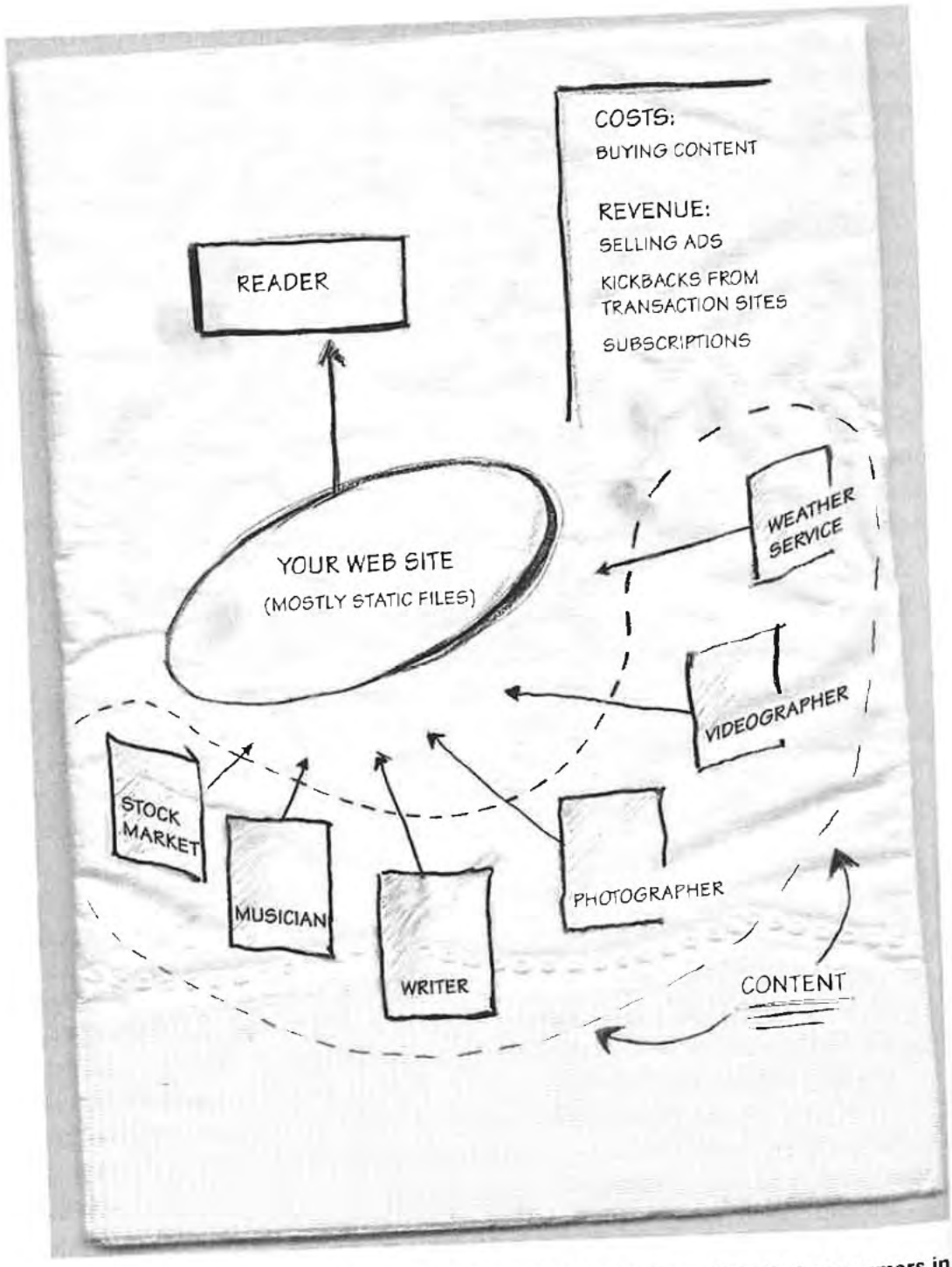


Figure 2.1: A Category 1 site simply provides bodies of information that consumers in the 1980s obtained offline

and write some programs. But once established, the site can grow in popularity with very little additional investment on your part. See Figure 2.2 for an example of a Category 2 site.

3 *Sites that provide a service via a server-side program.* An example of this would be providing a wedding planning program. The user tells you how much he or she wants to spend, when and where the wedding is, who is invited, and so on. Your program then figures a detailed budget, develops an invitation list, and maintains gift and thank-you lists. You are then in a position to sell an ad to the Four Seasons hotel that will be delivered to couples getting married on June 25, who live less than 100 miles away, with fewer than 80 guests, who have budgeted more than $17,000. Or, suppose that you can convince people to use your server to fill out their income tax return. Why should they go to the trouble of installing a program on their computer that they are only going to use once. Your Web site is so much more convenient! Meanwhile, as they are filling out their taxes, you are building up a database of information that they would ordinarily be reluctant to give you, such as how much money they made last year. Armed with this information, you can sell very expensive and highly targeted advertising. Did the user make more than $350,000 last year? Rolls-Royce might pay you $10 to show the user an ad for the Corniche IV. Did the user cheat on his return? VARIG would probably pay you $25 to run a page with Brazil's extradition laws and fares to Rio. An example of a Category 3 site is shown in Figure 2.3.

4 *Sites that define a standard that enables a consumer to seamlessly query multiple databases.* For example, car dealers have computers managing their inventory, but that data is imprisoned on the dealers' computers and is unavailable to consumers in a convenient manner. Suppose you define a standard that allows the inventory computers inside car dealerships to download their current selection of cars, colors, and prices. You get the car dealers to agree to provide their information to you. Then your site becomes a place where a consumer can say "I want a new dark green Dodge Grand Caravan with air conditioning and antilock brakes that's for sale within 60 miles of zip code 02176." From your query to the dealers' multiple databases, your user can get a list of all the cars available that match their criteria, and can jump right to the relevant dealer's Web site. For another example, note that just about every bank has a computer database of the interest rate they will pay on a certificate of deposit. But this data is not in any standard form. You engineer a standard that lets bank computers tell your Web server what CDs are available and at what rate. Then, when a

