1

Goals of Effective Database Design

Using modern database tools, just about anyone can build a database. The question is, will the resulting database be useful?

A database won't do you much good if you can't get data out of it quickly, reliably, and consistently. It won't be useful if it's full of incorrect or contradictory data. It also won't be useful if it is stolen, lost, or corrupted by data that was only half written when the system crashed.

You can address all of these potential problems by using modern database tools, a good database design, and a pinch of common sense, but only if you understand what those problems are so you can avoid them.

Step one in the quest for a useful database is understanding database goals. What should a database do? What makes a database useful and what problems can it solve? Working with a powerful database tool without goals is like flying a plane through clouds without a compass: you have the tools you need but no sense of direction.

This chapter describes the goals of database design. By studying information containers such as files that can play the role of a database, it defines properties that good databases have and problems that they should avoid.

In this chapter, you learn:

- Why a good database design is important.
- Strengths and weaknesses of different kinds of information containers that can act as databases.
- How computerized databases can benefit from those strengths and avoid those weaknesses.
- How good database design helps achieve database goals.
- What CRUD and ACID are, and why they are relevant to database design.

Understanding the Importance of Design

Forget for a moment that this book is about designing databases and consider software design in general. Software design plays a critical role in software development. The design lays out the general structure and direction that future development will take. It determines which parts of the system will interact with other parts. It decides which subsystems will provide support for other pieces of the application.

If an application's underlying design is flawed, the system as a whole is at risk. Bad assumptions in the design creep into the code at the application's lowest levels, resulting in flawed subsystems. Higher-level systems built on those subsystems inherit the design flaws and soon their code is corrupted, too.

Sometimes a sort of decay pervades the entire system and nobody notices until relatively late in the project. The longer the project continues, the more entrenched the incorrect assumptions become and the more reluctant developers are to suggest scrapping the whole design and starting over. The longer problems remain in the system, the harder they are to remove. At some point, it may be easier to throw everything away and start over from scratch, a decision that few managers will want to present to upper management.

Project Management

A friend of mine who is an engineer was working on a really huge satellite project. After a while, the engineers all realized that the project just wasn't feasible given the current state of technology and the design. Eventually the project manager was forced to admit this to upper management and he was fired. The new project manager stuck it out for a while and then he, too, was forced to confess to upper management that the project was unfeasible. He, too, was fired.

This process continued for a while with a new manager taking over, realizing the hopelessness of the design, and being fired until eventually even upper management had to admit the project wasn't going to work out and the whole thing collapsed.

They could have saved time, money, and several careers if they had spent more upfront time on the design and either fixed the problems or realized right away that the project wasn't going to work and scrapped it at the start.

Building an application is often compared to building a house or skyscraper. You probably wouldn't start building a multibillion dollar skyscraper without a comprehensive design that is based on well-established architectural principles. Unfortunately software developers often rush off to start coding as soon as they possibly can. Coding is more fun and interesting than design is. Coding also lets developers tell management and customers how many lines of code they have written so it seems like they are making progress even if the lines of code are corrupted by false assumptions. Only later do they realize that the underlying design is flawed, the code they wrote is worthless, and the project is in serious trouble.

Now back to database design. Few parts of an application's design are as critical as the database's design. The database is the repository of the information that the rest of the application manages and displays to the users. If the database doesn't store the right data, doesn't keep the data safe, or doesn't let the application find the data it needs, then the application has little chance for success. Here the GIGO (Garbage In, Garbage Out) principle is in full effect. If the underlying data is unsound, it doesn't matter what the application that uses it does; the results will be suspect at best.

For example, imagine that you've built an order tracking system that can quickly fetch information about a customer's past orders. Unfortunately every time you ask the program to fetch a certain customer's records it returns a slightly different result. Though the program can find data quickly, the results are not trustworthy enough to be usable.

Or imagine that you have built an amazing program that can track the thousands of tasks that make up a single complex job such as building a cruise liner or passenger jet. It can track each task's state of completion, determine when you need to order new parts for them to be ready for future phases of construction, and can even determine the present value of future purchases so you can decide whether it is better to buy parts now or wait until they are needed. Unfortunately the program takes hours to recalculate the complex task schedule and pricing details. Though the calculations are correct, they are so slow that users cannot reasonably make any changes. Changing the color of the fabric of a plane's seats or the tile used in a cruise liner's hallways could delay the whole project.

Or suppose you have built an efficient subscription application that lets customers subscribe to your company's quarterly newsletters and data services. It lets you quickly find and update any customer's subscriptions and it always shows the same values for a particular customer consistently. Unfortunately, when you change the price of one of your publications you find that not all of the customers' records show the updated price. Some customers' subscriptions are at the new rate, some are at the old rate, and some seem to be at a rate you've never seen before. (This example isn't as far-fetched as it may seem. Some systems allow you to offer sale prices or special incentives to groups of customers, or they allow sales reps to offer special prices to particular customers. That kind of system requires careful design if you want to be able to do things like change standard prices without messing up customized pricing.)

Poor database design can lead to these and other annoying and potentially expensive scenarios. A good design creates a solid foundation on which you can build the rest of the application.

Experienced developers know that the longer a bug remains in a system the harder it is to find and fix. From that it logically follows that it is extremely important to get the design right before you start building on top of it.

Database design is no exception. A flawed database design can doom a project to failure before it has begun as surely as ill-conceived software architecture, poor implementation, or incompetent programming can.

Information Containers

What is a database? This may seem like a trivial question, but if you take it seriously the result can be pretty enlightening. By studying the strengths and weaknesses of some physical objects that meet the definition of a database, you can learn about the features you might like a computerized database to have.

> **A database is a tool that stores data, and lets you create, read, update, and delete the data in some manner.**

This is a pretty broad definition and it includes a lot of physical objects that most people don't think of as modern databases. For example, an envelope full of business cards, a notebook, a filing cabinet full of

customer records, and your brain all fit this definition. Each of these physical databases has advantages and disadvantages that can give insight into the features you might like in a computer database.

An envelope of business cards is useful as long as it doesn't contain too many cards. You can find a particular piece of data (for example, a person's phone number) by looking through all of the cards. The database is easy to expand by shoving more cards into the envelope, at least up to a point. If you have more than a dozen or so business cards, finding a particular card can be time consuming. You can even rearrange the cards a bit to improve performance for cards you use often. Each time you use a card, move it to the front of the pile. Over time, those that are used most will be in front.

A notebook is small, easy to use, easy to carry, doesn't require electricity, and doesn't need to boot before you can use it. A notebook database is also easily extensible because you can buy another notebook to add to your collection when the first one is full. However, a notebook's contents are arranged sequentially. If you want to find information about a particular topic, you'll have to look through the pages one at a time until you find what you want. The more data you have, the harder this kind of search becomes.

A filing cabinet can store a lot more information than a notebook and you can easily expand the database by adding more files or cabinets. Finding a particular piece of information in the filing cabinet can be easier than finding it in a notebook as long as you are searching for the type of data used to arrange the records. If the filing cabinet is full of customer information sorted by customer name, and you want to find a particular customer's data, you're in luck. If you want to find all of the customers that live in a certain city, you'll have to dig through the files one at a time.

Your brain is the most sophisticated database ever created. It can store an incredible amount of data and it allows you to retrieve a particular piece of data in several different ways. For example, right now you could probably easily answer the following questions about the restaurants that you visit frequently:

- ❑ Which is closest to your current location?
- ❑ Which has the best desserts?
- ❑ Which has the best service?
- ❑ Which is least expensive?
- ❑ Which is the best for a business lunch?
- ❑ Which is your overall favorite?

Your brain provides many different ways you can access the same information about restaurants. You can search the same base of information based on a variety of keys (location, quality of dessert, expense, and so forth). To answer these questions with an envelope of business cards (or restaurant matchbooks), a notebook, or a filing cabinet would require a long and grueling search.

Still your brain has some drawbacks, at least as a database. Most notably it forgets. You may be able to remember an incredible number of things but some of them become less reliable or disappear completely over time. Do you remember the names of all of your elementary school teachers? I don't. (I don't remember my own teachers' names, much less yours!)

Your brain also gets tired and when it is tired it is less accurate.

Although your brain is good at certain tasks such as recognizing faces or picking restaurants, it is not so good at other tasks such as providing an accurate list of every item a particular customer purchased in the last year. Those items have less emotional significance than, for example, your spouse's name, so they're harder to remember.

All of these information containers (business cards, notebooks, filing cabinets, and your brain) can become contaminated with misleading, incorrect, and contradictory information. If you write different versions of the same information in a notebook, the data won't be consistent. Later when you try to look up the data, you may find either version first and you may not even realize there is another version. (Your brain can become especially cluttered with inconsistent and contradictory information, particularly if you listen to politicians during an election year.)

The following section summarizes some of the strengths and weaknesses of these information containers.

Strengths and Weaknesses of Information Containers

By understanding the strengths and weaknesses of information containers such as those described in the previous section, you can learn about features that would be useful in a computerized database. So what are some of those strengths and weaknesses?

The following list summarizes the advantages of some information containers:

- None of these databases require electricity so they are safe from power failures. (Although your brain requires food. As the dormouse said, feed your head.)
- These databases keep their data fairly safe and permanent (barring fires). The data doesn't just disappear.
- These databases (excluding your brain) are inexpensive and easy to buy.
- These databases have simple user interfaces so almost anyone can use them.
- Using these databases, it's fairly easy to add, edit, and remove data.
- The filing cabinet lets you quickly locate data if you search for it in the same way it is arranged (for example, by customer name).
- Your brain lets you find data by using different keys (for example, by location, cost, or quality of service).
- All of these allow you to find every piece of information that they contain, although it may take a while to dig through it all.
- All of these (except possibly your brain) provide consistent results as long as the facts they store are consistent. For example, two people using the same notebook will find the same data. Similarly if you look at the same notebook at a later time, it will show the same data you saw before (if it hasn't been modified).
- All of these except the filing cabinet are portable.
- Your brain can perform complex calculations, at least of a limited type and number.
- All of these provide atomic transactions.

The final advantage is a bit more abstract than the others so it deserves some additional explanation. An *atomic transaction* is a possibly complex series of actions that is considered as a single operation by those who are not involved directly in performing the transaction.

The classic example is transferring money from one bank account to another. Suppose Alice writes Bob a check for $100 and you need to transfer the money between their accounts. You pick up the account book, subtract $100 from Alice's record, add $100 to Bob's record, and then put the notebook down. Someone else who uses the notebook might see it before the transaction (when Alice has the $100) or after the transaction (when Bob has the $100) but they won't see it during the transaction where the $100 has been subtracted from Alice but not yet given to Bob. The office bully isn't allowed to grab the notebook from your hands when you're halfway through. It's an all-or-nothing transaction.

In addition to their advantages, information containers such as notebooks and filing cabinets have some disadvantages. It's worth studying these disadvantages so you can try to avoid them when you build computerized databases.

The following list summarizes some of the disadvantages that these information containers have:

- All of these databases can hold incomplete, incorrect, or contradictory data.
- Some of them are easy to lose or steal. Someone could grab your notebook while you're eating lunch or read over your shoulder on the bus. You could even forget your notebook at the security counter as you dash to catch your flight.
- In all of these databases, correcting large errors in the data can be difficult. For example, it's easy to use a pen to change one person's address in an address notebook. It's much harder to update hundreds of addresses if a new city is created in your area. (This recently happened near where I live.) Such a circumstance requires a tedious search through a set of business cards, a notebook, or a filing cabinet. It may be years before your brain makes the switch completely.
- These databases are relatively slow at creating, retrieving, updating, and deleting data. Your brain is much faster than the others at some tasks but is not good at manipulating a lot of information all at once. For example, how quickly can you list your 20 closest friends in alphabetical order? Even picking your closest friends can be difficult at times.
- Your brain can give different results at different times depending on uncontrollable factors such as your mood, how tired you are, and even whether you're hungry.
- Each of these databases is located in a single place so it cannot be easily shared. Each also cannot be easily backed up so if the original is lost or destroyed, you lose your data.

The following section considers how you can translate these strengths and weaknesses into features to prefer or avoid in a computerized database.

Desirable Database Features

By looking at the advantages and disadvantages of physical databases, you can create a list of features that a computerized database should have. Some of these are fundamental characteristics that any database must have. (''You should be able to get data from it.'' How obvious is that?)

Most of these features, however, depend at least in part on good database design. If you don't craft a good design, you'll miss out on some or all of the benefit of these features. For example, any decent

database provides backup features but a good design can make backup and recovery a lot quicker and easier.

The following sections describe some of the features that a good database system should provide and explain to what degree they depend on good database design.

CRUD

CRUD stands for the four fundamental database operations that any database should provide: Create, Read, Update, and Delete. If you read database articles and discussions on the Web, you will often see people tossing around the term CRUD. (They may be using the term just to sound edgy and cool. Now that you know the term, you can sound cool, too!)

You can imagine some specialized data gathering devices that don't support all of these methods. For example, the black box flight data recorders on airplanes record flight information and later play it back without allowing you to modify the data. In general, however, if it doesn't have CRUD it's not a database.

CRUD is more a feature of databases in general than it is a feature of good database design, but a good database design provides CRUD efficiently. For example, suppose you design a database to track times for your canuggling league (look it up online) and you require that the addresses for participants include a State value that is present in the States table. When you create a new record (the C in CRUD), the database must validate the new State entry. Similarly when you update a record (the U in CRUD), the database must validate the modified State entry. When you delete an entry in the States table (the D in CRUD), the database must verify that no Participant records use that state. Finally when you read data (the R in CRUD), the database design determines whether you find the data you want in seconds, hours, or not at all.

Many of the concepts described in the following sections relate to CRUD operations.

Retrieval

Retrieval is another word for ''read,'' the R in CRUD. The database should allow you to find every piece of data. There's no point putting something in the database if there's no way to get it back later. (That would be a ''data black hole,'' not a database.)

The database should allow you to structure the data so you can find particular pieces of data in one or more specific ways. For example, you should be able to find a customer's billing record by searching for customer name or customer ID.

Ideally the database will also allow you to structure the data so it is relatively quick and easy to fetch data in a particular manner.

For example, suppose you want to see where your customers live so you can decide whether you should start a delivery service in a new city. To get this information, it would be helpful to be able to find customers based on their addresses. Ideally you could optimize the database structure so you can quickly search for customers by address.

In contrast, you probably don't need to search for customers by middle name too frequently. (Imagine a customer calling you and saying, ''Can you look up my record? I don't remember if I paid my bill last

month. I also don't remember my account number or my last name but my middle name is 'Konfused'.'') It would be nice if the common search by address was faster than the rare search by middle name.

Being able to find all of the data in the database quickly and reliably is an important part of database design. Finding the data you need in a poorly designed database can take hours or days instead of mere seconds.

Consistency

Another aspect of the R in CRUD is consistency. The database should provide consistent results. If you perform the same search twice in a row, you should get the same results. Another user who performs the same search should also get the same results. (Of course this assumes that the underlying data hasn't changed in the meantime. You can't expect your net worth query to return the same results every day when stock prices fluctuate wildly.)

