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.