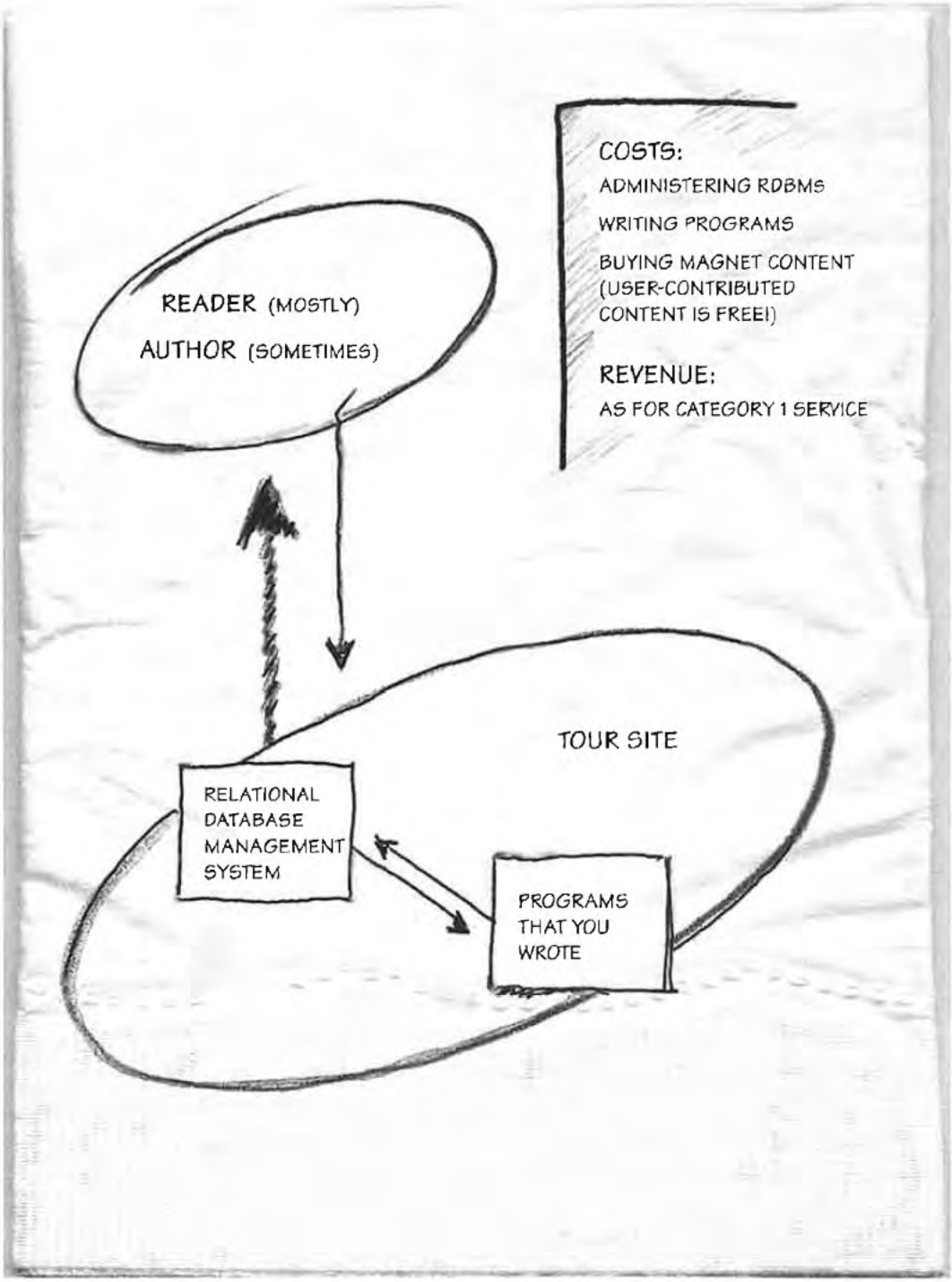


Figure 2.2: A Category 2 site provides collaboratively created information. Even a humble classified ad system falls into this category

Oracle Exhibit 1011
Page 37 of 171

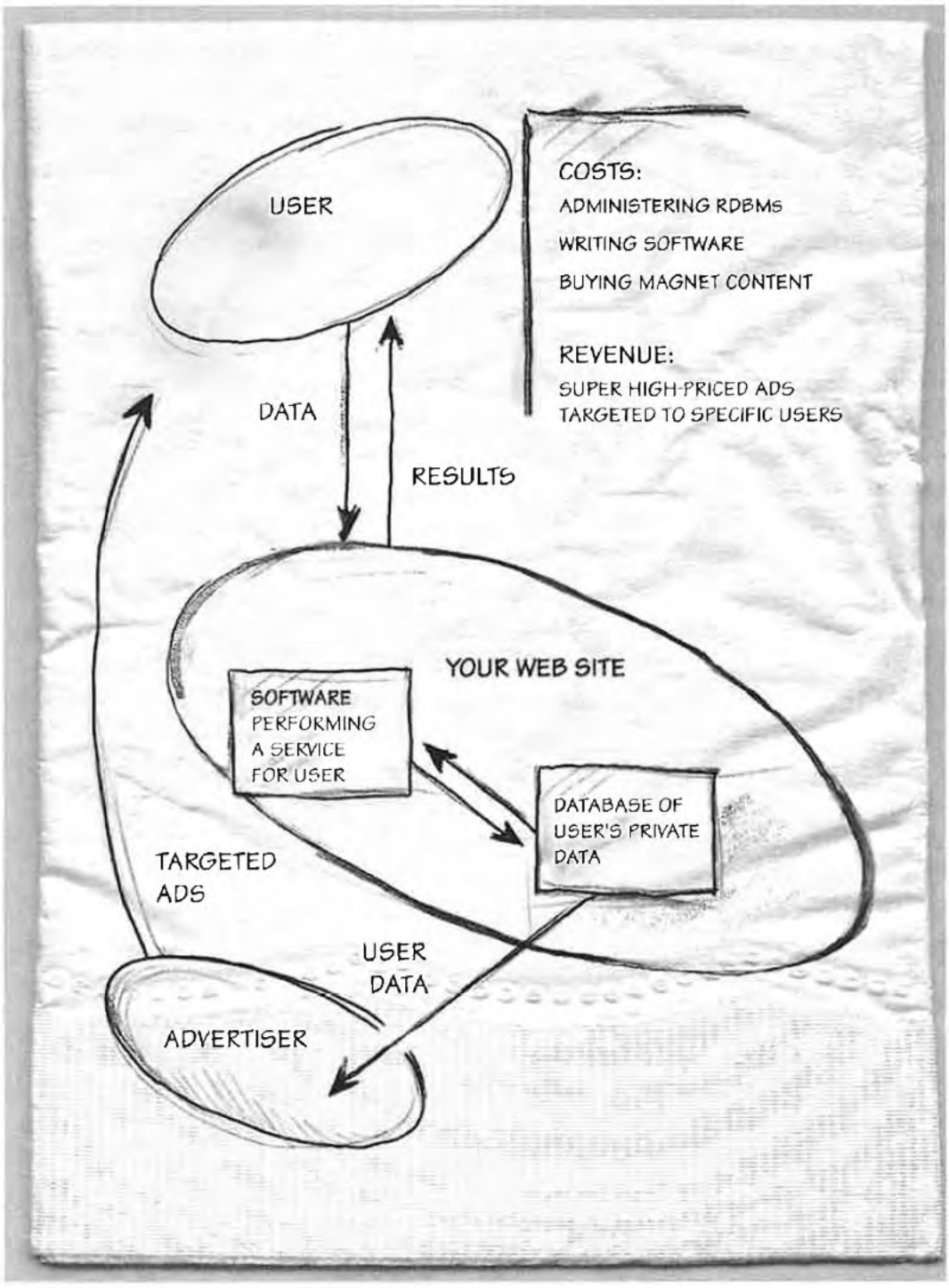


Figure 2.3: Category 3 sites provide a service to the user via a server-side program.

Oracle Exhibit 1011
Page 38 of 171

consumer visits your site and indicates an interest in six-month CDs, your server can display a list of the banks, ordered by interest rate offered. If the consumer clicks through to a bank and buys a CD, you collect a commission. See Figure 2.4 for an example of how a Category 4 site might be set up.

In terms of bang per dollar invested, the most expensive type of site is Category 1. To create a site like this, you have to hire writers, pay photographers and editors, and scrupulously maintain your site. If you stop updating for one day, people may turn away to a competitor's site.

A Category 2 site is very cheap to start up. You spend a little bit of money for programming, a data model, and "anchor content." Thereafter the site expands itself. Category 3 sites require much more investment and effort, particularly in selling ads and working with advertisers, but they have potentially much higher payoffs since you know so much more about your users. An advertiser won't pay too much for an ad on the Netscape home page, perhaps a few cents per impression. All that anyone knows about the readers is that they haven't changed their browser's default first page. An advertiser will pay a lot more for an ad on a search engine's site, if it is delivered only to those people who've entered query words relevant to the product advertised. For example, Century 21 would pay a lot to have their ad delivered to people who included "real estate" in their query string. If you learn enough about your reader, you might be able to charge an advertiser several dollars just to display one banner ad to that reader.

Category 4 is probably the most lucrative because it harnesses the full power of the Internet. Entering this area requires having the right connections and making the proper business arrangements. It is the Category 4 sites that will change the way the computer is seen. In a Category 3 car site, you might offer users a new car planning service. You know what kinds of car they need, their budgets, and where they live. You can charge car manufacturers and local dealerships quite a bit to serve banner ads that might tempt particular users. But in a Category 4 site, you know which user is about to drive to which dealer to buy which car with which options. You can demand a commission from the selling dealer. You can charge $50 to a competing dealership to run an ad that says "Just wait two days and Joe Foobar Toyota will have that same color sedan in stock. The price will be $75 lower and Joe Foobar Toyota is ten miles closer to your house than the dealer you're planning to buy from."

Let's consider how we'd use these categories to build services for a variety of fields.