A well-built database product can ensure that the exact same query returns the same result but design also plays an important role. If the database is poorly designed, you may be able to store conflicting data in different parts of the database. For example, you might be able to store one set of contact information in a customer's order and a different set of information in the main customer record. Later, if you need to contact the customer with a question about the order, which contact information should you use?

Validity

Validity is closely related to the idea of consistency. Consistency means different parts of the database don't hold contradictory views of the same information. Validity means data is validated where possible against other pieces of data in the database. In CRUD terms, data can be validated when a record is created, updated, or deleted.

Just like physical data containers, a computerized database can hold incomplete, incorrect, or contradictory data. You can never protect a database from users who can't spell or who just plain enter the wrong information, but a good database design can help prevent some kinds of errors that a physical database cannot prevent.

For example, the database can easily verify that data has the correct type. If the user sees a Date field and enters ''No thanks, I'm married,'' the database can tell that this is not a valid date format and can refuse to accept the value. Similarly it can tell that ''Old'' is not a valid Age, ''Lots'' is not a valid Quantity, and ''Confusion'' is too long to be a two-letter state abbreviation (although that value may correctly reflect the user's state of mind).

The database can also verify that a value entered by the user is present in another part of the database. For example, a poor typist trying to enter CO in a State field might type CP instead. The database can check a list of valid states and refuse to accept the data when it doesn't find CP listed. (If the database needs to work with only certain states, you can restrict the list to include only those states and make the test even tighter.)

The database can also check some kinds of conditions on the data. Suppose the database contains a book ordering system. When the customer orders 500 copies of this book (who wouldn't want that many copies?), the database can check another part of the database to see if that many copies are available (most bookstores carry only a few copies of any given book) and refuse the order if there aren't enough copies.

A good database design also helps protect the database against incorrect changes. Suppose a cappuccino machine repair service is dropping coverage for a nearby city. When you try to remove that city from your list of valid locations, the database can tell you if you have existing customers in that city. Depending on the database's design, it could refuse to allow you to remove the city until you apologized to those customers and removed them from the database.

All of these techniques rely on a good, solid database design. They still can't protect you from a user who types first names in the last name field or who keeps accidentally bumping the CAPS LOCK KEY, but it can prevent many types of errors that a notebook can't.

Easy Error Correction

Even a perfectly designed database cannot ensure perfect validity. How can the database know that a customer's name is supposed to be spelled Pheidaux not Fido as typed by the user?

Correcting a single error in a notebook is fairly easy. Just cross out the wrong value and write in the new one.

Correcting systematic errors in a notebook is a lot harder. Suppose you hire a summer intern to go door-to-door selling household products and he writes up a lot of orders for ''Duck Tape'' not realizing that the actual product is ''Duct Tape.'' Fixing all of the mistakes could be tedious and time-consuming. (Of course tedious and time-consuming jobs are what summer interns are for so you can make him fix it himself.) You could just ignore the problem and leave the orders misspelled, but then how would you tell when a customer really wants to tape a duck?

In a computerized database, this sort of correction is trivial. A simple database command can update every occurrence of the product name ''Duck Tape'' throughout the whole system. (In fact, this kind of fix is sometimes too easy to make. If you aren't careful, you may accidentally change the names of *every* product to Duct Tape, even those that were not incorrectly spelled Duck Tape. You can prevent this by building a safe user interface for the database or by being really careful.)

Easy correction of errors is a built-in feature of computerized databases, but to get the best advantage from this feature you need a good design. If order information is contained in a free-formatted text section, the database will have trouble fixing typos. If you put the product name in a separate field, the database can make this change easily.

Though easy corrections are almost free, you need to do a little design work to make them as efficiently and effectively as possible.

Speed

An important aspect of all of the CRUD components is speed. A well-designed database can create, read, update, and delete records quickly.

There's no denying that a computerized database is a lot faster than a notebook or a filing cabinet. Instead of processing dozens of records per hour, a computerized database can process dozens or hundreds per second. (I once worked with a billing center that processed around 3 million accounts every three days.)

Good design plays a critical role in database efficiency. A poorly organized database may still be faster than the paper equivalent but it will be a lot slower than a well-designed database.

Database Design

The billing center I mentioned in the previous paragraph had a simple problem: they couldn't find the customers who owed them the most money. Every three days the database would print out a list of customers who owed money. The list made a stack of paper almost three feet tall. Unfortunately the list was randomly ordered (probably ordered by customer ID or shoe size or something equally unhelpful) so they couldn't figure out who owed the most. The majority of the customers owed only a few dollars — too little to pursue — but a few customers owed tens of thousands of dollars.

We captured this printout electronically and sorted the accounts by balance. It turned out that the really problematic customers only filled a couple of pages and the first five or so customers owed more than all of the others combined.

I didn't include this story just to impress you with my programming prowess (to be completely honest, it was a pretty easy project) but to illustrate how database design can make a big difference in performance. Here a very simple change (which any database should be able to support) made the difference between finding the most troublesome customers in a few seconds or not at all.

Not all changes to a database's design can produce dramatic results, but design definitely plays an important role in performance.

Atomic Transactions

Recall that an atomic transaction is a possibly complex series of actions that is considered as a single operation by those not involved directly in performing the transaction. If you transfer $100 from Alice's account to Bob's account, no one else can see the database while it is in an intermediate state where the money has been removed from Alice's account and not yet added to Bob's.

The transaction either happens completely or none of its pieces happen — it cannot happen halfway.

Atomic transactions are important for maintaining consistency and validity, and are thus important for the R and U parts of CRUD.

Physical data containers such as notebooks support atomic transactions because typically only one person at a time can use them. Unless Derek the office bully grabs the notebook from your hands while you're writing in it, you can finish a series of operations before you let someone else have a turn.

Some of the most primitive kinds of databases, such as flat files and XML files (which are described later in this book) don't inherently support atomic transactions, but the more advanced relational database products do. Those databases allow you to start a transaction and perform a series of operations. You can then either *commit* the transaction to make the changes permanent or *rollback* the transaction to undo them all and restore the database to the state it had before you started the transaction.

These databases also automatically rollback any transaction that is open if the database halts unexpectedly. For example, suppose you start a transaction, take $100 from Alice's account, and then your company's mascot (a miniature horse) walks through the computer room, steps on a power strip, and kills the power to your main computer. When you restart the database (after sending the horse to the HR

department), it automatically rolls the transaction back so Alice gets her money back. You'll need to try the transaction again but at least no money has been lost by the system.

Atomic transactions are more a matter of properly using database features than database design. If you pick a reasonably advanced database product and use transactions properly, you gain their benefits. If you decide to use flat files to store your data, you'll need to implement transactions yourself.

ACID

This section provides some more detail about the transactions described in the previous section rather than discussing a new feature of physical data containers and computerized databases.

ACID is an acronym describing four features that an effective transaction system should provide. ACID stands for Atomicity, Consistency, Isolation, and Durability.

Atomicity means transactions are atomic. The operations in a transaction either all happen or none of them happen.

Consistency means the transaction ensures that the database is in a consistent state before and after the transaction. In other words, if the operations within the transaction would violate the database's rules, the transaction is rolled back. For example, suppose the database's rules say that an account cannot make a payment that would result in a balance less than zero. Also suppose that Alice's account holds only $75. Now you start a transaction, add $100 to Bob's account, and then try to remove $100 from Alice's. That would put Alice $25 in the red, violating the database's rules, so the transaction is canceled and we all try to forget that this ugly incident ever occurred. (Actually we probably bill Alice an outrageous surcharge for writing a bad check.)

Isolation means the transaction isolates the details of the transaction from everyone except the person making the transaction. Suppose you start a transaction, remove $100 from Alice's account, and add $100 to Bob's account. Another person cannot peek at the database while you're in the middle of this process and see a state where neither Alice nor Bob has the $100. Anyone who looks in the database sees the $100 *somewhere*, either in Alice's account before the transaction or in Bob's account afterwards.

In particular, two transactions operate in isolation and cannot interfere with each other. Suppose one transaction transfers $100 from Alice to Bob and then a second transaction transfers $100 from Bob to Cindy. Logically one of these transactions occurs first and finishes before the other starts. For example, when the second transaction starts, it will not see the $100 missing from Alice's account unless it is already in Bob's account.

> **Note that the order in which the transactions occur may make a big difference. Suppose Alice starts with $150, Bob starts with $50, and Cindy starts with $50.**
>
> **Now suppose the second Bob-to-Cindy transaction occurs first. If the transaction starts by removing $100 from Bob's account, Bob is overdrawn, this transaction is rolled back, we assess Bob a surcharge for being overdrawn, and we try to sell Bob overdraft protection for the low, low price of only $10 per month. After all of this, the Alice-to-Bob transaction occurs and we successfully move $100 into Bob's account.**

In contrast, suppose the Alice-to-Bob transaction occurs first. That transaction succeeds with no problem so, when the Bob-to-Cindy transaction starts, Bob has $150 and the second transaction can complete successfully.

The database won't determine which transaction occurs first, just that each commits or rolls back before the other starts.

Durability means that once a transaction is committed, it will not disappear later. If the power fails, when the database restarts, the effects of this transaction will still be there.

The durability requirement relies on the consistency rule. Consistency ensures that the transaction will not complete if it would leave the database in a state that violates the database's rules. Durability means that the database will not later decide that the transaction caused such a state and retroactively remove the transaction.

Once the transaction is committed, it is final.

A high-end database might provide durability through continuous shadowing. Every time a database operation occurs, it is shadowed to another system. If the main system crashes, the shadow database can spring instantly into service.
Other databases provide durability through logs. Every time the database performs an operation, it writes a record of the operation into the log. Now suppose the system crashes. When the database restarts, it reloads its last saved data and then reapplies all of the operations described by the log. This takes longer than restarting from a shadow database but requires fewer resources so it's generally less expensive.

To provide durability, the database cannot consider the transaction as committed until its changes are shadowed or recorded in the log so the database will not lose the changes if it crashes.

Persistence and Backups

The data must be persistent. It shouldn't change or disappear by itself. If you can't trust the database to keep the data safe, the database is pretty much worthless.

Database products do their best to keep the data safe, and in normal operation you don't need to do much to get the benefit of data persistence. When something unusual happens, however, you may need to take special action and that requires prior planning. For example, suppose the disk drives holding the database simply break. Or a fire reduces the computer to a smoldering puddle of slag. Or a user accidentally or intentionally deletes the database. (A user tried that once on a project I was working on. We were not amused!)

In these extreme cases, the database alone cannot help you. To protect against this sort of trouble, you need to perform regular backups.

Physical data containers such as notebooks are generally hard to back up, so they are hard to protect against damage. If a fire burns up your accounts receivable notebook, you'll have to rely on your customers' honesty in paying what they owe you. Though we like customers, I'm not sure most businesses trust them to that extent.

In theory you could make copies of a notebook and store them in separate locations to protect against these sorts of accidents, but in practice few businesses (except perhaps money laundering, smuggling, and other endeavors where it's handy to show law enforcement officials one set of books and the ''shareholders'' another) do.

Computerized databases, however, are relatively easy to back up. If the loss of a little data won't hurt you too badly, you can back up the database daily. If fire, a computer virus, or some other accident destroys the main database, you can reload the backup and be ready to resume operation in an hour or two.

If the database is very volatile or if losing even a little data could cause big problems (how much money do you think gets traded through the New York Stock Exchange in a busy hour?), then you need a different backup strategy. Many higher-end database products allow you to shadow every database operation as it occurs so you always have a complete copy of everything that happens. If the main database is destroyed, you can be back in business within minutes. Some database architectures can switch to a backup database so quickly the users don't even know it's happened.

Backup Plans

It's always best to store backups away from the computer that you're backing up. Then if a really big accident like a fire occurs and destroys the whole building holding the database, the backup is still safe.

I've known of several development groups that stored their backups right next to the computer they were backing up. That guards against some kinds of stupidity (in the teams I've worked on, about once every 10 person-years or so someone accidentally deleted a file that we needed to recover from backups) but doesn't protect against big accidents.

I've also known of companies that had an official backup plan, but once you submitted a backup for proper storage it was shipped off site and it took a long time to get it back if you needed it. A backup doesn't do much good if you can't use it!

In a very extreme example, I had a customer who was concerned that backups were stored only 30 miles from the database. Their thought was that the backups might not be safe in the event of a volcanic eruption or nuclear explosion.

Exactly how you implement database backups depends on several factors such as how likely you think a problem will be, how quickly you need to recover from it, and how disastrous it would be to lose some data and spend time to restore from a backup, but a computerized database gives you a lot more options than a notebook does.

Good database design can help make backups a bit easier. If you arrange the data so changes occur in a fairly localized area, you can back up that area fairly often and not waste time backing up data that changes only rarely.

Low Cost and Extensibility

Ideally the database should be easy to obtain and install, inexpensive, and easily extensible. If you discover that you need to process a lot more data per day than you had expected, you should be able to somehow increase the database's capacity.

Although some database products are quite expensive, most of them have reasonable upgrade paths so you can buy the least expensive license that will handle your needs, at least in the beginning. For example, SQL Server, Oracle, and MySQL provide free editions that you can use to get started building small single-user applications. They also provide more expensive editions that are suitable for very large applications that have hundreds of users.

Installing a database will never be as easy and inexpensive as buying a new notebook, but it also doesn't need to be a time-consuming financial nightmare.

Though expense and capacity are more features of the particular database product than database design, good design can help with a different kind of extensibility. Suppose you have been using a notebook database for a while and discover that you need to capture a new kind of information. Perhaps you decide that you need to track customers' dining habits so you know what restaurant gift certificate to give them on special occasions. In this case, it would be nice if you could extend the database design to hold this extra information.

Good database design can make this kind of extension possible.

Ease of Use

Notebooks and filing cabinets have simple user interfaces so almost anyone can use them effectively. (Although sometimes even they get messed up pretty badly. Should you file ''United States Postal Service'' under ''United States?'' ''Postal Service?'' ''Snail Mail?'')

A computer application's user interface determines how usable it is by average users. User interface design is not part of database design, so you may wonder why ease of use is mentioned here.

The first-level users of a database are often programmers and relatively sophisticated database users who understand how to navigate through a database. A good database design makes the database much more accessible to those users. Just by looking at the names of the tables, fields, and other database entities that organize the data, this type of user should be able to figure out how different pieces of data go together and how to use them to retrieve the data they need. If those sophisticated users can easily understand the database, they can build better user interfaces for the less advanced users.

Portability

A computerized database allows for a portability that is even more powerful than the portability of a notebook. It allows you to access the data from anywhere you have access to the Web *without actually moving the physical database*. You can access the database from just about anywhere while the data itself remains safely at home, far from the dangers of pickpockets, being dropped in a puddle, and getting forgotten on the bus.

In fact, the new kind of portability may be a little too easy. Though someone in the seat behind you on the airplane can't peek over your shoulder to read a computerized data the way he can a notebook (well, he can if you're using your laptop), a hacker located on the other side of the planet may try to sneak into your database and rifle through your customer data while you're asleep.

