, but if you will be searching a table for natural values such as names or phone numbers, those may make good keys.

For example, suppose the Phones table uses CustomerId to link back to Customers records. Typically to look up a phone number, you will look up the customer's record and then look at its related phone records. It's unlikely that you will know a customer's phone number and need to look up the customer, so you probably don't need to define a key on the phone number table's PhoneNumber field.

In contrast, suppose the Customers table has a CustomerId for a primary key and contains basic customer information such as name and address. What are the chances that you'll need to look up customers by ID? Unless the ID is something special (such as phone number if you're running a phone company), that number is meaningless to mere mortals, so you're more likely to look a customer up by name. You can make that faster by making the name a key. It can't be the primary key because it doesn't guarantee uniqueness and you might need to change a customer's name, but making it a non-primary key will make searches by customer name faster. If that's the search you perform the most, this key is a worthwhile addition to the database.

You might also consider making other fields keys, too. For example, you might want to be able to list customers by city or ZIP Code to prepare mass mailings (postal spam). In that case, making City and ZipCode keys might also be useful if you perform those searches often.

Summary

This chapter described some common mistakes that people make while designing databases. I admit some of these ideas seem a bit wishy-washy for a book about a computer-related subject, but they're pretty important. If you don't pay attention to the ideas described in this chapter, you may end up rediscovering the importance of proper planning, documentation, and testing by painful experience.

Some of these lessons I've learned the hard way, some by studying others' mistakes, and some through research. Take my word for it when I say it's a lot easier (and sometimes humorous) to learn about these issues here instead of through firsthand experience.

In this chapter, you learned the importance of:

- Advanced preparation through thorough requirements gathering.
- Good design practices such as using naming conventions and making a design before building the database.
- Anticipating changes and increased database load.
- Using the database's tools to ensure that values are within their allowed domains.
- Avoiding artificial keys if they are unnecessary and making natural keys even if they cannot be a primary key.

This chapter ends the book's main discussion of general database design topics. The next few chapters explain some practical database implementation issues, paying extra attention to the Microsoft Access and MySQL database management systems.

Before you move on to Chapter 11, 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 your client runs a ski rental shop and he's the most dreaded of all clients: one who thinks he knows something about databases. He has designed a Customers table that looks like this:

Name	Address	City	Zip	Phone1	Phone2	Stuff
Sue Rank	2832 Shush Ln. #2090	Boiler	72010	704-291-2039		Downhill, Snowboarding
Mark Bosc	276 1st Ave	East Fork	72013	704-829-1928	606-872-3982	X-country

The Stuff field contains a list of Downhill, X-country, Snowboarding, and Telemark. Your client plans to get the customer's state from the Zip value when he's sending out mailings.

Your client wants you to build Orders and other tables to go with this one.

For this exercise, explain to your client what's wrong with this table, paying particular attention to the ideas covered in this chapter.

2. Suppose you're building a system to track rentals for a company that owns two Blu-ray rental stores and plans to open a third next year. What special considerations do you need to ponder that might not be as important if the client were, for example, a well-established party rental store. (They rent chairs, punch bowls, big tents, and other stuff for large parties.)

3. What's wrong with an Addresses table that includes these fields:

❑ CustomerId

❑ StreetName

❑ StreetNumber

❑ StreetPrefix

❑ StreetSuffix

❑ StreetPreDirectional

❑ StreetPostDirectional

❑ ApartmentOrSuite

❑ Floor

❑ City

❑ Neighborhood

❑ State

❑ Zip

❑ PlusFour

The Zip and PlusFour fields hold detailed ZIP Code data. For example, if a customer's ZIP+4 is 02536-2918, the Zip field would hold 02536 and the PlusFour field would hold 2918.

4. Consider the relational design shown in Figure 10-3.

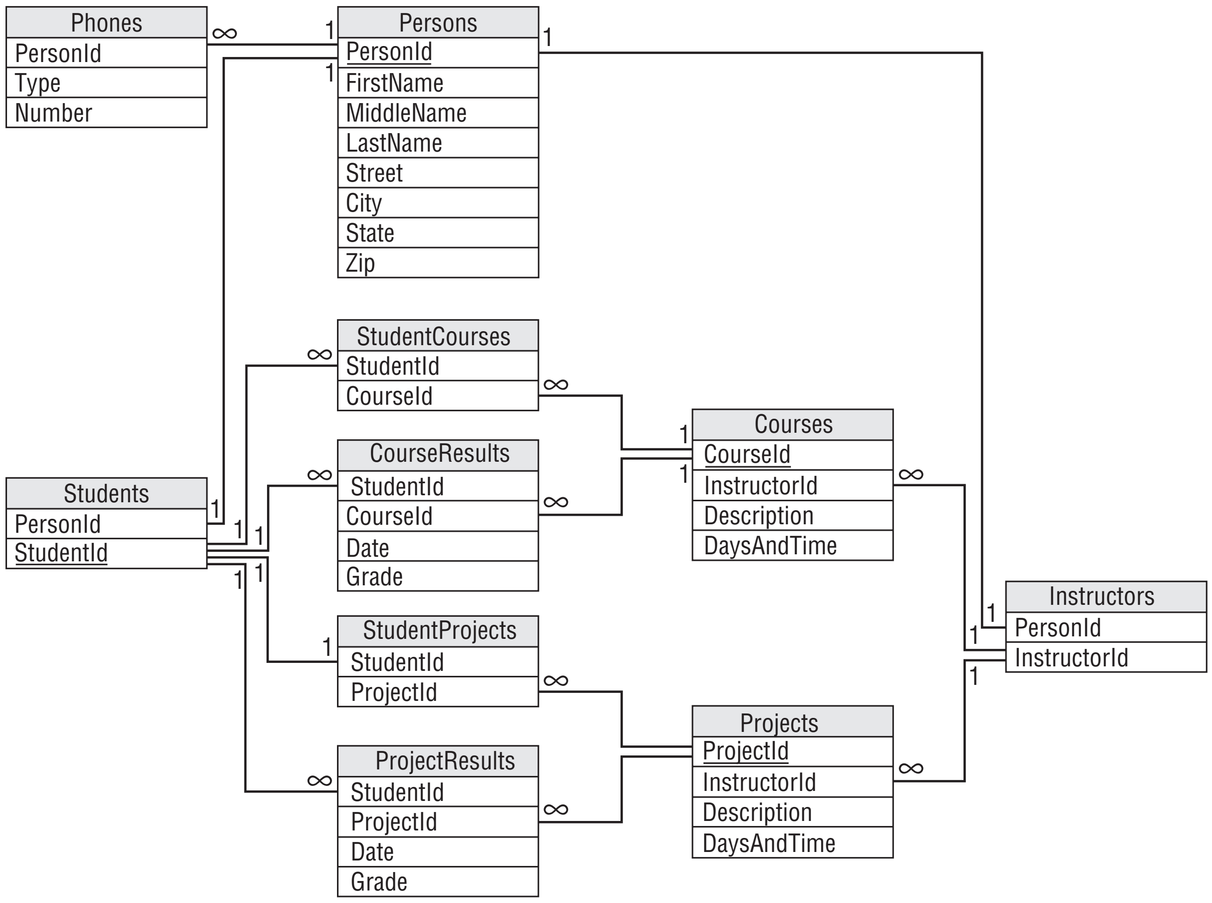

Figure 10-3

List the database constraints that you would place on the fields in this model and explain how you would implement each of those constraints. (Feel free to add new tables if necessary.)