Oracle Exhibit 1011
Page 39 of 171

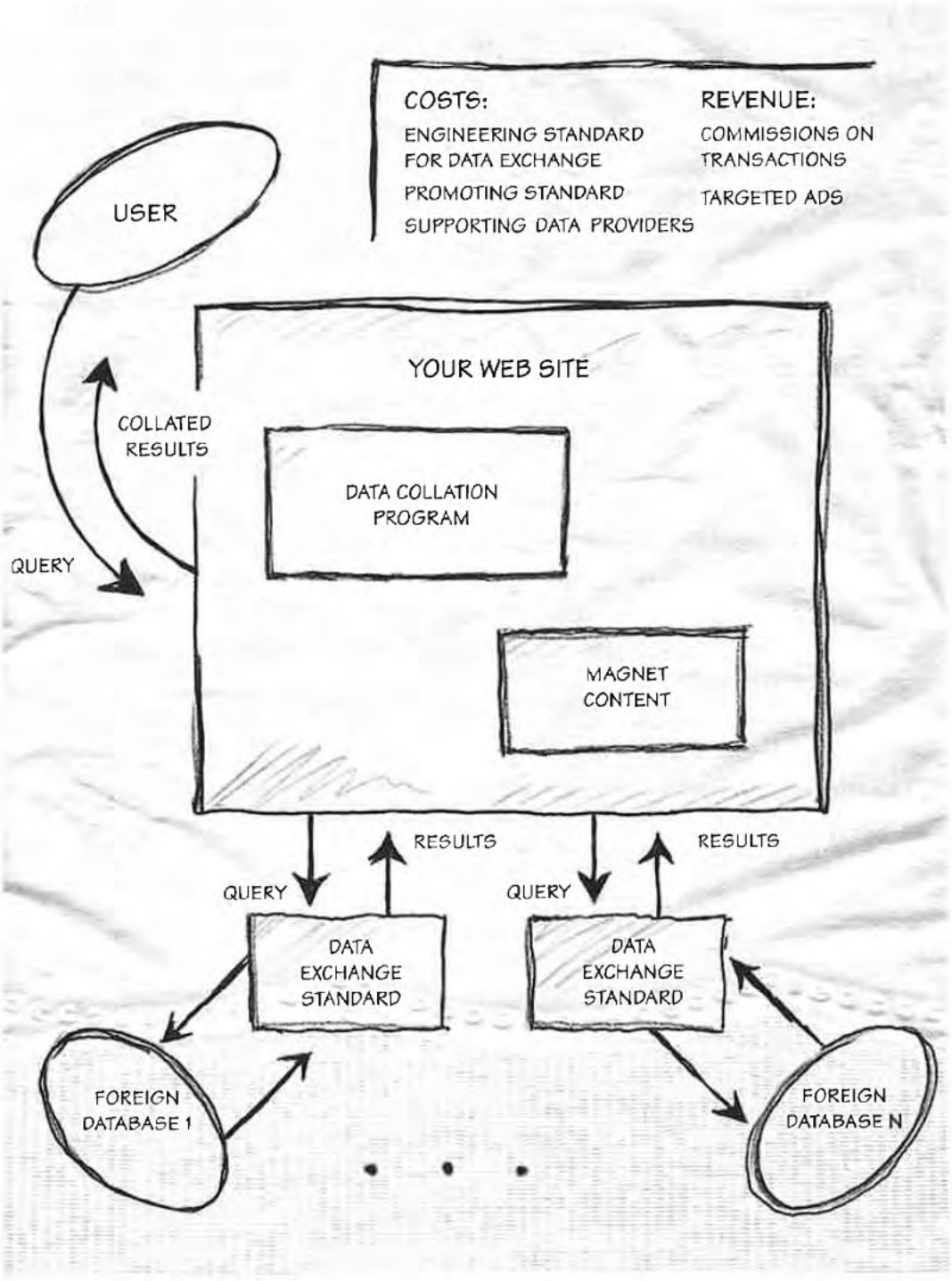


Figure 2.4: Category 4 sites define a standard for cooperation among databases and then let users seamlessly query them all.

Oracle Exhibit 1011
Page 40 of 171

TRAVEL EXAMPLE

A lot of money dangles from the travel tree. Vendors and consumers are separated by vast distances and numerous intermediaries. Each intermediary extracts a commission. It is the sum total of those commissions that is potentially available to Web travel sites.

Traditional publishers do their best to capture these commissions with guidebooks, magazines, advertising supplements, and plain old brochures. Despite the vast forests that are chopped down in this valiant attempt, dead-trees publishers aren't satisfying travelers. A subscriber opens a travel magazine and finds all of the articles either much too long, because he or she isn't planning to visit the city described, or much too short, because he or she has a definite trip planned and is hungry for detail. Travel books are better, though they generally suffer from the "one voice" problem and the best written books usually don't have any pictures because of the expense of color printing can't be justified when there aren't any ads.

Traditional travel publishers don't even try to give consumers the most critical information, such as how well the last 100 people who went there liked it. An Internet travel site starts off with a huge relevance advantage. Search engines consistently deliver travel URLs to people who are about to book a trip or leave for a destination. No other advertising-carrying medium comes close, except perhaps the airplane and hotel magazines that consumers get after they've started their trip.

The most important Category 1 service for a travel site is magnet content that will attract people to Category 2, 3, and 4 services. You can organize all kinds of Category 2 services around magnet content. For example, suppose that your magnet content in an Italian site says, "I went into the cathedral at Assisi to reload my camera; there were some pictures on the walls. I think somebody next to me said they were painted by a guy named 'Joe.'" Somewhere out there on the Net there is a guy who did his art history Ph.D. on those frescoes by Giotto. Your magnet content's function is to draw that expert into surfing your site and contributing a few paragraphs via your comment server.

You will also want to have classified ads for vacation homes and tour packages, Q&A forums, and a quality rating system for tour operators and hotels. What makes this all more useful than rec.travel.europe is that the magnet content gives structure to the user-contributed thoughts.

Note: Check out http://webtravel.org/webtravel/ for a glimpse at my idea of what a Category 1 and 2 travel Web site should be like.

The obvious Category 3 service for a travel site is a trip planning system. Your database will know where the consumer intends to be at every hour of every day of the trip. With that kind of information about your readers, if you can't sell high-priced ads for delivery to specific people, it won't be my fault!

The obvious Category 4 application is a hotel room booking system. Your server queries all the hotel reservation computers and the consumer can query you: "Show me the available hotel rooms in Paris for September 15–20, sorted by distance from the Louvre and then by price. Oh, and convert the prices to American dollars, please."

REAL ESTATE EXAMPLE

Real estate agents in the United States collect a 6 percent commission every time they connect buyer and seller. This leads to musings like "Why does New York have so much garbage and Los Angeles so many real estate agents?" (Standard answer: "Because New York had first choice.") This kind of humor would hurt real estate agents' feelings if they weren't raking in $142 billion each year (according to the 1992 Census).

Realtors are precisely the sorts of intermediaries that Internet technology is supposed to eliminate. If you were trying to capture a share of this $142 billion, you'd want to start in Category 1 with articles about moving or character sketches for neighborhoods. This content would attract search engine users. Moving into Category 2, you'd run a commission-based classified ad system to connect renters with landlords and buyers with sellers. For a Category 3 service, perhaps you could create a moving planner. You'd get the old address, the new address, and the moving date from the user. You could use that data to help the user disconnect and hook up utilities, compute cost-of-living differences, and arrange travel. Meanwhile, you are bombarding the person with ads from competing communications, furniture, and appliance vendors in the new locale. For a Category 4 application you'd cut deals with other real estate sites to query their databases. If a user makes a transaction on that foreign site, you split the commissions.

MEDICAL EXAMPLE

Joe Schmoe is treated at five different hospitals over a three-year period. Each hospital has a sophisticated computer medical record system. When Mr. Schmoe visits Hospital Number 6, they create a record for him on *their* big database. But even though Joe's entire medical record is available in electronic form, the new hospital has no way to import it from the previous five hospitals. So they don't realize that Joe is allergic to an obscure drug. Joe dies.

You could have saved Joe's life. You could have built a Web site where Joe can store and control his own medical record. When he goes to a new hospital,

he can authorize them to retrieve portions of his record and to store their findings back on his personal server. Then when he goes to a new hospital, they can download his record and find out just which drugs will provoke a life-threatening allergic reaction.