This leads to the next topic, security.

Security

A notebook is relatively easy to lose or steal but a highly portable database can be even easier to compromise. If you can access your database from all over the world, then so can cyber-banditos and other ne'er-do-wells.

Locking down your database is mostly a security issue that you should address by using your network's and database's security tools. However, there are some design techniques that you can use to make securing the database easier.

Information Theft

There have been a number of spectacular stories of lost or stolen laptops, hard drives, disks, and other media potentially exposing confidential information to bad guys.

- On January 22, 2005, a University of Northern Colorado hard drive containing personal information about 30,000 current and former University employees was apparently stolen.
- On December 22, 2005, a Ford Motor Company computer was stolen containing the names and Social Security Numbers of 70,000 current and former employees. Just three days later, on December 25, 2005, an Ameriprise Financial Inc. laptop containing sensitive information about 260,000 customers was stolen (the laptop was later recovered).
- On June 1, 2006, a laptop containing information about 243,000 `Hotel.com` customers was stolen.
- On January 13, 2007, a North Carolina Department of Revenue computer containing tax information from 30,000 taxpayers was stolen.
- On January 24, 2008, a Fallon Community Health Plan computer containing confidential information about 30,000 patients was stolen.
- Finally, in possibly the biggest data loss to date, on May 3, 2006, a U.S. Department of Veterans Affairs laptop containing information about 28.6 million veterans and active duty personnel was stolen.

I don't mean to single these victims out. This is a big issue and hundreds if not thousands of companies around the world have suffered similar data exposure. The Privacy Rights Clearinghouse Web page, ''A Chronology of Data Breaches'' at `www.privacyrights.org/ar/ChronDataBreaches.htm`, lists incidents totaling more than 230 million exposed records in the United States alone since the site began tracking incidents in 2005.

If you separate the data into categories that different types of users need to manipulate, you can grant different levels of permission to the different kinds of users. Giving users access to only the data they absolutely need not only reduces the chance of a legitimate user doing something stupid or improper, but it also decreases the chance that an attacker can pose as that user and do something malicious. Even if Clueless Carl won't mistreat your data intentionally, an online mugger might be able to guess Carl's password (which naturally is ''Carl'') and try to wreak havoc. If Carl doesn't have permission to trash the accounting data, neither does the mugger.

Yet another novel aspect to database security is the fact that users can access the database remotely without actually holding a copy of the database locally. You can use your palmtop computer to access a database without storing the data on your computer. That means if you do somehow lose your computer, the data may still be safe on the database's computer.

This is more an application architecture issue than a database design issue (don't store the data locally on laptops) but using a database design that restricts users' access to what they really need to know can help.

Sharing

It's not easy to share a notebook or envelope full of business cards among a lot of people. No two people can really use a notebook at the same time and there's some overhead in shipping the notebook back and forth among users. Taking time to walk across the room a dozen times a day would be annoying; express mailing a notebook across the country every day would be just plain silly.

Modern networks can let hundreds or even thousands of users access the same database at the same time from locations scattered across the globe. Though this is largely an exercise in networking and the tools provided by a particular database product, some design issues come into play.

If you compartmentalize the data into categories that different types of users need to use as described in the previous section, this not only helps with security but it also helps reduce the amount of data that needs to be shipped across the network.

Breaking the data into reasonable pieces can also help coordinate among multiple users. When your coworker in London starts editing a customer's record, that record must be locked so other users can't sneak in and mess things up before the edit is finished. Grouping the data appropriately lets you lock the smallest amount of data possible so more data is available for other users to edit.

Careful design can allow the database to perform some calculations and ship only the results to your boss who's working hard on the beaches of Hawaii instead of shipping the whole database out there and making the user's computer do all of the work.

Good application design is also important. Even after you prepare the database for efficient use, the application still needs to use it properly. But without a good database design, these techniques aren't possible.

Ability to Perform Complex Calculations

Compared to the human brain, computers are idiots. It takes seriously powerful hardware and frighteningly sophisticated algorithms to perform tasks that you take for granted such as recognizing faces, speaker-independent speech recognition, and handwriting recognition (although neither the human brain nor computers have yet deciphered doctors' prescriptions). The human brain is also self-programming, so it can learn new tasks flexibly and relatively quickly.

Though a computer lacks the adaptability of the human brain, it is great at performing a series of well-defined tasks quickly, repeatedly, and reliably. A computer doesn't get bored, let its attention wander, and make simple arithmetic mistakes (unless it suffers from the infamous Pentium FDIV bug, the f00f bug, the Cyrix coma bug, or a few others). The point is, if the underlying hardware and software works correctly, the computer can perform the same tasks again and again millions of times per second without making mistakes.

When it comes to balancing checkbooks, searching for accounts with balances less than zero, and performing a host of other number-crunching tasks, the computer is much faster and less error-prone than a human brain.

The computer is naturally faster at these sorts of calculations, but even its blazing speed won't help you if your database is poorly designed. A good design can make the difference between finding the data you need in seconds rather than hours, days, or not at all.

Consequences of Good and Bad Design

The following table summarizes how good and bad design can affect the features described in the previous sections.

Feature	Good Design	Bad Design
CRUD	You can find the data you need quickly and easily. The database prevents inconsistent changes.	You find the data you need either very slowly or not at all. You can enter inconsistent data or modify and delete data to make the result inconsistent. (Your products ship to the wrong address or the wrong person.)
Retrieval	You can find the correct data quickly and easily.	You cannot find the data you need quickly. (Your customer waits on hold for 45 minutes to get a simple account balance.)
Consistency	All parts of the database agree on common facts.	Different pieces of information hold contradictory data. (A customer's bills are sent to one address but late payment notices are sent to another.)
Validity	Fields contain valid data.	Fields contain gibberish. (Your company's address has the State value ''Confusion.'' Although if the database does hold that value, it's probably correct on some level.)

Feature	Good Design	Bad Design
Error Correction	It's easy to update incorrect data.	Simple and large-scale changes never happen. (Thousands of your customers' bills are returned to you because their ZIP Code changed and the database didn't get updated.)
Speed	You can quickly find customers by name, account number, or phone number.	You can only find a customer's record if he knows his 37-digit account number. Searching by name takes half an hour.
Atomic Transactions	Related transactions either all happen or all don't happen.	Related transactions may occur partially. (Alice loses $100 but Bob doesn't receive it. Prepare for customer complaints.)
Persistence and Backups	You can recover from computer failure. The data is safe.	Recovering lost data is slow and painful or even impossible. (You lose all of the orders placed in the last week!)
Low Cost and Extensibility	You can move to a bigger database when your need grows.	You're stuck on a small-scale database. (When your Web site starts getting hundreds of orders per second, the database cannot keep up and you lose thousands per day. Don't we all wish we had this problem!)
Ease of Use	The database design is clear so developers understand it and build a great user interface.	The database design is confusing so the developers produce an ''anthill'' program — confusing and buggy. (I've worked on projects like that and it's no picnic!)
Portability	The design allows different users to download relevant data quickly and easily.	Users must download much more data than they need, slowing performance and giving them access to sensitive data (such as the Corporate Mission Statement, which proves management has no clue.)
Security	Users have access to the data they need and nothing else.	Hackers and disgruntled employees have access to everything.
Sharing	Users can manipulate the data they need.	Users lock data they don't really need and get in each others' way, slowing them down.
Complex Calculations	Users can easily perform complex analysis to support their jobs.	Poor design makes calculations take far longer than necessary. (I worked on a project where a simple change to a data model could force a 20-minute recalculation.)

Summary

This chapter explained the important position that database design plays in application development. If the database design doesn't provide a solid foundation for the rest of the project to build upon, the application as a whole will fail.

This chapter then described physical data containers that can behave as databases. It discussed the strengths and weaknesses of those objects and explained how a computerized database can provide the strengths while avoiding the weaknesses.

In this chapter you learned that a good database provides:

- CRUD
- Retrieval
- Consistency
- Validity
- Easy error correction
- Speed
- Atomic transactions
- ACID
- Persistence and backups
- Low cost and extensibility
- Ease of use
- Portability
- Security
- Sharing
- Ability to perform complex calculations

This chapter used physical objects such as notebooks and filing cabinets to study database goals and potential problems. These physical systems meet some but not all of the database goals fairly effectively.

The next chapter describes several different kinds of computerized databases. It explains which goals each type of database meets and which it does not.

Though this book focuses mostly on relational databases, some of these other kinds of databases are simpler and useful enough for some applications.

Before you move on, however, take a look at the following exercises and test your knowledge of database design goals described in this chapter. You can find the solutions to these exercises in Appendix A.

Exercises

1. Compare this book to a database (assuming you don't just use it as a notebook, scribbling in the margins). What features does it provide? What features are missing?
2. Describe two features that this book provides to help you look for particular pieces of data in different ways.
3. What does CRUD stand for? What do the terms mean?
4. How does a chalkboard implement the CRUD methods? How does a chalkboard's database features compare to those of this book?
5. Consider a recipe file that uses a single index card for each recipe with the cards stored alphabetically. How does that database's features compare to those of a book?
6. What does ACID stand for? What do the terms mean?
7. Suppose Alice, Bob, and Cindy all have account balances of $100 and the database does not allow an account's balance to ever drop below zero. Now consider three transactions: 1) Alice transfers $125 to Bob, 2) Bob transfers $150 to Cindy, and 3) Cindy transfers $25 to Alice and $50 to Bob. In what order(s) can the transactions be executed successfully?
8. Explain how a central database can protect your confidential data.

2

Database Types

Recall the question posed at the beginning of Chapter 1: What is a database? The answer given there was:

> **A database is a tool that stores data, and lets you create, read, update, and delete the data in some manner.**

This broad definition allows you to consider all sorts of odd things as databases including notebooks, filing cabinets, and your brain. If you're flexible about what you consider data, this definition includes even stranger objects such as a chess set (which stores board positions) or a parking lot (which stores car types and positions, although it might be hard for you to update any given car's position without the owner's consent).

This chapter moves into the realm of computerized databases. Relational databases are by far the most commonly used computerized databases today and most of this book (and other database books) focus on them, but it's still worth taking some time first to learn a bit about other kinds of computerized databases that are available. Relational databases are extremely useful in a huge number of situations but they're not the only game in town. Sometimes a different kind of database may make more sense for your particular problem.

Before you start frantically throwing tables together, building indexes, and normalizing everything in sight, it's worth taking some time to study some of the other kinds of databases that are available.

This chapter describes different types of databases including flat files, spreadsheets, hierarchical databases (XML), object databases, and relational databases. Relational databases are the most common of these, but this chapter describes the others and gives some tips on deciding whether one of the others would be more appropriate.

In this chapter, you learn:

- What kinds of databases are most common.
- The strengths and weaknesses of these database types.
- How to decide which kind of database to use.

Why Bother?

There's an expression, ''If all you have is a hammer, everything looks like a nail.'' If the only kind of database you understand is the relational database, you'll probably try to hammer every kind of data into a relational database, and that can sometimes lead to trouble.

Comparing Database Types

I once worked on a fairly large database application with around 40 developers and more than 120,000 lines of code. The program loaded some fairly large relational databases and used their data to build huge tree-like structures. Those structures allowed sales representatives to design and modify extremely complicated projects for customers involving tens of thousands of line items.

The data was naturally hierarchical but was stored in relational databases, so the program was forced to spend a long time loading each data set. Many projects took 5 to 20 minutes to load. When the user made even a simple change to the data, the program's design required it to recalculate parts of the tree and then save the changes back into the database, a process that took another 5 to 30 minutes depending on the complexity of the model. The program was so slow that the users couldn't perform the kinds of experiments they really needed to optimize the projects they were building. You couldn't quickly see the effects of tweaking a couple of numbers here and there.

To make matters worse, loading and saving all of that hierarchical data in a relational database required tens of thousands of lines of moderately tricky code that was hard to debug and maintain.

At one point, I did a quick experiment to see what would happen if the data were stored in an XML database, a database that naturally stores hierarchical data. My test program was able to load and save data sets containing 20,000 items in three or four seconds.

At this point, the project was too big and the design too entrenched to make such a fundamental change. (After that, political pressure within the company pulled the project in too many directions and it eventually shredded like a tissue in a tug-of-war.)

The lesson is clear: before you spend a lot of time building the ultimate relational database and piling thousands of lines of code on top of it, make sure that's really the kind of database you need. Had this project started with an XML database, it probably would have had a simpler, more natural design with much less code and would probably have lasted for many years to come.

The following sections describe some of the most commonly used database types. They are listed more or less in order of increasing complexity, although it is possible to create very complicated flat files or relatively simple hierarchical databases.

Flat Files

Flat files are simply files containing text. Nothing fancy. No **bold**, *italic*, ***different font faces***, or other special font tricks. Just text.

You can add structure to these files, for example by separating values with commas or using indentation to show structure, but the basic file is just a pile of characters. Some structured variations such as INI files and XML files are described later in this chapter.

Text files provide no special features. Flat files don't help you search for data and don't provide aggregate functions such as total, average, and minimum. Writing code to perform any one of those kinds of searches is fairly easy, but it's extra work and providing flexible ad hoc search capabilities is hard.

Programs cannot modify flat files in general ways. For example, you may be able to truncate a file, add data to the end, or change specific characters within the file, but you cannot insert or delete data in the middle of the file. Instead you must rewrite the entire file to make those sorts of changes.

Though flat files don't provide many services, don't scoff at their use. They are extremely simple and easy to understand, so they are a good choice for some kinds of data. You can open a flat file in any text editor and make changes without needing to write a complex user interface.

If a piece of data is relatively simple and seldom changes, a flat file may be an effective, simple way to store the data. For example, a flat file is a fine place to store a message of the day. Each day you can type in one of your favorite obscure quotes for a program to display when it starts. (''The next thing to saying a good thing yourself, is to quote one.'' –Ralph Waldo Emerson.)

Flat files are also good places to store configuration settings. A configuration file lists a series of named values that a program can read when it needs them. Often a program loads its configuration information when it starts and doesn't look at the configuration file again.

> *Lately some programming environments such as Microsoft's Visual Studio have started saving configuration information in XML files instead of flat files. This lets the application store values in a hierarchical arrangement. The section ''XML'' later in this chapter has more to say about XML.*

Flat files work well if:

- Values are fairly small and simple.
- Values don't change too often.
- You want to be able to change values with a simple text editor.
- You want to be able to distribute settings by copying files to new locations.
- You want to keep a simple historical list of previous values, such as a list of previous daily memos or welcome messages.
- You want to use tools to quickly compare two files.

Flat files don't work well if:

- You need to perform complex searches through the values.
- Values change often.
- You don't want others to be able to view and modify the values easily.
- The values are hierarchical.

Two particularly common places to store configuration information are INI files and the Windows system registry. The following sections describe these two approaches.

INI Files

One common type of flat file database is the INI file (INI stands for ''initialization''). An INI file contains section names surrounded by square brackets. Each section can hold any number of setting names and values separated by an equal sign. For example, the following INI file contains configuration values for a fictitious application named RBP (Really Big Project):

```
[WebSites]
VbTips=http://www.vb-helper.com/whats_new.html
Quote=http://www.quotationspage.com/qotd.html
AstroPicture=http://antwrp.gsfc.nasa.gov/apod/
Comic=http://www.userfriendly.org/

[Directories]
Image=C:\RBP Project\Pictures
Text=C:\RBP Project\Documents
Data=C:\RBP Project\DB
```

The file's first section is called `WebSites`. It contains four values named `VbTips`, `Quote`, `AstroPicture`, and `Comic` that contain URLs leading to Web sites that the application might use. (These pages are my Web site's ''what's new'' page, a quote-of-the-day page, the astronomy picture-of-the-day site, and the User Friendly daily comic strip page.)

The file's second section is named `Directories`. It contains three directory paths that the program can use to locate different kinds of files.

When the RBP application starts, it opens this INI file, reads these values into variables, and uses those variables as it runs.

Later, if you need to change any of these settings, you can simply edit the INI file. For example, suppose your data files fill your 250GB C drive. Rather than replacing your C drive with a slightly bigger drive and filling it up in the next few weeks, you decide to add a new G drive that holds 10 petabytes (a petabyte is 1 million gigabytes so this should last you for a while) and move only your data files to that drive. To make the program use the new directory, you only need to change the value of the Directories section's Data setting to:

```
Data=G:\RBP Project\Data
```

Some applications store more volatile settings such as the MRU (Most Recently Used) file list in the File menu. That works if users have separate INI files but doesn't work if they all share the same INI file. To handle both common and individual settings, some programs use one INI file in a shared location to hold shared values and then other INI files in user-specific locations (for example, in each user's My Documents folder) for their personal settings.

Windows System Registry

The Windows system registry is actually not a flat file, although many applications use it as if it were one. The registry is a hierarchical database that holds configuration information for the operating system

and many of the programs installed on the system. It contains information such as the locations of key executable programs and libraries.

The registry is extremely important to the operating system and if you mess it up you could seriously confuse the system. You can even make it unbootable, so it doesn't pay to fool around in there casually. However, some programming languages have tools that make using certain parts of the registry reasonably easy and safe. If you stick to those tools and don't get carried away, you should be able to store values with little risk of a serious meltdown.

The root of the registry contains several *hives* (that's what Microsoft calls the areas in the registry) that define branches for the local computer, users in general, and the current user. Those branches provide places for you to store both global and user-specific settings.

Many applications store shared settings in the `HKEY_LOCAL_MACHINE\SOFTWARE` branch of the registry. For example, the RBP application mentioned in the previous section might store its Text directory setting at the registry path `HKEY_LOCAL_MACHINE\SOFTWARE\RBP\Directories\Text`.

The registry automatically builds a separate `HKEY_CURRENT_USER` hive for each user so many applications store user-specific information there. The GBP application might store a user's color preferences so users who are color-deficient (color-blind) can adjust the colors so they are easy to see. The program can store the color settings in the `HKEY_CURRENT_USER\Software\GBP\Colors` area so different users see a different set of values.

> *Although if you provide this feature, some of the users will spend time fiddling with the colors to match their moods each day. Sooner or later, someone will set his foreground and background colors to black just to see what will happen. He won't be able to see anything and you'll have to fix it. (I knew someone who did this intentionally to her Windows system colors just to see what would happen. It took her most of a day to recover. Curiosity may not kill the programmer but it can sure make things interesting.)*

The registry is hierarchical and you can build branches within other branches, but it really isn't intended for constructing elaborate data hierarchies. It also isn't intended for storing huge amounts of data or data that changes very frequently. It's a good place to store user-specific configuration information such as MRU lists that might change a few times per day, but it's not a good place to store customer orders and minute-to-minute stock prices.

Relational Databases

This book is mostly about relational databases. Chapter 3 provides an introduction to relational databases. This chapter needs to describe them in enough detail for you to decide whether they're the right choice for you.

Without getting into too much detail (I don't want to spoil the next chapter's surprise), a relational database contains tables that hold rows and columns. Each row holds related data about a particular entity (person, vehicle, sandwich, or whatever). Each column represents a piece of data about that entity (name, street address, number of pickles, and so forth).

Sometimes a piece of data naturally has more than one value. For example, a single customer might place lots of orders. To make it easy to add multiple values, those values are stored in a separate table linked to the first by some value that the corresponding records share.

For example, suppose you build a relational database to track your favorite street luge racers. The Racers table stores information about individual racers. Each row corresponds to a particular racer. The columns represent basic information for a racer such as name, age, height, weight, and so forth. A very important column stores each racer's ID number.

Over time, each racer will have lots of race results (although there will probably be lots of blank spots for bale chuckers — see `www.skateluge.com/lugetalk.htm`). You store race results in a separate RaceResults table. Each row records the final standings for a single racer in a single race. The columns record the racer's ID number, the race's name and date, the racer's finishing position, and the points that position is worth for overall ranking.

To find all of the finishing positions and points for a particular racer, you look up the racer's row in the Racers table, find the racer's ID number, and then find all of the rows in the RaceResults table that have this racer ID.

Figure 2-1 shows this simple database design. (No, this is not a finished design nor a very good one. It's just a start to give you the flavor of a relational database. Let's not get ahead of ourselves!)

Racers Table

RacerName	Nationality	RacerId
Michael Serek	Austria	1
Chris McBride	United States	2
Sebastien Tournissac	France	3

RaceResults Table

RaceName	Division	Dates	RacerId	FinishingPosition	Points
Go Fast Speed Days	Pro Classic Luge Mass	9/1/2007-9/2/2007	1	1	450.0024
Rock and Roll	Pro Classic Luge Mass	7/27/2007-7/28/2007	1	1	450.0024
Almabtrieb World Championships	Pro Classic Luge Mass	7/11/2007-7/14/2007	1	6	403.3633
Almabtrieb World Championships	Pro Classic Luge Mass	7/11/2007-7/14/2007	2	24	321.1366
Almabtrieb World Championships	Pro Classic Luge Mass	7/11/2007-7/14/2007	3	2	432.6154
Go Fast Speed Days	Pro Classic Luge Mass	9/1/2007-9/2/2007	2	2	432.6154
Go Fast Speed Days	Pro Classic Luge Mass	9/1/2007-9/2/2007	3	3	424.3687
Top Challenge	Pro Street Luge Mass	8/25/2007-8/26/2007	2	13	0

Figure 2-1

Relational databases have been around for a long time. (Edgar Codd started laying the foundations in 1970.) They are the most commonly used kind of database today and have been for years, so a lot of very powerful companies have spent a huge amount of time building them. All of that means that relational databases have been thoroughly studied and have evolved over time to the point where they are quite useful and effective.

Relational databases provide a number of features that make working with databases such as the street luge database easier. Some of the features they provide include:

- **Data types:** Each column has a particular data type (text, numeric, date, and so forth) and the database will not allow values of other types in a column.
- **Basic constraints:** The database can enforce constraints such as requiring that a luge racer's top speed be between 50 and 250mph (no one with a top speed less than 50 is worth recording) or it can require certain fields.
- **Referential integrity:** The database can prevent you from adding a RaceResults record for a racer who doesn't exist in the Racers table. Similarly, the database can prevent you from modifying a racer's ID if that would leave rows in the RaceResults table with invalid racer IDs, and it could prevent you from modifying a RaceResults row's racer ID to an invalid value.
- **Cascading deletes and updates:** If you delete a racer from the Racers table, the database can automatically delete all of that racer's RaceResults records. Similarly if you change a racer's ID number, the database can update the ID numbers in that racer's RaceResults records.
- **Joins:** The database can quickly gather related records from different tables. For example, it can easily list every racer with his or her corresponding finishing positions sorted alphabetically and by race date.
- **Complex queries:** Relational databases support all sorts of interesting query and aggregation functions such as SUM, AVG, MIN, COUNT, STDEV, and GROUP BY.

Relational databases work well if:

- You need to perform complicated queries and joins among different tables.
- You need to perform data validations such as verifying that related rows in other tables exist.
- You need to allow for any number of values for a particular piece of data (for example, race finishing positions).
- You want to be able to flexibly build new queries that you didn't plan when you started designing the project.

Relational databases don't work well if:

- You need to use a special data topology to perform the application's main function. For example, you can beat a hierarchy or network with a brick until it fits in a relational database but you may get better performance using a more specialized type of database.

Unless you have special needs, relational databases are usually an excellent choice. Hence the need for this book!

Some of the later sections in this chapter discuss variations on relational databases or other kinds of databases that provide relational features.

Spreadsheets

Spreadsheets display rows and columns of data. They allow the user to create formulas that depend on other data in the spreadsheet, make charts and graphs to visualize the data, print the data, and import and export the data in text and other formats. A spreadsheet may also support relatively sophisticated analysis tools such as statistical functions and iterated solution finding (basically making a bunch of guesses to see which ones work best).

Spreadsheets allow you to easily update some or all of the data and they automatically recalculate values that depend on the data you change.

Because many users understand spreadsheets and are comfortable with them, they can perform some of their own analysis, so you may be able to avoid some work generating a zillion different kinds of output.

> *In most of the larger projects I've worked on, we tried to build in ad hoc query tools so the users could define their own reports. That not only lets you save all the time you would have spent building dozens of reports yourself (one application had more than 100 reports), but it also keeps the users busy so they have less time to dream up gratuitous feature change requests while you're trying to implement the basic functionality.*

If these are the sorts of things you need to do with your data, using a spreadsheet may save you a lot of time and trouble building a more complicated database.

However, spreadsheets don't support complex queries. They also don't automatically check the data's integrity, so it's easy for you to enter incorrect or inconsistent values.

Some spreadsheets allow you to write scripting code that can add a lot of features such as integrity checks and complex analysis that isn't provided by the spreadsheet itself. If you're going to go to all that trouble, however, you may as well admit that you need more than the spreadsheet was intended to do and consider using a more powerful database such as a relational database.

> *Many applications provide spreadsheet data as a form of output. They store their data in a relational or other kind of database and then dump results into a spreadsheet format for users to manipulate.*

Spreadsheets work well if:

- ❑ The data fits naturally in a simple tabular format.
- ❑ You need to visualize the data in charts and graphs.
- ❑ The end users are comfortable with spreadsheets.
- ❑ The end users want to be able to experiment with the data on their own.

Spreadsheets don't work well if:

- ❑ You need complex relationships among the values on different worksheets.
- ❑ You need to perform complex calculations that a spreadsheet cannot easily handle.

- You need data validation.
- You need to perform complex queries.
- You need to update large amounts of data automatically.

Hierarchical Databases

Hierarchical data includes values that are naturally arranged in a tree-like structure. One piece of data somehow logically contains or includes other pieces of data.

Files on a disk drive are typically arranged in a hierarchy. The disk's root directory logically contains everything in the file system. Inside the root are files and directories or folders that break the disk into (hopefully) useful categories. Those folders may contain files and other folders that further refine the groupings.

The following listing shows a tiny part of the folders that make up a file system. It doesn't list the many files that would be in each of these folders.

```
C:\
    Documents and Settings
        Administrator
        All Users
        Ben Grim
        Groo
        Rod
    Temp
        Art
        Astro
    Windows
        Config
        Cursors
        Debug
        system
        system32
            1025
            1031
            1040 short form
            1099 int
```

The disk's root directory is called `C:\`. It contains the `Documents and Settings`, `Temp`, and `Windows` directories. `Documents and Settings` contains folders for the administrator and all users in general, in addition to folders for the system's other users.

The `Temp` directory contains temporary files. It contains `Art` and `Astro` folders that hold temporary files used for specific purposes.

The `Windows` directory contains various operating systems files that you should generally not mess with.

If your file system is designed logically, you should be able to tell from a file's position in the hierarchy what its purpose is. If you found the file `iss_sts122.jpg` in the folder `C:\Temp\Astro`, you

could guess that this was a temporary astronomy image. (If you know your astronomy, you might also guess that it is a picture of the International Space Station taken on Space Shuttle mission TST-122. See `antwrp.gsfc.nasa.gov/apod/image/0803/iss_sts122.jpg`.)

Many other kinds of data can also be arranged hierarchically. Figure 2-2 shows a business organization chart that is arranged hierarchically. The lines indicate which people report to which others.

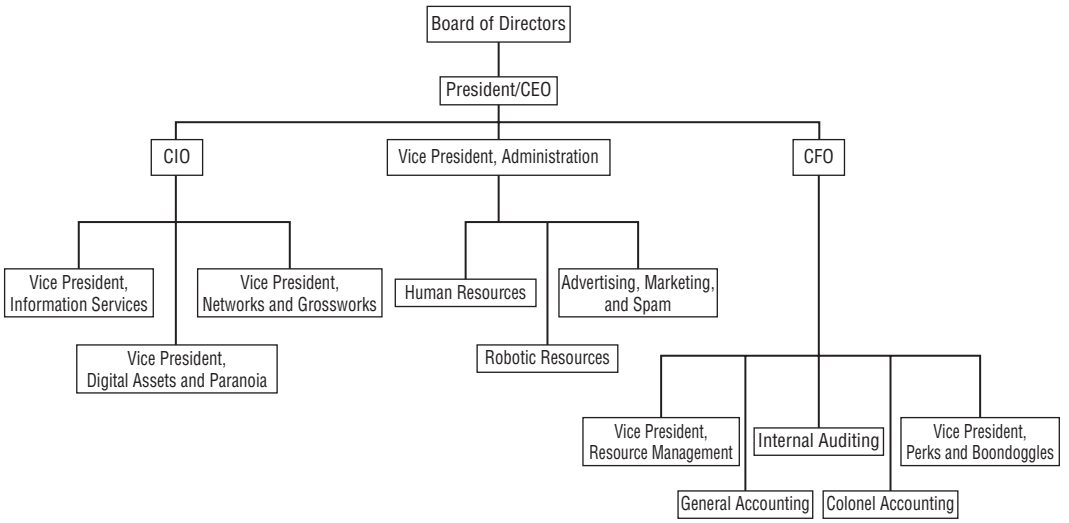


Figure 2-2

Figure 2-3 shows the same information in a slightly different format. This version is arranged more vertically in a way similar to that used by Windows Explorer to show a disk's file system.

In a pipe system, typically big pipes feed into smaller ones to form a hierarchy. Water flows from a treatment plant to large distribution pipes that break into smaller and smaller pipes that eventually feed into houses, bookstores, and coffee houses.

Similarly, electricity flows from a power plant across high-voltage long-distance transmission lines at a few hundred thousand volts (there's less power loss at higher voltages). Next a transformer lowers the voltage to 13,800 or so volts for more local transport. Some is used by factories and large businesses. The rest moves through more transformers that reduce the voltage to 110 or 220 volts (in the United States anyway) for use by your latte machine and desktop computer. (It doesn't even stop there. Your computer again reduces the voltage to 5 volts or so to power your USB plasma ball and missile launcher with Web camera.)