This is a Category 4 site with a vengeance. All of the challenge is in figuring out to make disparate databases talk to each other. The data models on the hospital systems can be radically different. Even trivial differences, such as different abbreviations for the same disease, can prevent databases from understanding each other. You have to find a way to get these databases to cooperate even though they were never designed to do that.

Big companies do this all the time, of course. They'll decide that, starting July 1, everyone is moving to the big new central database system and everyone will use the same part number for a #8 machine screw. They have the money, resources, and authority to make this happen. On the Internet, there are many more organizations who could benefit from data exchange, but there is much less trust, less money, and no central authority.

There is a huge technical challenge in figuring out a cheap automated or semi-automated method of integrating databases. In fact, this is my primary research focus at MIT. Though my partners and I have done some interesting things in the medical domain, I won't say that we've got any kind of general solution. As soon as we do, we'll have a killer Web site.

That's a whirlwind tour of the four ways to break down consumer Web services. Assuming you run with one of my ideas or come up with a great one of your own, you'll be on your way to achieving the Holy Grail of the Web: traffic.

WE LOSE MONEY ON EVERY HIT BUT MAKE IT UP ON VOLUME

If nobody ever visits your site, then you won't make any money. Advertisers don't like to place ads where nobody will see them. You can't sell stuff to nobody. Collecting a middleman's commission on no transactions is not very attractive. If you have visitors then you at least have a chance to make money, even if you didn't have a specific plan when you set up the site.

Internet commerce is the most obvious method of making money from a popular site, but it may not be the best. Internet commerce appeals to a particularly male fantasy. Guys like the idea that after a short initial period of programming, a computer will tirelessly slave away for them, making them money 24 hours a day. Set up the site, walk away, and watch the money pile up in your bank account.

You can feed this fantasy by reading articles in the business press about http://amazon.com, the perennial poster children for Internet commerce.

They set up what is essentially a front-end to a wholesale book distributor's database, and now they are selling books every few seconds. It sounds like they are rolling in money.

Well, it turns out that I know some people who work at amazon.com. The customers don't always fill out the forms exactly right. The books aren't always in stock like they should be. The customers send e-mail asking when their books are going to be shipped. So instead of one Unix box and a big vault for the cash, the company has 200 employees sucking all the money out of it. And remember, this is the best that anyone has really done: high expenses and high sales. More typical is an Internet store with high expenses and low or no sales.

If amazon.com offers a cautionary lesson to those who would make their fortune processing transactions, it offers an encouraging lesson to those who would publish deep content. They have an "associated bookstore" program. You become an associated bookstore by adding an encoded link from your pages to amazon. This doesn't mean you have to uglify your site with sales promotions. In my case, I just went through http://photo.net/photo and replaced

```
You should check the Kodak Professional Photoguide
```

with

```
You should check the
<a href="http://www.amazon.com/exec/obidos/ISBN=0879857595/photonetA/">
Kodak Professional Photoguide
</a>
```

When users click on the link, they are presented with an order form from amazon.com. Should the order be completed, amazon will kick back to me 8 percent of the price of the book. I got some e-mail from a travel agent offering a similar arrangement. If I linked to them from my travel pages, they'd kick back to me half of their travel agent's commission (about 12 percent) on any sales that resulted from the link.

It occurred to me that, in a perfectly competitive environment, referral arrangements like this would eventually result in almost all of the profit being paid to the owner of the content. Transaction processors will earn a normal return on their investment, but nothing extra. Why? I'm currently linking to amazon.com for an 8 percent blandishment. If Barnes & Noble offers me 12 percent, I can write a perl script to grind over my 1,000 .html pages and replace references to amazon.com with references to Barnes & Noble. If I were linked to a travel agent offering me 50 percent of their commission, a travel agent late to the Internet booking market might offer me 100 percent of their commission in hopes of capturing my users as repeat customers. Another five minutes with perl and all of my static .html pages are pointing to a new travel agent.

Word to the wise: Be sure to carefully check out the conditions under which you get paid. All of the revenue from my site is donated to Angell Memorial Animal Hospital here in Boston. With 5,000 readers on my Web site every day, I figured all that money flowing from amazon.com would buy the dogs filet mignon in gold-plated bowls. But it turns out that special orders are excluded from my take. And a lot of the books that I recommend are special order only. And if the user buys ten books in the session, I only get paid for the one title that I recommended. This was beginning to seem unfair. After all, suppose I refer Joe User to amazon for a special order book that I think he should read. While he is there, he swells amazon's coffers by picking up ten more books. I get nothing. Nada. Zip.

The worst was yet to come. Amazon only pays a referral fee if the user you refer buys the book that you recommend and does so from the very first amazon page he or she is shown. Suppose I refer Jane Clever to amazon.com to buy *The Practical SQL Handbook*. She looks at the amazon page and is happy with the book, price, and so on. But she decides to browse around a bit before ordering. She ultimately ends up buying *The Practical SQL Handbook* and $500 of other database books. I get nothing. Nada. Zip.

The amazon folks say that they don't expect too many people to go browsing before purchasing. This, of course, is common sense: When you go to a store knowing that you need something, you always walk right up the register and buy it before turning your head to the left or right and looking to see if there is anything else you need. Yeah. And just in case the user isn't predisposed to browse, amazon provides about 25 internal links on the page that they serve for their associated bookstores (see Figures 2.5 and 2.6). If the user clicks the order button then the associated bookstore will get a referral fee. If the user clicks any of the other 25 links, such as "books on SQL," "books on relational databases," "browse," "books by the same author," or "search," then amazon has gotten itself a new customer without having to pay anything.

How does this work in practice? Listing 2.1 shows a weekly report from amazon.com to me.

I sent people to the amazon site 2,651 times. Only four of those people ignored the 25 extra links and bought books off the very first page. One of them bought a special order book for which the dogs at Angell got nothing. Bottom line: The standard Internet price for a click-through is 10 cents; it would have cost amazon.com $265 per week to get these users by purchasing ads on other folks' sites; amazon got them from me for $3.95.

Oracle Exhibit 1011
Page 45 of 171

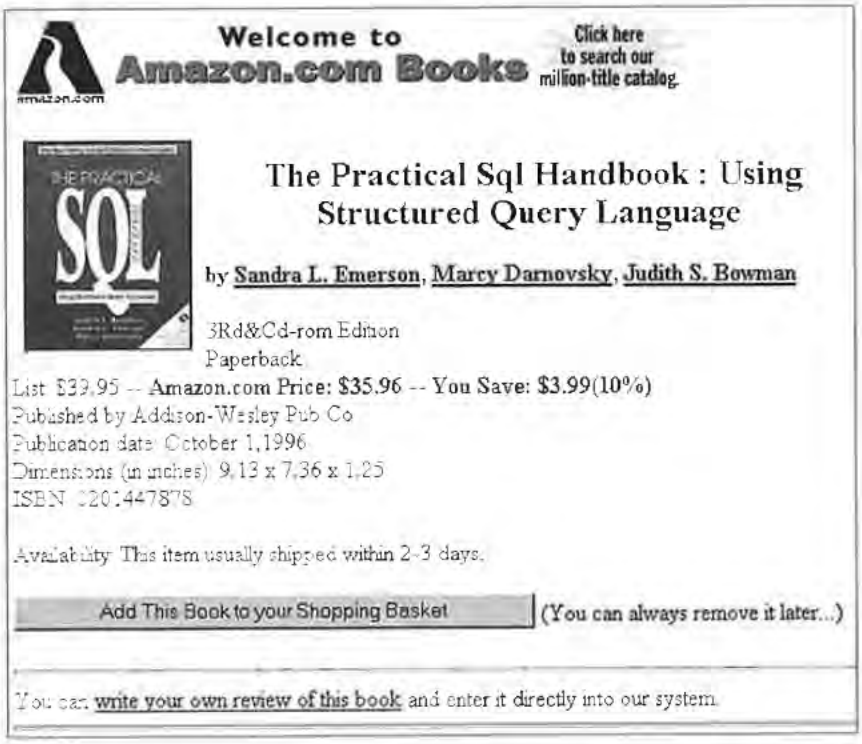


Figure 2.5: The first page a user sees when referred to amazon.com from an amazon.com "associated bookstore." The referring site will get an 8 percent commission, but only if the user orders immediately without being tempted by the 25 other links on the page.