Some other examples of data that you can arrange hierarchically include a family tree tracing your ancestors back in time (two parents, who each have two parents, who each have two parents, and so forth), the parts of any complicated object (a computer has a keyboard, mouse, screen, and system box; the system box includes a fan, power supply, peripherals, and a motherboard; the motherboard includes a chip, heat sink, memory, and so forth), and order tracking information (customers have orders; orders have basic information such as dates and addresses, in addition to order items and possibly sub-orders; order items have an inventory item and quantity; inventory item has description, part number, price, and in some applications sub-items).

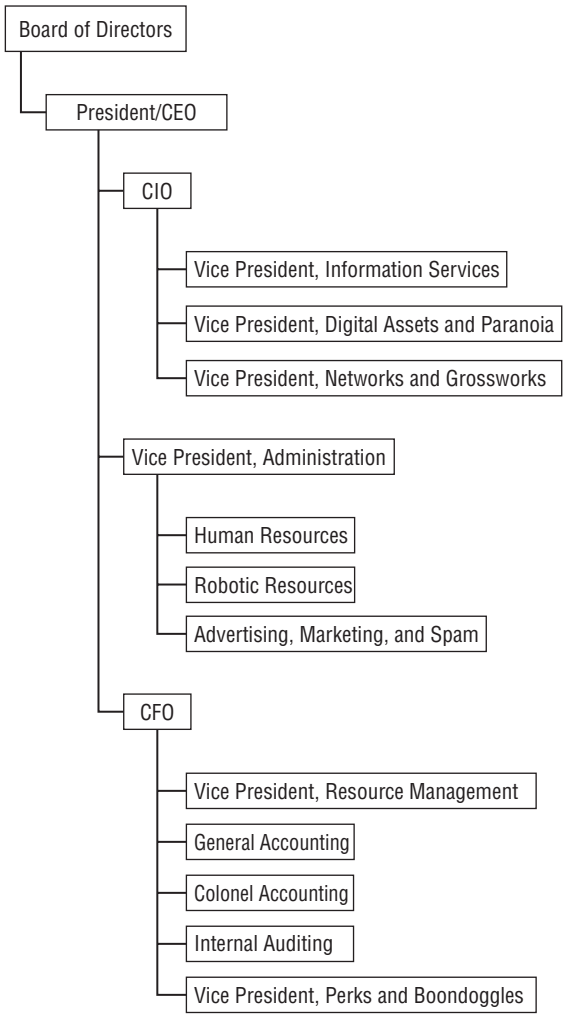


Figure 2-3

You can even think of the information in this book hierarchically. (It's made of chapters that contain paragraphs and sections; sections contain paragraphs and sub-sections; paragraphs contain sentences, which contain words, which contain characters.)

A hierarchical database stores these kinds of data in a way that makes it relatively easy to manipulate the data hierarchically. For example, it may be easy to add a new branch to the tree of data at a particular point. It may be easy to enumerate the ''children'' of a particular location in the tree. Or it may be easy to search the ''ancestors'' or ''descendants'' of a particular piece of data. (For example, in an organizational chart, you might want to list every employee in the Human Resources part of the hierarchy.)

At the same time, a hierarchical database may not support other operations as well as hierarchical operations. For example, it may be hard to search for every employee who filled in more than 60 hours on last

week's timesheet (so you can buy these people cots to put in their offices so they can work even more). If this is a common search, a relational database could use an index to find these employees very quickly. A straightforward hierarchical database would need to examine every employee's data individually.

Hierarchical databases work well if:

- The data is naturally hierarchical.
- You need to perform operations that take advantage of the hierarchical structure.

Hierarchical databases don't work well if:

- The data is not naturally hierarchical.
- You need to perform complex calculations or searches that do not use the hierarchical structure.
- You need complex data validation.
- You need to update large amounts of data automatically.

In the past few years, the XML hierarchical data format has come into widespread use. XML is not actually a database; it's just a text-based method for storing hierarchical data. Although XML is not a database by itself, it's useful and common enough to deserve more in-depth coverage, so the following section provides a brief introduction to XML.

XML

XML (eXtensible Markup Language) is a language for storing hierarchical data. XML itself doesn't provide any tools for building, searching, updating, validating, or otherwise manipulating data and anyone who tells you otherwise is trying to sell you something.

However, XML is a fairly useful format for storing, transferring, and retrieving hierarchical data, and there are several common tools that can make working with XML files easy. This book doesn't explain everything there is to know about XML files. The following sections just provide an overview of XML to help you recognize when an XML database might be a better choice than other kinds of databases. Several other books cover XML in excruciating detail.

XML Basics

An XML file is a relatively simple text file that uses special tokens to define a structure for the data that it contains. People often compare XML to the Web language HTML (HyperText Markup Language) because both use tokens surrounded by pointy brackets, but the two languages have several large differences.

One major difference between XML and HTML is that XML is extensible (the X isn't part of the name just to sound edgy and cool). HTML commands are predefined by the language specification and if you try to invent new ones it's unlikely that a typical browser will know what to do with them. In contrast, XML defines some syntax and options but you get to make up the tokens that contain the data as you go along. All you need to do is start using a token surrounded by pointy brackets. You follow the token by whatever data it should contain and finish with a closing token that is the same as the opening token except it starts with a slash.

For example, the following text shows a single XML token called Name with value Rod Stephens:

```
<Name>Rod Stephens</Name>
```

Programs that read XML ignore whitespace (non-printing characters such as spaces, tabs, and carriage returns) so you can use them to make the data more readable. For example, you can use carriage returns and tabs to indent the data and show the hierarchical structure.

You can make new tokens at any time. For example, the following code shows a `Person` element that includes three fields called `FirstName`, `LastName`, and `NetWorth`. The text uses carriage returns and indentation to make the data easy to read:

```
<Person>
    <FirstName>Rod</FirstName>
    <LastName>Stephens</LastName>
    <NetWorth>$16.32</NetWorth>
</Person>
```

A second important way in which XML and HTML differ is that XML is much stricter about properly nesting and closing opened tokens. For example, the HTML `<P>` command tells a browser to start a new paragraph. Because this command cannot contain any text, there's no need to end it with a closing `</P>` token. The browser just assumes that the `<P>` token immediately ends with a corresponding `</P>` tag. Similarly, a browser assumes an immediate closing tag for a horizontal rule `<HR>` element, and assumes a closing tag for a list item element `<LI>` when it encounters another `<LI>` element or a list ending tag such as `</OL>` or `</UL>`.

In XML every opening token must have a corresponding closing token. (However, XML does allow you to use a shorthand syntax for tokens that immediately open and then close. Just put a slash before the closing pointy bracket as in `<Closed />`.)

XML requires that elements be properly nested. One element may completely contain another, but they may not overlap so one contains only part of another.

For example, the following text includes a `FirstName` element. While that element is open, the text defines a `LastName` element but the `FirstName` element closes before the `LastName` element does. (The indentation makes the overlap easier to see.) This violates XML's nesting rules, so this is not a properly formed piece of XML:

```
<Person>
    <FirstName>Rod
        <LastName>Stephens
    </FirstName>
        </LastName>
    <NetWorth>$16.32</NetWorth>
</Person>
```

An XML file can define *attributes* for an element. For example, in the following XML code, the `Person` element has an attribute named `profession` with value `Dilettante`:

```
<Person Profession="Dilettante">
    <FirstName>Rod</FirstName>
```

```
    <LastName>Stephens</LastName>
    <NetWorth>$16.32</NetWorth>
</Person>
```

You can enclose a comment in an XML file by starting it with the characters `<!--` and ending it with the characters `-->`. For example, the following XML code adds a comment to the previous code:

```
<!-- The book's author -->
<Person Profession="Dilettante">
    <FirstName>Rod</FirstName>
    <LastName>Stephens</LastName>
    <NetWorth>$16.32</NetWorth>
</Person>
```

The final XML rule covered here is that the file must have a single root element that contains all other elements. This makes the file an absolutely pure, true hierarchy of data. Actually, the file can also begin with an optional XML declaration that gives the XML version.

The following text shows a slightly more elaborate XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ClassSchedule>
    <Class Name="Ascension for Beginners" Room="Atrium">
        <!-- Note: Requires Falling 101. -->
        <Instructor>Peter Parker</Instructor>
        <Students>
            <Student>
                <FirstName>Ben</FirstName>
                <LastName>Breaker</LastName>
            </Student>
            <Student>
                <FirstName>Carla</FirstName>
                <LastName>Crash</LastName>
            </Student>
            <Student>
                <FirstName>Dirk</FirstName>
                <LastName>Drop</LastName>
            </Student>
        </Students>
    </Class>

    <Class Name="Advanced Pyrotechnics" Room="Field 3">
        <!-- Note: Requires fire-retardant suit. -->
        <Instructor>Johnny Storm</Instructor>
        <Fees Materials="$45" />
        <Students>
            <Student>
                <FirstName>Erica</FirstName>
                <LastName>Enflame</LastName>
            </Student>
            <Student>
                <FirstName>Frank</FirstName>
                <LastName>Flammable</LastName>
                <NickName>Flamb&#xE9;</NickName>
```

```
            </Student>
        </Students>
    </Class>
</ClassSchedule>
```

This file begins with an XML declaration indicating that it uses XML version 1.0 and the UTF-8 character encoding. It then starts a `ClassSchedule` element that holds all of the document's other content.

The `ClassSchedule` element contains two `Class` elements. Those elements have `Name` and `Room` attributes that give the class's name and location.

The `Class` elements contain `Instructor` and `Fees` elements that define basic information about the classes. Each also includes a `Students` element that contains information about all of the students enrolled in the class. The detailed student information is contained in `Student` elements that hold `FirstName` and `LastName` elements.

Note that the elements need not contain exactly the same kinds of content. For example, the second class contains a `Fees` element but the first does not. Similarly, the final `Student` element contains a `NickName` element but none of the other `Student` elements do. (The text `&#xE9;` in that value makes the `NickName` data include the character with Unicode hexadecimal value E9. That's the character ''e'' with an acute accent: é.)

Because you can make up XML elements as you go along, they allow more flexibility than some other kinds of databases. A relational database, for example, defines exactly what fields are contained in every record in a table. In an XML file, you can add new elements at any point in the file. The XML file's elements provide self-documenting names (if you give your elements reasonable names and not just ''e1'' and ''N32''). This kind of flexible, self-describing database is called *semi-structured*.

XML schema files allow you to provide some validation. For example, they let you indicate that a particular element must contain certain other elements, that an element must contain a date or number, or that an element is required.

XML Structures

In practice I typically see XML files used most often in one of three ways.

First, XML files are hierarchical so it's natural to use them to hold hierarchical data. It's straightforward to map purely hierarchical data such as a simple family tree or organizational chart into an XML file.

Second, XML files are often used to hold table-like data. The basic structure closely follows the structure of a relational database. The root element holds several table elements. Each of those elements holds ''records'' that hold ''fields.''

For example, the following XML document holds data about a simple company's customers and their orders:

```
<AllData>
    <Customers>
        <Customer ID="1">
            <FirstName>Alfred</FirstName>
```

```
            <LastName>Gusenbauer</LastName>
        </Customer>
        <Customer ID="2">
            <FirstName>David</FirstName>
            <LastName>Thompson</LastName>
        </Customer>
        <Customer ID="3">
            <FirstName>Alberto</FirstName>
            <LastName>Selva</LastName>
        </Customer>
    </Customers>
    <Products>
        <Product ID="273645" Description="Toothbrush" Price="$1.95" />
        <Product ID="78463" Description="Pencil" Price="$0.15" />
        <Product ID="48937" Description="Notepad" Price="$0.75" />
    </Products>
    <CustomerOrders>
        <CustomerOrder Date="12/27/2008" CustomerId="2">
            <Item ID="1" ProductId="78463" Quantity="12" />
            <Item ID="2" ProductId="48937" Quantity="2" />
        </CustomerOrder>
    </CustomerOrders>
</AllData>
```

The file starts with an `AllData` root element. That element contains three more elements that define table-like structures holding customer, product, and customer order information.

Each of these ''tables'' defines ''records.'' For example, the `Customers` element includes Customer ''records'' that hold `FirstName` and `LastName` values.

This XML document uses ID numbers to link records in different ''tables'' together. In this example, the single `CustomerOrder` element represents an order placed by customer 2 (David Thompson) who ordered 12 items with ID 78463 (pencils) and 2 items with ID 48937 (notepads).

The third XML file structure I've seen regularly is a simple list of values. The following XML document uses this structure to hold configuration settings for an application:

```
<Settings>
    <NormalColor>Black</NormalColor>
    <WarningColor>Green</WarningColor>
    <ErrorColor>Yellow</ErrorColor>
    <PanicSound>panic.wav</PanicSound>
    <BugEmail>bugs@panic.com</BugEmail>
</Settings>
```

This kind of XML file gives a little more structure than a flat text file used to hold settings and lets a program use XML tools to easily load and read setting values.

This flat structure is also useful when each XML document corresponds directly to some sort of object that a program will use. For example, the following XML file defines a letter. A program could load this data and use its fields to print and mail the letter.

```
<Letter>
    <ToName>Hulk Hogan</ToName>
    <ToStreet>2615 Grappler St, #12</ToStreet>
```

```
    <ToCity>Gripper</ToCity>
    <ToState>CA</ToState>
    <FromName>Yokozuna Hakuho</FromName>
    <Body>
Respected Sir,

Regarding your challenge: Bring it! Your dojo or mine?

Sincerely,
    </Body>
</Letter>
```

The following code shows the same data but in a more structured format:

```
<Letter>
    <To>
        <Name>Hulk Hogan</Name>
        <Address>
            <Street>2615 Grappler St, #12</Street>
            <City>Gripper</City>
            <State>CA</State>
        </Address>
    </To>
    <From>Yokozuna Hakuho</From>
    <Body>
Respected Sir,

Regarding your challenge: Bring it! Your dojo or mine?

Sincerely,
    </Body>
</Letter>
```

This version creates a `To` element that includes all of the information about the letter's recipient. The `To` element contains an `Address` element that holds the recipient's address information. You could add similar information for the sender.

XML Summary

XML files are hierarchical so they are a natural choice for storing hierarchical data. Though you can store hierarchical data in other types of databases, they are unlikely to be able to re-create the hierarchical object structure as quickly as XML tools can. (See the story in the ''Why Bother?'' section at the beginning of this chapter.)

XML files allow you to create elements within other elements just about anywhere you like, so they are semi-structured. This can be convenient if you're not sure of the data's exact format ahead of time. For example, you could easily add extra `To`, `Cc`, or `Bcc` elements to the previous letter example even if you didn't realize you would need them when you wrote the original letter. (Of course, the program that prints the letter may need some modifications to use the new fields but at least you can store valid data.)

Because XML files are plain old text files, they have some of the limitations of text files. In particular, you cannot add, delete, or modify data in the middle of an XML file. To update an XML file, a program typically reads the file into memory, makes its changes, and then writes the result back into the file.

This read-modify-write nature means XML documents are not great multi-user databases. An XML document works fine if many users need to read it but it's harder to allow them to update the file without interfering with each other.

Note that recent versions of some other kinds of databases provide XML support. For example, Excel workbooks can save their data in XML files. SQL Server and Oracle can execute queries to extract data and then return the result in an XML format for the program to manipulate or save into a file.

XML files work well if:

- The data is naturally hierarchical.
- Available XML tools provide the features you need.
- You want the kinds of validation that schema files can provide.
- You want to import and export the data in products that understand XML.

XML files don't work well if:

- You use non-hierarchical data such as networks (described in the following section).
- You need more complex data validation than schema files can provide.
- You need to perform relational rather than hierarchical queries.
- The database is very large so rewriting the entire file to update a small bit of data in the middle is cumbersome.
- You need to allow multiple users to frequently update the database without interfering with each other.

You can find lots of free tutorials covering XML and its related technologies such as XSL, XSLT, XPath, XQuery, and others on the W3 Schools Web site (`www.w3schools.com`*).*

Network

A network contains a collection of *nodes* that are connected by *links*. The nodes and links can represent all sorts of things such as telephone lines, streets, airline routes, and electrical circuits. Links can be unordered (you can travel either way across a link) or ordered (each link is one-way).

Figure 2-4 shows a simple ordered street network. The numbers on the links represent the average time in seconds to cross the link. The letters on the nodes are just there for identification.

A typical problem for this street network might be to find the shortest route from the police station at node A to the donut store at node D. The police will be using their lights and sirens so you don't need to worry about turn penalties (a common feature in shortest path algorithms makes it take longer to turn than to go straight). See if you can find the solution.

Often what appears to be a hierarchical database is really a network. For example, Figure 2-2 shows an idealized corporate organizational chart where lines indicate which people report to which others.

In practice, organizational charts are often more complex and convoluted. Many companies practice ''matrix management'' where employees may work in more than one department and have several

managers for different purposes. Sometimes a person who normally reports to one superior also reports to someone else, either temporarily for the duration of a special project or permanently if more than one executive shares the same area of interest.

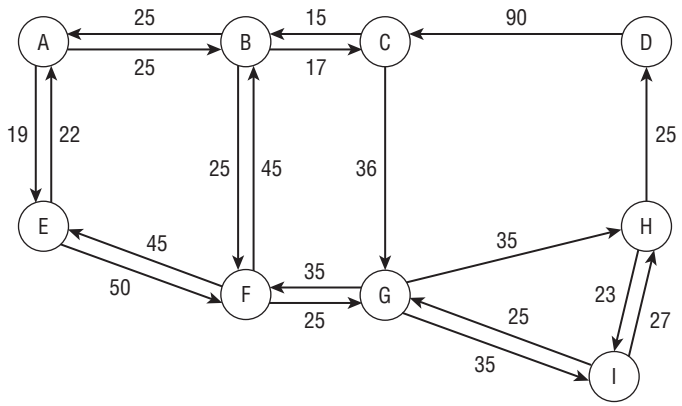


Figure 2-4

Figure 2-5 shows the organizational chart from Figure 2-2 with a few modifications. Here dashed lines indicate that the Robotics Resources director also reports to the CIO, and that the Vice President of Perks and Boondoggles also reports to the President, in addition to reporting through the normal chain of command.

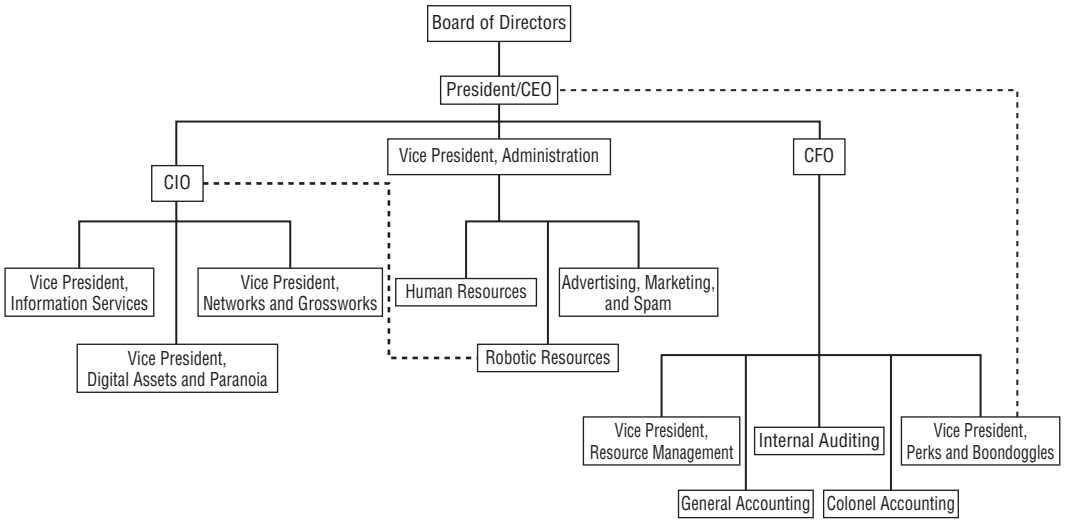


Figure 2-5

Some operating systems allow you to create links from one part of the file system to another. This lets you create a folder that seems to be inside one directory when really entering it warps you to a completely different part of the file system. In that case, the file system isn't really a hierarchy any more. Instead of a

tree, it's more like a bush with grafts and intertwined branches that make strange backwards connections like some sort of alien hybrid, or the organizational chart shown in Figure 2-5. (Real organizational charts often look like the products of some bizarre alien intelligence or plates of spaghetti.)

Network databases are uncommon, although network data structures are very useful for operational algorithms such as shortest path finding, task scheduling, and network flow (think water flowing through pipelines).

Some XML tools allow you to easily save and restore networks in addition to hierarchical data. For example, Microsoft's Visual Studio programming languages C# (pronounced ''C sharp'') and Visual Basic include tools that can save and restore networks in XML files. The resulting files use automatically generated ID numbers to link nodes together and they can be hard to read, but if you only need to save and restore network structures they can be quite handy.

When you use a network file to store data, the program does all of the work. The file itself provides no special features.

Network files work well if:

- ❑ The data is naturally a network (or almost a hierarchy).
- ❑ You need to perform network operations on the data such as finding shortest paths or calculating network flows.
- ❑ You don't need to perform complex queries on the data.

Network files don't work well if:

- ❑ The data does not represent a network.
- ❑ You need to validate the data.
- ❑ You need to perform queries on the data.
- ❑ You need to allow multiple users to frequently update the data without interfering with each other.

Object

Modern programming languages are object-oriented. They use programming abstractions called objects to represent items such as customers, orders, penny stocks, and betting slips.

An *object database* manages objects. It provides some sort of query syntax for retrieving objects from the database. It also provides tools for saving changes to an object back into the database.

Object databases also provide some useful concurrency features. For example, if two users of a program need to work with an object representing the November 12th episode of the television show *Deal or No Deal*, the database gives them the same logical object. If the two users access the object simultaneously, the database referees so the users don't interfere with each other.

Object databases are also sometimes called object-oriented databases, object database management systems (ODBMS), and object stores. Some developers make a distinction among these different terms but at this level they're close enough to the same thing.

Object databases work well if:

- Your programming environment and architecture favors using objects.
- You don't need to perform complex queries on the data (which tend to slow these databases down considerably).

Object databases don't work well if:

- Your program needs to interact with external tools where storing the data in a more common format such as a relational database is an advantage.
- You need to perform complicated queries that will execute faster in a relational database.
- You aren't using an object-oriented language (for example, if a Microsoft Access database can do everything you need without any programming).
- You need to perform data validations that the object database cannot provide.

Object-Relational

An object-relational database (ORD) or object-relational database management system (ORDBMS) is a relational database that provides extra features for integrating object types into the data. Like a relational database, it can perform complex queries relatively quickly. Like an object database, it uses some special syntax to simplify the creation of objects.

Over time, many of the features originally designed for use by object-relational databases have been added to relational databases.

A closely related concept is the *object-relational mapping* system. An object-relational mapping system provides a layer between the object-oriented code and a relational database to convert between objects and relational data. If this layer does a good job of separating the objects and the database, programmers and database developers can ignore the details of each others' work. This lets them work more independently, makes them more productive, and makes it easier for either group to accommodate changes in the other group's work. (It may also help keep them from getting into brawls during project get-togethers.)

Object-relational databases and object-relational mappings work well if:

- Your programming environment and architecture favors using objects.
- You need to perform complicated relational-style queries.
- You need to perform relational-style data validations.
- Your program needs to interact with external tools where storing the data in a common relational format is an advantage.
- You have separate programmers and database developers so maintaining a strict separation can make the project more manageable.

Object-relational databases and object-relational mappings don't work well if:

- You aren't using an object-oriented language (for example, if a Microsoft Access database can do everything you need without any programming).

Exotic

These kinds of databases are more unusual than those described previously. They tend to be very specialized and work well only for a specific subset of database problems. Some are variations on other, less unusual kinds of databases.

Document-Oriented

A document-oriented database is designed to work with document-oriented applications. A typical document-oriented application allows the user to open a ''document'' that represents something. Usually this is an actual file such as a letter, video clip, or Web page, but it might be something more abstract such as student transcripts that are not actually stored in separate physical documents.

A good example of a document-oriented application might manage the files that make up a Web site. (My VB Helper Web site `www.vb-helper.com` holds more than 5,000 files in a dozen or so directories and keeping track of them all is quite a chore. I should build a document management system!)

Some document-oriented databases are simply constructs within a file system that use directories and subdirectories to hold the files that make up the documents. Unfortunately this kind of file system offers limited tools for sorting, searching, and performing other database-related tasks.

Other document-oriented databases are built as a layer on top of some other database system, for example, a relational database. The database might store the contents of the documents themselves or it might store the documents' locations on disk.

Deductive

A deductive database is one that can make deductions based on rules and facts contained within the database. They are a sort of cross between logic programming and relational databases. Some of these databases allow the programmer to guide the evaluation of a program.

Dimensional

A dimensional database (sometimes called a multi-dimensional database) represents different aspects of data as dimensions rather than as separate tables in a relational database.

You can think of dimensional data as forming a multi-dimensional rectangular box (also called a hypercube or multi-dimensional array) where each dimension represents some important facet of the data. For example, Figure 2-6 shows a three-dimensional picture with Year, Sales Rep, and Product Line as dimensions. Each little cube or *cell* in the larger box contains information relating to a particular selection of the dimensions. In other words, a particular cell would contain information about a selected sales rep's sales for a selected product line in a particular year (Crazy Bob's yo-yo sales for 2008).

Dimensional databases are particularly useful for scrounging through old data looking for patterns and they make useful data warehouses. However, if the data is sparse (a lot of the cells in Figure 2-6 are empty), they can waste a lot of space, so they are not usually appropriate for day-to-day use for new data entry. Truly native dimensional databases (as opposed to a dimensional database built on top of a relational database) may be optimized to handle sparse data and can save space while still providing fast results.

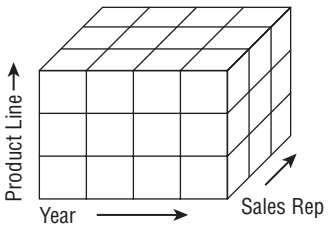


Figure 2-6

Temporal

A temporal database has built-in time information. One of the simplest pieces of temporal data that this kind of database stores is the data's *valid time*: the time during which it is valid.

For example, suppose you build an inventory and sales database for your jewelry store. To make mall visitors think they are getting a good deal, you constantly raise and lower prices. When you're ready for your bi-monthly vacation, you raise prices so the reduced sales load won't overwhelm your brother-in-law Joey (who's otnay ootay ightbray) while you're gone. When you get back, you have your bi-monthly ''Once-In-A-Lifetime Blockbuster Overstock Sale!'' to clear inventory. To be able to later track sales at various prices over time, you need to know the times during which different prices were in effect.

If you were on vacation from April 1 until April 14 and prices were ''normal,'' then those prices have a valid time of those two weeks. If you return and cut prices by 40% from April 15 until your next vacation on May 22, the new prices have a valid time of April 15 through May 22.

For other examples, imagine tracking employee addresses as they move, mileage and fuel use in your fleet of rental scooters, or daily coffee prices. You could store only the latest information in each of these cases, but then you lose the ability to look back in time for important trends.

> *In fact, I've heard a reasonably plausible argument that a database should never delete or overwrite any information. Instead it should just mark the old data as deleted and optionally create a new record for the new data. With disk space as cheap as it is these days (as little as $0.22 or so per GB), it's easy to imagine saving every piece of data, at least for small- and medium-sized databases. (In this case, the database wouldn't need the D in CRUD (Create, Read, Update, Delete) so it goes from being CRUD to being CRU, a fine French wine.)*

It's not too hard to add time fields to other kinds of databases such as relational databases. You'll need a little extra programming to keep track of which values you should use at different times, but this technique can be very useful for data that changes frequently over time.

Summary

Before you launch into an exhausting year-long process of building a relational database, it's worth taking at least a few minutes to decide what kind of database would best fit your needs. Though you can probably use any kind of database for most purposes, some lend themselves more naturally to certain problems than others. Though you can store a data hierarchy or network in a relational database, it may

be a lot faster to use a simpler XML file. Though you certainly can store simple configuration settings in an object-oriented database, a flat text file will do just as well with a lot less trouble.

In this chapter you learned how to pick the database type that will work best with your data. You learned that:

- Flat files are good for storing simple values or complete documents, although they lack features for concurrency and easy updating.
- INI files are good for storing simple values that are easy to look up, although they lack features for concurrency and easy updating.
- The Windows system registry is good for storing simple values that are easy to look up and can handle system-wide or user-specific settings.
- Relational databases are the workhorse of the database world. They allow complex data relations, sophisticated data validation, integrity constraints, cascading updates and deletes, ad hoc queries, and many more useful features.
- Spreadsheets are good for drawing charts and graphs, and are convenient for users who already know how to use them.
- Hierarchical databases are good for storing and manipulating hierarchical data such as organizational charts and family trees.
- XML files are good for storing hierarchical data, although they lack features for concurrency and easy updating.
- Network databases are good for storing network data such as street or telephone networks.
- Object databases are good for integrating programming objects into the database.
- Object-relational databases and object-relational mapping systems combine some of the best features of object-databases and relational databases. An object-relational mapping layered on top of a relational database can provide a useful separation between programmers and database developers.
- Document-oriented databases are useful in document management systems (such as the one I need to build some day to manage my Web site).
- Deductive databases can make logical deductions based on rules and data stored in the database.
- Dimensional databases consider data in hypercubes and make it easy to study data based on dimensional selections (such as sales by a particular representative or during a particular year).
- Temporal databases integrate time with the data so they can record and work with information that changes over time.

Relational databases are by far the most common type of database in use today and they are the topic of most of this book and the majority other database books. The next chapter describes relational databases in greater detail. It explains the basic relational concepts that you need to understand to design and build effective relational databases.

Before you move on to Chapter 3, 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

For the following scenarios, list the type(s) of database that might make good choices for storing the data.

1. A dog breeding database that records the ancestors of a single dog for five generations.
2. A similar dog breeding database that records the ancestors and descendants of a single dog for five generations each way.
3. Application settings that record which windows a user had open and where they were positioned the last time the application was used.
4. Total sales figures by month, arranged to make it easy to see trends graphically.
5. The same as Exercise 4 but the users want to be able to draw similar data for several product lines on the same graph.
6. And they want to be able to print it and tweak the numbers to see what the graph would look like if they exceed expectations next quarter (wishful thinking).
7. A map showing the main vessels and arteries leaving the human heart.
8. A very large amount of sales data including information about customers, orders, inventory items, and sales representatives. You need to be able to perform ad hoc queries.
9. The same as Exercise 8 except your manager just returned from a technical seminar where he learned the phrase ''object-oriented'' and now he's determined to use object-oriented techniques in everything.
10. A simple recipe book. You should be able to find recipes by name, part of meal (entrée, aperitif, dessert, and so forth), or main ingredient.
11. A ''Magic: The Gathering'' game and trading card tracking system. You need to be able to sort cards by their monetary value, number of duplicates, and power in the game. You also want to be able to define ''power decks'' for competition.
12. A DVD, CD, or video collection. You want to be able to search by title, star rating (how good you thought it was), Motion Picture Association of America rating (G, PG, and so forth), actor, or director. And perhaps studio. And genre. And group titles by studio. And anything else you might think of later.
13. A database to hold statistics for your favorite sports teams: football, baseball, polo, hurling, or whatever. You need to be able to find all of the players on a particular team, find players with the best stats (most yards rushing, highest hitting percentage, most chuckers chucked, most . . . uh . . . hurls hurled?).
14. A database to hold the materials I use to write books. This includes author guidelines, chapters, figures, scheduling workbooks, and the like.
15. An inspirational message of the day database. Every day when the application starts, it should display a random message selected from the database.

3

Relational Database Fundamentals

The previous chapters discussed databases in general terms. Chapter 1 explained the general goals of database design. Chapter 2 described some of the many kinds of databases that you might decide to use.

With this chapter the book starts to focus on a particular kind of database: the relational database. Relational databases are very powerful, are the most commonly used kind of database in computer applications today, and are the focus of the rest of this book.

Before you can start learning how to properly design a relational database, you must understand the basic concepts and terms that underlie relational databases.

This chapter provides an introduction to relational databases. It explains the major ideas and terms that you need to know before you can start designing and building relational databases.

In this chapter, you learn about relational database terms such as:

- Table and relation
- Record, row, and tuple
- Column, field, and attribute
- Constraint, key, and index

Finally, you learn about the operations that you can use to get data out of a relational database.

Relational Points of View

Relational databases play a critical role in many important (that is, money-related) computer applications. As is the case whenever enormous amounts of money are at stake, people have spent a huge amount of time and effort building, studying, and refining relational databases. Database researchers usually approach relational databases from one of three points of view.

The first group approaches the problem from a database-theoretical point of view. These people tend to think in terms of provability, mathematical set theory, and propositional logic. You'll see them at the local rave throwing around phrases such as *relational algebra*, *Cartesian product*, and *tuple relational calculus*. This approach is intellectually stimulating (and looks good on a resume) but can be a bit intimidating. These researchers focus on logical design and idealized database principles.

The second group approaches the matter from a less formal ''just build the database and get it done'' point of view. Their terminology tends to be less precise and rigorous but more intuitive. They tend to use terms that you may have heard before such as table, row, and column. These people focus on physical database design and pay more attention to concrete bits-and-bytes issues dealing with actually building a database and getting the most out of it.

The third group tends to think in terms of flat files and the underlying disk structure used to hold data. Though these people are probably in the minority these days, their terms file, record, and field snuck into database nomenclature and stuck. Many of those who still use these terms are programmers and other developers who look at the database from a consumer's ''how do I get my data out of it'' point of view.

These differing points of view have led to several different and potentially confusing ways to view relational databases. This can cause some serious confusion, particularly because the different groups have latched on to some of the same terms but used for different meanings. In fact, they sometimes use the term ''relation'' in very different ways (that are described later in this chapter).

This chapter loosely groups these terms into ''formal'' and ''informal'' categories, where the formal category includes the database theoretical terms and the informal category includes everything else.

This chapter starts with informal terms. Each section initially focuses on informal terms and concepts, and then explains how they fit together with their more formal equivalents.

Table, Rows, and Columns

Informally you can think of a relational database as a collection of *tables*, each containing *rows* and *columns*. At this level, it looks a lot like a workbook containing several worksheets (or spreadsheets), although a worksheet is much less constrained than a database table is. You can put just about anything in any cell in a worksheet. In contrast, every entry in a particular column of a table is expected to contain the same kind of data. For example, all of the cells in a particular column might contain phone numbers or last names.

> *Actually a poorly designed database application may allow the user to sneak some strange kinds of data into other fields. For example, if the database and user interface aren't designed properly, you might be able to enter a string such as ''none'' in a telephone number field. That's not the field's intent, however. In contrast, a spreadsheet's cells don't really care what you put in them.*

The set of the values that are allowed for a column is called the column's *domain*. For example, a column's domain might be telephone numbers, bank account numbers, snowshoe sizes, or hang glider colors.

Domain is closely related to data type but it's not quite the same. A column's *data type* is the kind of data that the column can hold. The data types that you can use for a column depend on the particular

database you are using but typical data types include integer, floating point number (a number with a decimal point), string, and date.

To see the difference between domain and data type, note that street address (323 Relational Rd) and jersey color (red) are both strings. However, the domain for the street address column is valid street addresses, whereas the domain for the jersey color column is colors (and possibly not even all colors if you only allow a few choices). (You can think of the data type as the highest level or most general possible domain. For example, an address or color domain is a more restrictive subset of the domain allowing all strings.)

The rows in a table correspond to column values that are related to each other according to the purpose of the table. For example, suppose you have a Competitors table that contains typical contact information for participants in your First (and probably Last) Annual Extreme Pyramid Sports Championship. This table includes columns to hold competitor name, address, event, blood type, and next of kin as shown in Figure 3-1. (Note that this is not a good database design. You'll see why in later chapters.)

Name	Address	Event	Blood Type	NextOfKin
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Boarding	A+	Art Adventure
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Luge	A+	Art Adventure
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Camel Drafting	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Pyramid Boarding	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Sphinx Jumping	O–	Betty Bold
Cindy Copes	271 Sledding Hill, Ricky Ride CO 80281	Camel Drafting	AB–	John Finkle
Cindy Copes	271 Sledding Hill, Ricky Ride CO 80281	Sphinx Jumping	AB–	John Finkle
Dean Daring	73 Fighter Ave, New Plunge UT 78281	Pyramid Boarding	O+	Betty Dare
Dean Daring	73 Fighter Ave, New Plunge UT 78281	Pyramid Luge	O+	Betty Dare
Frank Fiercely	3872 Bother Blvd, Lost City HI 99182	Pyramid Luge	B+	Fred Farce
Frank Fiercely	3872 Bother Blvd, Lost City HI 99182	Sphinx Jumping	B+	Fred Farce
George Forman	73 Fighter Ave, New Plunge UT 78281	Sphinx Jumping	O+	George Forman
George Forman	73 Fighter Ave, New Plunge UT 78281	Pyramid Luge	O+	George Forman
Gina Gruff	1 Skatepark Ln, Forever KS 72071	Camel Drafting	A+	Gill Gruff
Gina Gruff	1 Skatepark Ln, Forever KS 72071	Pyramid Boarding	A+	Gill Gruff

Figure 3-1

A particular row in the table holds all of the values for a given competitor. For example, the values in the first row (Alice Adventure, 6543 Flak Ter, Runner AZ 82018, Pyramid Boarding, A+, Art Adventure) all apply to the competitor Alice Adventure.

Back in olden times when database developers worked with primitive tools by candlelight, everyone lived much closer to nature. In this case that means they needed to work more closely with the underlying file system. It was common to store data in ''flat'' files without any indexes, search tools, or other fancy modern luxuries. A file would hold the related information that you might represent at a higher level as a table. The file was divided into chunks called *records* that each had the same size and that each

corresponded to a row in a table. The records were divided into fixed-length *fields* that corresponded to the columns in a table.

For example, if you think of the table shown in Figure 3-1 as a flat file, the first row corresponds to a record in the file. Each record contains Name, Address, Event, and other fields to hold the data.

Though relatively few people still work with flat files at this level, the terms file, record, and field are still with us and are often used in database documentation and discussions.

Relations, Attributes, and Tuples

The values in a row are *related* by the fact that they apply to a particular person. Because of this fact, the formal term for a table is a *relation*. This can cause some confusion because the word ''relation'' is also used informally to describe a relationship between two tables. This use is described in the section ''Foreign Key Constraints'' later in this chapter.

The formal term for a column is an *attribute* or *data element*. For example, in the Competitors relation shown in Figure 3-1, Name, Address, BloodType, and NextOfKin are the attributes of each of the people represented. You can think of this as in: ''each person in the relation has a Name attribute.''

The formal term for a row is a *tuple* (rhymes with ''scruple''). This almost makes sense if you think of a two-attribute relation as holding data pairs, a three-attribute relation as holding value triples, and a four-attribute relation as holding data quadruples. Beyond four items, mathematicians would say 5-tuple, 6-tuple, and so forth, hence the name tuple.

Don't confuse the formal term *relation* (meaning table) with the more general and less precise use of the term that means ''related to'' as in ''these fields form a relation between these two tables'' (or ''that psycho is no relation of mine''). Similarly, don't confuse the formal term *attribute* with the less precise use that means ''feature of'' as in ''this field has the 'required' attribute'' (or ''don't attribute that comment to me!''). I doubt you'll confuse the term *tuple* with anything — it's probably confusing enough all by itself.

Theoretically a relation does not impose any ordering on the tuples that it contains nor does it give an ordering to its attributes. Generally the orderings don't matter to mathematical database theory. In practice, however, database applications usually sort the records selected from a table in some manner to make it easier for the user to understand the results. It's also a lot easier to write the program (and for the user to understand) if the order of the fields remains constant, so database products typically return fields in the order in which they were created in the table unless told otherwise.

Keys

Relational database terminology includes an abundance of different flavors of keys. In the loosest sense, a key is a combination of one or more columns that you use to find rows in a table. For example, a Customers table might use CustomerID to find customers. If you know a customer's ID, you can quickly find that customer's record in the table. (In fact, many ID numbers, such as employee IDs, student IDs, driver's licenses, and so forth, are invented just to make searching in database tables easier. My library card certainly doesn't include a 10-character ID number for my convenience.)

The more formal relational vocabulary includes several other more precise definitions of keys.

In general, a key is a set of one or more columns in the table that have certain properties. A *compound key* or *composite key* is a key that includes more than one column. For example, you might use the combination of FirstName and LastName to look up customers.

A *superkey* is a set of one or more columns in a table for which no two rows can have the exact same values. For example, in the Competitors table shown in Figure 3-1, the Name, Address, and Event columns together form a superkey because no two rows have exactly the same Name, Address, and Event values. Because superkeys define fields that must be unique within a table, they are sometimes called *unique keys*.

Because no two rows in the table have the same values for a superkey, a superkey can uniquely identify a particular row in the table. In other words, a program could use a superkey to find any particular record.

A *candidate key* is a minimal superkey. That means if you remove any of the columns from the superkey, it won't be a superkey anymore.

For example, you already know that Name/Address/Event is a superkey for the Competitors table. If you remove Event from the superkey, Name/Address is not a superkey because everyone in the table is participating in multiple events so they have more than one record with the same name and address.

If you remove Name, Address/Event is not a superkey because Dean Daring and his roommate George Foreman share the same address and are both signed up for Pyramid Luge. (They also have the same blood type. They became friends and decided to become roommates when Dean donated blood for George after a particularly flamboyant skateboarding accident.)

Finally if you remove Address, Name/Event is still a superkey. That means Name/Address/Event is not a candidate key because it is not minimal. However, Name/Event is a candidate key because no two rows have the same Name/Event values and you can easily see neither Name nor Event is a superkey, so the pair is minimal.

You could still have a problem if one of George's other brothers, who are all named George, moves in. If they compete in the same event, you won't be able to tell them apart. Perhaps we should add a CompetitorId column to the table after all.

Note that there may be more than one superkey or candidate key in a table. In Figure 3-1, Event/NextOfKin also forms a candidate key because no two rows have the same Event and NextOfKin values. (That would probably not be the most natural way to look up rows, however. ''Yes sir, I can look up your record if you give me your event and next of kin.'')

A *unique key* is a superkey that is used to uniquely identify the rows in a table. The difference between a unique key and any other candidate key is in how it is used. A candidate key *could* be used to identify rows if you wanted it to, but a unique key *is* used to constrain the data. In this example, if you make Name/Event be a unique key, the database will not allow you to add two rows with the same Name and Event values. A unique key is an implementation issue, not a more theoretical concept like a candidate key is.

A *primary key* is a superkey that is actually used to uniquely identify or find the rows in a table. A table can have only one primary key (hence the name ''primary''). Again, this is more of an implementation issue than a theoretical concern. Database products generally take special action to make finding records based on their primary keys faster than finding records based on other keys.

Some databases allow alternate key fields to have missing values, whereas all of the fields in a primary key are required. For example, the Competitors table might have Name/Address/Event as a unique key and Name/Event as a primary key. Then it could contain a record with Name and Event but no Address value. (Although that would be a bit strange. We might want to require that all of the fields have a value.)

An *alternate key* is a candidate key that is not the primary key. Some also call this a *secondary key*, although others use the term secondary key to mean any set of fields used to locate records even if the fields don't define unique values.

That's a lot of keys to try to remember! The following list briefly summarizes the different flavors:

- **Compound key or composite key:** A key that includes more than one field.
- **Superkey:** A set of columns for which no two rows can have the exact same values.
- **Candidate key:** A minimal superkey.
- **Unique key:** A superkey used to require uniqueness by the database.
- **Primary key:** A unique key that is used to quickly locate records by the database.
- **Alternate key:** A candidate key that is not the primary key.
- **Secondary key:** A key used to look up records but that may not guarantee uniqueness.

One last kind of key is the *foreign key*. A foreign key is used as a constraint rather than to find records in a table, so it is described a bit later in the section ''Constraints.''

Indexes

An index is a database structure that makes it quicker and easier to find records based on the values in one or more fields. Indexes are not the same as keys, although the two are related closely enough that many developers confuse the two and use the terms interchangeably.

For example, suppose you have a Customers table that holds customer information: name, address, phone number, Swiss bank account number, and so forth. The table also contains a CustomerId field that it uses as its primary key.

Unfortunately customers usually don't remember their customer IDs, so you need to be able to look them up by name or phone number. If you make Name and PhoneNumber be two different keys, you can quickly locate a customer's record in three ways: by customer ID, by name, and by phone number.

> *Relational databases also make it easy to look up records based on non-indexed fields, although it may take a while. If the customer only remembers his address and not his customer ID or name, you can search for the address even if it that field isn't part of an index. It may just take a long time. Of course if the customer cannot remember his name, he's got bigger problems.*

Building and maintaining an index takes the database some extra time, so you shouldn't make indexes gratuitously. Place indexes on the fields that you are most likely to need to search and don't bother indexing fields such as apartment number that you are unlikely to need to search.

Constraints

As you might guess from the name, a *constraint* places restrictions on the data allowed in a table. In formal database theory, constraints are not considered part of the database. However, in practice constraints play such a critical role in managing the data properly that they are informally considered part of the database. (Besides, the database product enforces them!)

The following sections describe some of the kinds of constraints that you can place on the fields in a table.

Basic Constraints

Relational databases let you specify some simple basic constraints on a particular field. For example, you can make a field required. The special value *null* represents an empty value. For example, suppose you don't know a customer's income. You can place the value null in the Income field to indicate that you don't know the correct value. This is different from placing 0 in the field, which would indicate that the customer doesn't have any income.

Making a field required means it cannot hold a null value, so this is also called a *not null* constraint.

The database will also prevent a field from holding a value that does not match its data type. For example, you cannot put a 20-character string in a 10-character field. Similarly, you cannot store the value ''twelve'' in a field that holds integers.

These types of constraints restrict the values that you can enter into a field. They help define the field's domain so they are called *domain constraints*. Some database products allow you to define more complex domain constraints, often by using check constraints.

Check Constraints

A *check constraint* is a more complicated type of restriction that evaluates a Boolean expression to see if certain data should be allowed. If the expression evaluates to true, the data is allowed.

A *field-level* check constraint validates a single column. For example, in a SalesPeople table you could place the constraint `Salary > 0` on the Salary field to mean that the field's value must be positive.

A *table-level* check constraint can examine more than one of a record's fields to see if the data is valid. For example, the constraint `(Salary > 0) OR (Commission > 0)` requires that each SalesPeople record have a positive salary or a positive commission (or both).

Primary Key Constraints

By definition, no two records can have identical values for the fields that define the table's primary key. That greatly constrains the data.

In more formal terms, this type of constraint is called *entity integrity*. It simply means that no two records are exact duplicates (which is true if the fields in their primary keys are not duplicates) and that all of the fields that make up the primary key have non-null values.

Unique Constraints

A *unique constraint* requires that the values in one or more fields be unique. Note that it only makes sense to place a uniqueness constraint on a superkey. Recall that a superkey is a group of one or more fields that cannot contain duplicate values. It wouldn't make sense to place a uniqueness constraint on fields that can validly contain duplicated values. For example, it would be silly to place a uniqueness constraint on a Gender field.

Foreign Key Constraints

A *foreign key* is not quite the same kind of key defined previously. Instead of defining fields that you use to locate records, a foreign key refers to a key in another (foreign) table. The database uses it to locate records in the other table but you don't. Because it defines a reference from one table to another, this kind of constraint is also called a *referential integrity constraint*.

A foreign key constraint requires that a record's values in one or more fields in one table (the referencing table) must match the values in another table (the foreign or referenced table). The fields in the referenced table must form a candidate key in that table. Usually they are that table's primary key, and most database products try to use the foreign table's primary key by default when you make a foreign key constraint.

For a simple example, suppose you want to validate the entries in the Competitors table's Event field so the minimum wage interns manning the phones cannot assign anyone to an event that doesn't exist.

To do this with a foreign key, create a new table named Events that has a single column called Event. Make this the new table's primary key and make records that list the possible events: Pyramid Boarding, Pyramid Luge, Camel Drafting, and Sphinx Jumping.

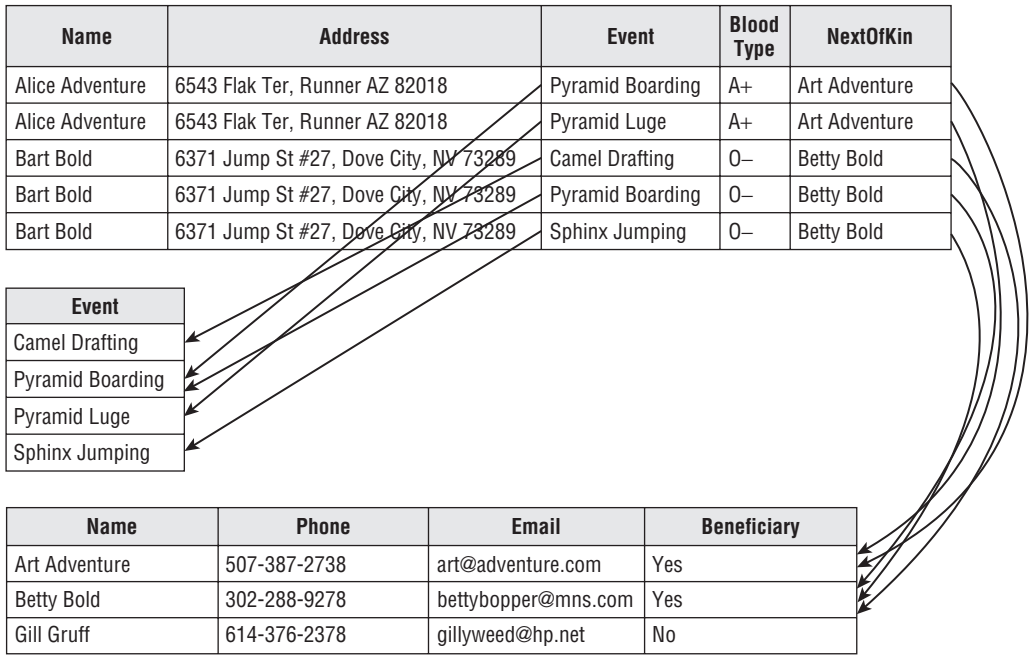
Next, make a foreign key that relates the Competitors table's Event field with the Events table's Event field. Now whenever someone adds a new record to the Competitors table, the foreign key constraint will require that the new record's Event value be listed in the Events table.

The database will also ensure that no one modifies a Competitors record to change the Event value to something that is not in the Events table.

Finally, the database will take special action if you try to delete a record in the Events table if its value is being used by a Competitors record. Depending on the type of database and how you have the relationship configured, the database will either refuse to remove the Events record or it will automatically delete all of the Competitors records that use it.

This example uses the Events table as a lookup table for the Competitors table. Another common use for foreign key constraints is to make sure related records always go together. For example, you could build a NextOfKin table that contains information about the competitors' next of kin (name, phone number, email address, beneficiary status, and so forth). Then you could make a foreign key constraint to ensure that every Competitor record's NextOfKin value is contained in the Name fields in some NextOfKin table record. That way you know that you can always contact the next of kin for anyone in the Competitors table.

Figure 3-2 shows the Competitors, Events, and NextOfKin tables with lines showing the relationships among their related fields.



Name	Address	Event	Blood Type	NextOfKin
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Boarding	A+	Art Adventure
Alice Adventure	6543 Flak Ter, Runner AZ 82018	Pyramid Luge	A+	Art Adventure
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Camel Drafting	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Pyramid Boarding	O–	Betty Bold
Bart Bold	6371 Jump St #27, Dove City, NV 73289	Sphinx Jumping	O–	Betty Bold

Event
Camel Drafting
Pyramid Boarding
Pyramid Luge
Sphinx Jumping

Name	Phone	Email	Beneficiary
Art Adventure	507-387-2738	art@adventure.com	Yes
Betty Bold	302-288-9278	bettybopper@mns.com	Yes
Gill Gruff	614-376-2378	gillyweed@hp.net	No

Figure 3-2

Foreign keys define associations between tables that are sometimes called *relations*, *relationships*, or *links* between the tables. The fact that the formal database vocabulary uses the word *relation* to mean table sometimes leads to confusion. Fortunately, the formal and informal database people usually get invited to different parties so the terms usually don't collide in the same conversation.

Database Operations

The final topic in this chapter covers database operations. (I'll save the rest so I have something for the rest of the book.)

Eight operations were originally defined for relational databases and they form the core of modern database operations. The following list describes those original operations:

- **Selection:** This selects some or all of the records in a table. For example, you might want to select only the Competitors records where Event is Pyramid Luge so you can know who to expect for that event (and how many ambulances to have standing by).
- **Projection:** This drops columns from a table (or selection). For example, when you make your list of Pyramid Luge competitors you may only want to list their names and not their addresses, blood types, events (which you know is Pyramid Luge anyway), or next of kin.

- **Union:** This combines tables with similar columns and removes duplicates. For example, suppose you have another table named FormerCompetitors that contains data for people who participated in previous years' competitions. Some of these people are competing this year and some are not. You could use the union operator to build a list of everyone in either table. (Note that the operation would remove duplicates, but for these tables you would still get the same person several times with different events.)
- **Intersection:** This finds the records that are the same in two tables. The intersection of the FormerCompetitors and Competitors tables would list those few who competed in previous years and who survived to compete again this year (the slow learners).
- **Difference:** This selects the records in one table that are not in a second table. For example, the difference between FormerCompetitors and Competitors would give you a list of those who competed in previous years but who are not competing this year (so you can email them and ask them what the problem is).
- **Cartesian Product:** This creates a new table containing every record in a first table combined with every record in a second table. For example, if one table contains values 1, 2, 3 and a second table contains values A, B, C, then their Cartesian product contains the values 1/A, 1/B, 1/C, 2/A, 2/B, 2/C, 3/A, 3/B, and 3/C.
- **Join:** This is similar to a Cartesian product except records in one table are paired only with those in the second table if they meet some condition. For example, you might join the Competitors records with the NextOfKin records where a Competitors record's NextOfKin value matches the NextOfKin record's Name value. In this example, that gives you a list of the competitors together with their corresponding next of kin data.
- **Divide:** This operation is the opposite of the Cartesian product. It uses one table to partition the records in another table. It finds all of the field values in one table that are associated with every value in another table. For example, if the first table contains the values 1/A, 1/B, 1/C, 2/A, 2/B, 2/C, 3/A, 3/B, and 3/C and a second table contains the values 1, 2, 3, then the first divided by the second gives A, B, C. (Don't worry, I think it's pretty weird and confusing, too, so it won't be on the final exam. Probably.)

The workhorse operation of the relational database is the join, often combined with selection and projection. For example, you could *join* Competitors records with NextOfKin records that have the correct name. Next you could *project* to select only the competitors' names, the next of kin names, and the next of kin phone numbers. You could then *select* only Bart Bold's records. Finally, you could *select* for unique records so the result would contain only a single record containing the values Bart Bold, Betty Bold, 302-288-9278.

The following SQL query produces this result:

```
SELECT DISTINCT Competitors.Name, NextOfKin.Name, Phone
FROM Competitors, NextOfKin
WHERE Competitors.NextOfKin = NextOfKin.Name
  AND Competitors.Name = 'Bart Bold'
```

The `SELECT` clause performs selection, the `FROM` clause tells which tables to join, the first part of the `WHERE` clause (`Competitors.NextOfKin = NextOfKin.Name`) gives the join condition, the second part of the `WHERE` clause (`Competitors.Name = 'Bart Bold'`) selects only Bart's records, and the `DISTINCT` keyword selects unique results.

The results of these operations are table-like objects that aren't permanently stored in the database. They have rows and columns so they look like tables, but their values are generated on the fly when the database operations are executed. These result objects are called *views*. Because they are often generated by SQL queries, they are also called *query results*. Because they look like tables that are generated as needed, they are sometimes called *virtual tables*.

Chapter 20 has more to say about relational database operations as they are implemented in practice.

Summary

Before you can start designing and building relational databases, you need to understand some of the basics. This chapter provided an introduction to relational databases and their terminology.

In this chapter you learned about:

- Formal relational database terms such as relation, attribute, and tuple.
- Informal terms such as table, row, record, column, and field.
- Several kinds of keys including superkeys, candidate keys, and primary keys.
- Different kinds of constraints that you can place on columns or tables.
- Operations defined for relational databases.

The following chapters change the book's focus from general database concepts and terminology to design techniques. They describe the tasks you must perform to design a database from scratch. Chapter 4 starts the process by explaining how to gather user requirements so the database you design has a good chance of actually satisfying the users' needs.

Before you move on to Chapter 4, 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. What does the following check constraint on the SalesPeople table mean?