A Final Plea

I suppose that it is possible to make money with a Web site. But before you build your site, take a moment to think about the things that you went into because you thought you could make money. How many of them proved satisfying in the long run? How many actually made money?

You could start your Web site by asking, "What can I get from this right now?" Alternatively, you could start out by asking, "What can I give people?" I expect that in the long run, you'd be about equally likely to make money with either approach. I started with the latter.

I gave away my pictures. I gave away my stories. I gave away to "competitors" my advice and software. The Web gave me back a large and growing audience

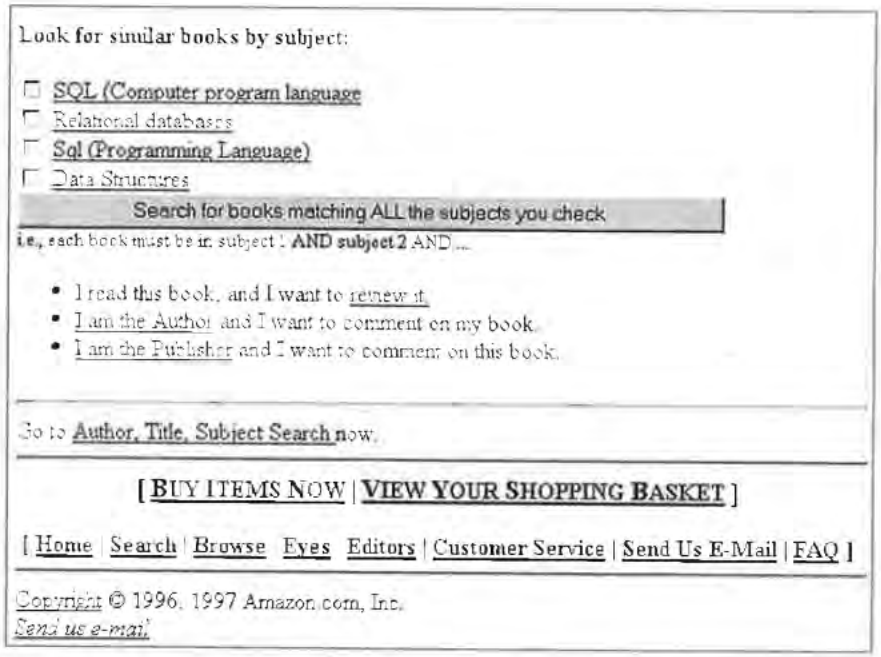


Figure 2.6: The bottom of the same amazon.com page. Note the profusion of links; clicking on any one of which will result in the referring site getting no commission.

for my work. The Web gave me back some money, to be sure. But I actually do place a higher value on some of the e-mail that I've gotten:

> "Thanks for getting me thru some sloooow weekends at work. . . . I have been working all the holiday weekends here at Directory Assistance & I wouldn't have made it out alive without your book. Thanks for sharing."
>
> —*Mary in Wisconsin*

> "I love your book for two reasons. First, it's great, the story is fresh and honest. The second is of course that it is the kind of thing that this technology has been building for—it somehow makes all those millions of dollars spent on computers, on the Net, on decades of development seem like there might have been a reason for it all. Thanks for the warm glow."
>
> —*Jonathan in New Zealand*

Listing 2.1: Weekly report on my commission from amazon.com

```
Click-throughs and sales by individual book
For the week of 15-Dec-96 through 21-Dec-96
Store ID photonet

                                YOUR
   ISBN       HITS ORDERED      FEE  TITLE
------------ ---- ------- ------ ----------------------------------------
0028604881      5       0   0.00 Baedeker Prague (Baedeker Guides)
0028609034      3       0   0.00 Frommer's Prague and the Best of the Cz
006091985X      1       0   0.00 Modern Baptists
0062771590      1       0   0.00 Access Cape Cod Martha's Vineyard & Nan
0070074178      1       0   0.00 **2** Principles of Corporate Finance (
014012991X      4       0   0.00 Tales of a Shaman's Apprentice : An Eth
0201447878      3       0   0.00 The Practical Sql Handbook : Using Stru
0201622165     10       0   0.00 MacIntosh Human Interface Guidelines
0240800265      9       1   0.00 **2** Basic Photographic Materials and
                1            sold at  0% off list price of   44.95
024080158X     11       0   0.00 **2** View Camera Technique
0240802519     28       0   0.00 **2** The Hasselblad Manual : A Compreh
0393315290      4       0   0.00 A Random Walk Down Wall Street : Includ
0679025723     15       0   0.00 Fodor's : The Czech Republic & Slovakia
0679733485      1       0   0.00 The Marriage of Cadmus and Harmony
0679765425      2       0   0.00 **1** The New Yorker Book of Dog Cartoo
0684830426      1       0   0.00 **1** The Great Gatsby (Scribner Classi
0810960885      1       0   0.00 *** This item no longer in our catalog
0811807622    726       0   0.00 The Body : Photographs of the Human For
0821223321      2       0   0.00 America in Passing
0823049752     56       0   0.00 Photography
082306459X   1152       0   0.00 Graphis Nudes
0855339438    135       0   0.00 **2** The Workbook of Nudes and Glamour
0864422458      2       0   0.00 Lonely Planet Czech and Slovak Republic
0871062380      4       0   0.00 Berlin (Cadogan City Guides)
0879857595      7       0   0.00 Kodak Professional Photoguide
0893815322    329       0   0.00 Edward Weston : Nudes
089526529X      2       0   0.00 Hill Rat : Blowing the Lid Off Congress
0899330169      5       0   0.00 Vermont Atlas & Gazetteer
0899332501      2       0   0.00 Maine Atlas and Gazetteer
0899332544      5       0   0.00 New Hampshire Atlas and Gazetteer
0911515003      2       0   0.00 **2** The Permanence and Care of Color
0918373883      2       0   0.00 **2** New Zealand Handbook
0936262389     30       0   0.00 Infrared Photography Handbook
1558603239      2       0   0.00 **2** Joe Celko's Sql for Smarties : Ad
1560986220      5       0   0.00 Pinhole Photographs : Photographs (Phot
1564400069      1       0   0.00 **2** New York (Cadogan City Guides)
```

Oracle Exhibit 1011
Page 48 of 171

Listing 2.1: Weekly report on my commission from amazon.com (Continued)

```
1564581845      1     0     0.00 New York (Eyewitness Travel Guides)
1564585034      9     0     0.00 Prague (Eyewitness Travel Guides)
1566910153     22     1     1.08 New Mexico Handbook
                      1          sold at 10% off list price of   14.95
1566910358      8     2     2.87 Costa Rica Handbook (2nd Ed)
                      2          sold at 10% off list price of   19.95
1568842031      2     0     0.00 The Unix-Haters Handbook/Book and Barf
1858281210     10     0     0.00 The Czech & Slovak Republics : The Roug
1860110150     11     0     0.00 Prague : Cadogan City Guides (3rd Ed)
1880559034      1     0     0.00 Mastering Black-And-White Photography :
188332338X      1     0     0.00 Czech and Slovak Republics Guide
1885492022      1     0     0.00 The New York Dog Owner's Guide : Everyt
1885559003      2     0     0.00 Passage to Vietnam : Through the Eyes o
188555902X      2     0     0.00 Passage to Vietnam/Cd-Rom
2061569072     11     0     0.00 Michelin Green Guide : New England (7th
6301973259      1     0     0.00 **2** Marty
------------ ---- ------- ------ ---------------------------------------
     Totals: 2651     4     3.95
```

NOW THAT YOU KNOW HOW TO WRITE HTML, DON'T

> "Owing to the neglect of our defences and the mishandling of the German problem in the last five years, we seem to be very near the bleak choice between War and Shame. My feeling is that we shall choose Shame, and then have War thrown in a little later, on even more adverse terms than at present."
>
> —Winston Churchill in a letter to Lord Moyne, 1938 [Gilbert, 1991]

HTML represents the worst of two worlds. We could have taken a formatting language and added hypertext anchors so that users had beautifully designed documents on their desktops. We could have developed a powerful document structure language so that browsers could automatically do intelligent things with Web documents. What we actually *have* with HTML is a hybrid: ugly documents without formatting *or* structural information.