```
((Salary > 0) AND (Commission = 0)) OR ((Salary = 0) AND (Commission > 0))
```

2. In Figure 3-3, draw lines connecting the corresponding terms.

Attribute	Row	File
Relation	Column	Relationship
Foreign Key	Table	Virtual Table
Tuple	Foreign Key	Record
View	Query Result	Field

Figure 3-3

For questions 3 through 6, suppose you're a numismatist and you want to track your progress in collecting the 50 state quarters created by the United States Mint. You start with the following table and plan to add more data later (after you take out the trash and finish painting your lead miniatures).

State	Abbr	Title	Engraver	Year	Got
Arizona	AZ	Grand Canyon State	Joseph Menna	2008	No
Idaho	ID	Esto Perpetua	Norm Nemeth	2007	No
Iowa	IA	Foundation in Education	John Mercanti	2004	Yes
Michigan	MI	Great Lakes State	Donna Weaver	2004	Yes
Montana	MT	Big Sky Country	Don Everhart	2007	No
Nebraska	NE	Chimney Rock	Charles Vickers	2006	Yes
Oklahoma	OK	Scissortail Flycatcher	Phebe Hemphill	2008	No
Oregon	OR	Crater Lake	Charles Vickers	2005	Yes

3. Is State/Abbr/Title a superkey? Why or why not?
4. Is Engraver/Year/Got a superkey? Why or why not?
5. What are all of the candidate keys for this table?
6. What are the domains of each of the table's columns?

For questions 7 through 10, suppose you are building a dorm room database. Consider the following table. For obscure historical reasons, all of the rooms in the building have even numbers. The Phone field refers to the number of the phone in the room. Rooms that have no phone cost less but students in those rooms are required to have a cell phone (so you can call them and nag if they miss too many classes).

Room	FirstName	LastName	Phone	CellPhone
100	John	Smith	Null	202-837-2897
100	Mark	Garcia	Null	504-298-0281
102	Anne	Johansson	202-237-2102	Null
102	Sally	Helper	202-237-2102	Null
104	John	Smith	202-237-1278	720-387-3928
106	Anne	Uumellmahaye	Null	504-298-0281
106	Wendy	Garcia	Null	202-839-3920
202	Mike	Hfuhruhurr	202-237-7364	Null
202	Jose	Johansson	202-237-7364	202-839-3920

7. If you don't allow two people with the same name to share a room (due to administrative whimsy), what are all of the possible candidate keys for this table?
8. If you do allow two people with the same name to share a room, what are all of the possible candidate keys for this table?
9. What field-level check constraints could you put on this table's fields? Don't worry about the syntax for performing the checks, just define them.
10. What table-level check constraints could you put on this table's fields? Don't worry about the syntax for performing the checks, just define them.