Eventually the Web will work like a naïve user would expect it to. You ask your computer to find you the cheapest pair of blue jeans being hawked on the World Wide Web and ten seconds later you're staring at a photo of the product and being asked to confirm the purchase. You see an announcement for a concert and click a button on your Web browser to add the date to your calendar; the information gets transferred automatically. More powerful formatting isn't far off, either. Eventually there will be browser-independent ways to render the average novel readably.

Note:
See my paper "We Have Chosen Shame and Will Get War" at http://photo.net/philg/research/shame-and-war.html for more on the subject of semantic tags.

None of this will happen without radical changes to HTML, however. We'll need *semantic tags* so that publishers can say, in a way that a computer can understand, "this page sells blue jeans," and "the price of these jeans is $25 U.S." Whether we need them or not, we are sure to get new formatting tags with every new generation of browser. (Personally I can't wait to be able to caption photographs and figures, an idea that was new in word processing programs of the 1960s.)

If the information that you are publishing is at all structured, it doesn't make sense to store it in HTML files. You are throwing all of that structure away. When HTML version 7.3 comes out, you'll have to manually edit 1,000 files to take advantage of the new features.

What you need is a database. You don't need a scary huge relational database management system like I discuss later in the book. If you aren't updating your data in real-time, an ordinary text file is fine. You just don't want it to be formatted in HTML.

For example, suppose that you are putting a company phone directory on the Web. You can define a structured format like this:

```
first name|last name|department|office number|home number|location
```

There is one line for each person in the directory. Fields are separated by vertical bars. So a file at MIT might look like this:

```
Philip|Greenspun|eecs|253-8574|864-6832|ne43-609
Rajeev|Surati|athletics|253-8581|555-1212|dupont gym
```

and so on. From this one file, a 20-line Perl or Tcl script can generate

- A public Web page showing names and office phone numbers
- A public Web page for each department showing names and office phone numbers
- A private Web page for each department showing names and home phone numbers

If you decide to start using a new HTML feature, you don't have to edit all these pages manually. You just need to change a few lines in the Perl or Tcl script and then rerun it to regenerate the HTML pages.

The high level message here is that you should think about the structure of the information you are publishing first. Then think about the best way to build an investment in that structure and preserve it. Finally, devote a bit of time to the formatting of the final HTML that you generate.

IT'S HARD TO MESS UP A SIMPLE PAGE

People with limited time, money, and experience usually build fairly usable Web sites. However, there is no publishing concept so simple that money, knowledge of HTML arcana, and graphic design can't make slow, confusing, and painful for users. After you've tarted up your site with frames, graphics, and color, check the server log to see how much traffic has fallen. Then ask yourself whether you shouldn't have thought about user interface stability.

CD-ROMs are faster, cheaper, more reliable, and a more engaging audio/visual experience than the Web. Why then do they sit on the shelf while users greedily surf the slow, unreliable, expensive Web? Stability of user interface.

There are many things wrong with HTML. It is primitive as a formatting language and it is almost worthless for defining document structure. Nonetheless, the original Web/HTML model has one big advantage: All Web pages look and work more or less the same. You see something black, you read it. You see

When Is a Site Really a Database?

The Hard Part

The Easy Part

Prototyping the Site, My Theory

Prototyping the Site, the Reality

Do I Need a College Education to Understand Your System?

Chapter

10

The last few chapters have focused on technology. Technology for its own sake does not make for a popular Web site, though. Let's step back to thinking about the publishing model again. Technology should always serve the publishing model, not vice versa. In Chapter 2, I presented four ways of standing tall on the Web:

1. *Sites that provide traditional information.* This is the type of site that requires the least imagination but also the most capital investment. Find bodies of information that consumers in the 1980s bought offline and sell them online. This includes movies/videos/television, newspapers, magazines, weather reports, and stock market information. Revenue comes from advertising, links to sites that do retail transactions and give you a kickback, and occasionally subscriptions.
2. *Sites that provide collaboratively created information.* This is information that was virtually impossible to collect before the Internet. A dead-trees example would be the *Consumer Reports* annual survey of automobile reliability. They collect information from their readers via mail-in forms, collate the results, and publish them once a year. The Internet makes this kind of activity less costly for the provider and provides much more immediate and deeper information for the user. Revenue comes from the same sources as in Category 1 but production expenses are lower.
3. *Sites that provide a service via a server-side program.* An example of this would be providing a wedding planning program. The user tells you how much he or she wants to spend, when and where the wedding is, who is

Oracle Exhibit 1011
Page 53 of 171

invited, and so on. Your program then figures a detailed budget, develops an invitation list, and maintains gift and thank-you lists. You are then in a position to sell an ad to the Four Seasons hotel that will be delivered to couples getting married on June 25, who live less than 100 miles away, with fewer than 80 guests, who have budgeted more than $17,000.

4 *Sites that define a standard that enables a consumer to seamlessly query multiple databases.* For example, car dealers have computers managing their inventory, but that data is imprisoned on the dealers' computers and is unavailable to consumers in a convenient manner. Suppose you define a standard that allows the inventory computers inside car dealerships to download their current selection of cars, colors, and prices. You get the car dealers to agree to provide their information to you. Then your site becomes a place where a consumer can say, "I want a new dark green Dodge Grand Caravan with air conditioning and antilock brakes that's for sale within 60 miles of zip code 02176." From your query to the dealers' multiple databases, your user can get a list of all the cars available that match their criteria, and can jump right to the relevant dealer's Web site.

If you are trying to get rich and famous under Category 1, providing traditional information on the Web, then your site isn't really a database except in a degenerate way. On the other hand, unless you have Time-Warner's budget to keep it up to date, you probably won't be standing all that tall.

Categories 2 through 4 are really databases disguised as Web servers. Information is collected in small structured chunks from a variety of simultaneous sources. It is distributed in response to queries that cannot be anticipated when the data are stored. You might not be an Oracle Achiever. You might not know how to spell "SQL." But if your site falls into Categories 2 through 4 then you are running a database.

As long as you are running a database, you might as well do it right. Building a database-backed Web site requires a hard part that is heavy on the thinking and an easy part that is heavy on the implementation. Before you start building, though, you need a publishing idea. Let's suppose that you want to build a mailing list system for your site. You want users to be able to add and remove themselves. You need their e-mail addresses and you want their real names. You want to be the only one who can view the list and the only one who can send mail to the list.

Oracle Exhibit 1011
Page 54 of 171

THE HARD PART

We have to take our publishing idea (the mailing list) and apply three standard steps:

1. Develop a data model. What information are you going to store and how will you represent it?
2. Develop a collection of legal transactions on that model, such as inserts and updates.
3. Come up with a set of Web forms that let the user build up to one of those transactions.

STEP 1: THE DATA MODEL

You've decided to store e-mail address and real name. If you are a Defense Department-funded artificial intelligence researcher then you might spend a few years studying this "knowledge representation" problem. Otherwise, you'd probably budget two variable-length ASCII strings, one for the e-mail address and one for the name.

This is the step at which you decide whether or not you are going to allow two entries with the same e-mail address, whether you're going to assign some kind of guaranteed-unique key to each entry, and whether to also store the time and date of the entry. When I built a system like this for myself, I decided that there wasn't any point in having two entries with the same e-mail address. So the e-mail address could serve as the key. I also chose not to record the time and date of the entry, but partly because I knew that my database management software was keeping this information anyway.

STEP 2: LEGAL TRANSACTIONS

Users have to be able to add themselves to the list. So one legal transaction is the insertion of a name/e-mail pair into the database. Users should also be able to remove themselves. In the system I built, the e-mail address is the database key so the removal transaction is "delete the entry for joe@bigcompany.com." To simplify the user interface, I decided against having any update transactions. If a user changed his name or e-mail address, he could just remove the old entry and make a new one.

STEP 3: MAPPING TRANSACTIONS ONTO WEB FORMS

The only way that users can interact with your database is via the Web. So you have to come up with ways for them to formulate legal transactions using HTML forms. If the transaction is complicated and early choices change the

Oracle Exhibit 1011
Page 55 of 171

Relational Databases

What's Wrong with a File System (and Also What's Right)

Enter the Relational Database

How Does This RDBMS Thing Work?

SQL the Hard Way

Brave New World

Braver New World

Choosing an RDBMS Vendor

Paying an RDBMS Vendor

Performance

Don't Forget to Back Up

Reliability

Oracle Exhibit 1011
Page 56 of 171

Chapter

11

When you build an information system, of which a Web site is one example, you have to decide how much responsibility for data management your new custom software will take and how much you will leave to packaged software and the operating system. This chapter explains what kind of packaged data management software is available, covering files, flat file database management systems, relational database management systems (RDBMSs), object-relational database management systems, and object databases. Because RDBMS is the most popular technology, I cover it in the most depth and include a brief tutorial on SQL.

What's Wrong with a File System (and Also What's Right)

The file system that comes with your computer is a very primitive kind of database management system. Whether your computer came with the Unix file system, NTFS, or the Macintosh file system, the basic idea is the same. Data is kept in big, unstructured, named clumps. For example, suppose that you are storing a mailing list in a file system file. You could decide that no e-mail address or person's name can contain a newline character. Now you can store one entry per line. Then you could decide that no e-mail address or name may contain a vertical bar. That lets you separate e-mail address and name fields with the vertical bar character.

So far, everything is great. As long as you are careful to never try to store a newline or vertical bar, you can keep your data in this "flat file." Searching can

Oracle Exhibit 1011
Page 57 of 171

be a slow and expensive, though. What if you want to see if philg@mit.edu is on the mailing list? Then you have to read through the entire file to check.

Let's say that you write a program to process "insert new person" requests. It works by appending a line to the flat file with the new information. Suppose, however, that several users are simultaneously using your Web site. Two of them ask to be added to the mailing list at exactly the same time. Depending on how you wrote your program, the particular kind of file system that you have, and luck, you could get any of the following behaviors:

1. Both inserts succeed.
2. One of the inserts is lost.
3. Information from the two inserts is mixed together so that both are corrupted.

In the last case, the programs you've written to use the data in the flat file may no longer work.

So what? Emacs may be ancient but it is still the best text editor in the world. You love using it so you might as well spend your weekends and evenings manually fixing up your flat-file databases with Emacs. Who needs concurrency control?

It all depends on what kind of stove you have.

Yes, that's right, your stove. Suppose that you buy a $268,500 condo in Harvard Square. You think to yourself, "Now my friends will be really impressed with me," and invite them over for brunch. Not because you like them, but just to make them envious of your large lifestyle. Imagine your horror when all they can say is, "What's this old range doing here? Don't you have a Viking stove?"

A *Viking stove?!?* They cost $5,000. The only way you are going to come up with this kind of cash is to join the growing ranks of online entrepreneurs. So you open an Internet bank. An experienced Perl script/flat-file wizard by now, you confidently build a system where all the checking account balances are stored in one file, checking.text, and all the savings balances are stored in another file, savings.text.

A few days later, an unlucky combination of events occurs. Joe User is transferring $10,000 from his savings to his checking account. Judy User is simultaneously depositing $5 into her savings account. One of your Perl scripts successfully writes the checking account flat file with Joe's new, $10,000 higher balance. It also writes the savings account file with Joe's new, $10,000 lower savings balance. However, the script that is processing Judy's deposit started at about the same time and began with the version of the savings file that had Joe's original balance. It eventually finishes and writes Judy's $5 higher balance

Oracle Exhibit 1011
Page 58 of 171

but also overwrites Joe's new lower balance with the old high balance. Where does that leave you? $10,000 poorer, cooking on an old GE range, and wishing you had concurrency control.

After a few months of programming and reading operating systems theory books from the 1960s that deal with mutual exclusion, you've solved your concurrency problems. Congratulations. However, like any good Internet entrepreneur, you're running this business out of your house and you're getting a little sleepy. So you heat up some coffee in the microwave and simultaneously toast a bagel in the toaster oven. The circuit breaker trips. This is the time when you are going to regret having bought that set of Calphalon pots to go with your Viking stove rather than investing in an uninterruptible power supply for your server. You hear the sickening sound of disks spinning down. You scramble to get your server back up and don't really have time to look at the logs and notice that Joe User was back transferring $25,000 from savings to checking. What happened to Joe's transaction?

The good news for Joe is that your Perl script had just finished crediting his checking account with $25,000. The bad news for you is that it hadn't really gotten started on debiting his savings account. You're so busy preparing the public offering for your online business that you fail to notice the loss. But your underwriters eventually do and your plans to sell the bank to the public go down the toilet.

Where does that leave you? Cooking on an old GE range and wishing you'd left the implementation of transactions to professionals.

WHAT DO YOU NEED FOR TRANSACTION PROCESSING?

Data processing folks like to talk about the "ACID Test" when deciding whether or not a database management system is adequate for handling transactions. An adequate system has the following properties:

- *Atomicity*–Results of a transaction's execution are either all committed or all rolled back. All changes take effect, or none do. That means, for Joe User's money transfer, that both his savings and checking balances are adjusted or neither is.
- *Consistency*—The database is transformed from one valid state to another valid state. This defines a transaction as legal only if it obeys user-defined integrity constraints. Illegal transactions aren't allowed, and if an integrity constraint can't be satisfied then the transaction is rolled back. For example, suppose that you define a rule that, after a transfer of more than $10,000 out of the country, a row is added to an audit table so that you can prepare a legally required report for the IRS. Perhaps for performance reasons that audit table is stored on a separate disk from the rest

Oracle Exhibit 1011
Page 59 of 171

of the database. If the audit table's disk is offline and can't be written then the transaction is aborted.

- *Isolation*—The results of a transaction are invisible to other transactions until the transaction is complete. For example, if you are running an accounting report at the same time that Joe is transferring money, the accounting report program will either see the balances before Joe transferred the money or after, but never the intermediate state where checking has been credited but savings not yet debited.
- *Durability*—Once committed (completed), the results of a transaction are permanent and survive future system and media failures. If the airline reservation system computer gives you seat 22A and crashes a millisecond later, it won't have forgotten that you are sitting in 22A and also give it to someone else. Furthermore, if a programmer spills coffee into a disk drive, it will be possible to install a new disk and recover the transactions up to the coffee spill, showing that you had seat 22A.

That doesn't sound too tough to implement, does it? And, after all, one of the most refreshing things about the Web is how it encourages people without formal computer science backgrounds to program. So why not build your Internet bank on a transaction system implemented by an English major who has just discovered Perl?

Because you still need indexing.

FINDING YOUR DATA (AND FAST)

One facet of a database management system is processing inserts, updates, and deletes. This all has to do with putting information into the database. Sometimes it is also nice, though, to be able to get data out. And with popular sites getting 20 hits per second, it pays to be conscious of speed.

Flat files work OK if they are very small. A Perl script can read the whole file into memory in a split second and then look through it to pull out the information requested. But suppose that your online bank grows to have 250,000 accounts. A user types his account number into a Web page and asks for his most recent deposits. You've got a chronological financial transactions file with 25 million entries. Crunch, crunch, crunch. Your server laboriously works through all 25 million to find the ones with an account number that matches the user's. While it is crunching, 25 other users come to the Web site and ask for the same information about their accounts.

You have two choices: Buy an eight-headed DEC Alpha with 2GB of RAM, or build an index file. If you build an index file that maps account numbers to

sequential transaction numbers, then your server won't have to search all 25 million records anymore. However, you have to modify all of your programs that insert, update, or delete from the database to also keep the index current. It works great until two years later when a brand-new MBA arrives from Harvard. She asks your English major cum Perl hacker for a report of all customers who have more than $5,000 in checking, or live in Oklahoma and have withdrawn more than $100 from savings in the last 17 days. It turns out that you didn't anticipate this query so your indexing scheme doesn't speed things up. Your server has to grind through all the data over and over again.

ENTER THE RELATIONAL DATABASE

You are a Web publisher. On the cutting edge. You need the latest and greatest in computer technology. That's why you use, uh, Unix. Yeah. Anyway, even if your operating system harks back to the 1960s, you definitely can't live without the most modern database management system available. Maybe this guy E.F. Codd can help:

> "Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). . . . Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.
>
> "Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for database relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model."

Sounds pretty spiffy, doesn't it? Just like what you need. That's the abstract to *A Relational Model of Data for Large Shared Data Banks*, a paper Codd wrote while working at IBM's San Jose research lab. It was published in the *Communications of the ACM* in June 1970.

Yes, that's right, 1970. What you need to do is move your Web site into the '70s with one of these newfangled relational database management systems. Actually, as Codd notes in his paper, most of the problems we've encountered so far in this chapter were solved in the 1960s by off-the-shelf mainframe software sold by IBM and the "seven dwarves" (as IBM's competitors were known). By the early 1960s, businesses had gotten tired of losing important transactions and manually uncorrupting databases. They began to think that their applications programmers shouldn't be implementing transactions and indexing on an ad hoc basis for each new project. Companies began to buy database management software from computer vendors like IBM. These products worked fairly well but resulted in brittle data models. If you got your data representation correct the first time and your business needs never changed then a 1967-style hierarchical database was great. Unfortunately, if you put a system in place and subsequently needed new indices or a new data format then you might have to rewrite all of your application programs.

From an application programmer's point of view, the biggest innovation in the relational database is that one uses a *declarative* query language: SQL (an acronym for Structured Query Language and pronounced "ess-cue-el" or "sequel"). Most computer languages are *procedural*. The programmer tells the computer what to do, step by step, specifying a procedure. In SQL, the programmer says, "I want data that meets the following criteria," and the RDBMS query planner figures out how to get it. There are two advantages to using a declarative language. The first is that the queries no longer depend on the data representation. The RDBMS is free to store the data however it wants. The second is increased software reliability. It is much harder to have a little bug in an SQL query than in a procedural program. Generally it either describes the data that you want and works all the time or it completely fails in an obvious way.

Another benefit of declarative languages is that less sophisticated users are able to write useful programs. For example, many computing tasks that required professional programmers in the 1960s can be accomplished by non-technical people with spreadsheets. In a spreadsheet, you don't tell the computer how to work out the numbers or in what sequence. You just *declare* "This cell will be 1.5 times the value of that other cell over there."

RDBMSs can run very, very slowly. Depending on whether you are selling or buying computers, this may upset or delight you. Suppose that the system takes 30 seconds to return the data you asked for in your query. Does that mean you have a lot of data? That you need to add some indices? That the RDBMS query planner made some bad choices and needs some hints? Who knows? The RDBMS is an enormously complicated program that you didn't

write and for which you don't have the source code. Each vendor has tracing and debugging tools that purport to help you, but the process is not simple. Good luck figuring out a different SQL incantation that will return the same set of data in less time. If you can't, call 1-800-DIGITAL and say, "I'd like an eight-processor DEC Alpha with 2GB of RAM." Alternatively, you could keep running the non-relational software you used in the 1960s, which is what the airlines do for their reservations systems.

How Does This RDBMS Thing Work?

Database researchers love to talk about relational algebra, n-tuples, normal form, and natural composition, while throwing around mathematical symbols. This patina of mathematical obscurity tends to distract your attention from their bad suits and boring personalities, but is of no value if you just want to use a relational database management system.

In fact, this is all you need to know to be a Caveman Database Programmer: A relational database is a big spreadsheet that several people can update simultaneously.

Each *table* in the database is one spreadsheet. You tell the RDBMS how many columns each row has. For example, in our mailing list database, the table would have two columns: "name" and "email." Each entry in the database consists of one row in this table. An RDBMS is more restrictive than a spreadsheet in that all the data in one column must be of the same type, such as integer, decimal, character string, or date. Another difference between a spreadsheet and an RDBMS is that the rows in an RDBMS are not ordered. You can have a column named "row_number" and ask the RDBMS to return the rows ordered according to the data in this column. But the row numbering is not implicit as it would be with Visicalc or its derivatives such as Lotus 1-2-3 and Excel. If there is a "row_number" column or some other unique identifier for rows in a table, then it is possible for a row in another table to refer to that row by including the value of the unique id.

Here's what some SQL looks like for the mailing list application:

```
create table mailing_list (
        email           text not null primary key,
        name            text
);
```

The table will be called "mailing_list" and will have two columns, both variable length character streams. We've added a couple of integrity constraints on the "email" column. The "not null" will prevent any program from inserting a

row where "name" is specified but "email" is not. After all, the whole point of the system is to send people e-mail so it isn't much use to have a name with no e-mail address. The "primary key" tells the database that this column's value can be used to uniquely identify a row. That means that the system will reject an attempt to insert a row with the same e-mail address as an existing row. This sounds like a nice feature, but it can have some unexpected performance implications. For example, every time anyone tries to insert a row into this table, the RDBMS will have to look at all the other rows in the table to make sure that there isn't already one with the same e-mail address. For a really huge table, that could take minutes. If you had also asked the RDBMS to create an index for mailing_list on "email" then the check becomes almost instantaneous. However, the integrity constraint still slows you down because every update to the mailing_list table will also require an update to the index and there you'll be doing twice as many writes to the hard disk.

That is the joy and the agony of SQL. Inserting two innocuous looking words can cost you a factor of 1,000 in performance. Then inserting a sentence (to create the index) can bring you back so that it is only a factor of two or three.

Anyway, now that we've executed the Data Definition Language "create table" statement, we can move on to *Data Manipulation Language:* an INSERT.

```
insert into mailing_list (name, email)
values ('Philip Greenspun','philg@mit.edu');
```

Note that we specify into which columns we are inserting. That way if someone comes along later and does

```
alter table mailing_list add column phone_number text;
```

then our INSERT will still work. Note also that the string quoting character in SQL is a single quote. Hey, it was the '70s. If you visit the newsgroup comp.databases right now, I'll be that you can find someone asking, "How do I insert a string containing a single quote into an RDBMS?" Here's one I harvested from http://www.dejanews.com just a few minutes ago:

```
Subject:      HELP/BUG? Single Quotes in SQL
From:         "James Boswell" <j.boswell@ukonline.co.uk>
Date:         1997/02/11
Message-Id:   <01bc182d$98f21c50$8429bfc7@teme>
Newsgroups:   comp.lang.pascal.delphi.databases
[More Headers]

When submitting an SQL query through a Delphi TQuery component to an
Interbase database (using BDE), is there a problem with single quotes in parameters?
```

```
For example:
I want to put the following into a database field:

  John's record

And here's the code that I started with:

  strValue := 'John' + Chr(39) + 's record';
  with MyQueryObject do
  begin
      SQL.Clear;
      SQL.Add('INSERT INTO MYTABLE (MyStrFld) VALUES (' + strValue + ');
      Open;
  end;

The query looks like:

  INSERT INTO MYTABLE (MyStrFld) VALUES ('John's record');

which of course confuses the SQL interpreter.  So, normal programming
convention is to replace one quote character with two quote characters:

INSERT INTO MYTABLE (MyStrFld) VALUES ('John''s record');

But this puts two quotes into the database!!!

What is the convention in SQL, Interbase and BDE for allowing single quote
characters in paramters?

James Boswell
```

If you want to avoid humiliating yourself on Usenet, remember that doubling the single quote will let you insert one into the database . . .

```
insert into mailing_list (name, email)
values ('Michael O''Grady','ogrady@fastbuck.com');
```

So you've done DDL and DML statements. At last you are ready to experience the awesome power of the third type of SQL statement: a SELECT. Want your data back?

```
select * from mailing_list;
```

If you were to type this query into a standard shell-style RDBMS client program, like SQL*PLUS for Oracle or, in this case, MSQL for Illustra, here's what you'd get:

```
* select * from mailing_list;
-----------------------------------------
|email                |name               |
-----------------------------------------
|philg@mit.edu        |Philip Greenspun   |
|ogrady@fastbuck.com  |Michael O'Grady    |
-----------------------------------------
2 rows selected
```

Now you realize that you want to also store phone numbers for these folks. Since this is the Internet, many of your users will have their pants hanging around their knees under the weight of their cell phones, beepers, and other personal communication accessories. So if you just added work_phone and home_phone columns, you wouldn't be able to accommodate the wealth of information users might want to give you.

The clean database-y way to do this is to define another table.

```
create table phone_numbers (
  email         text not null,
  number_type   text check (number_type in ('work','home','cell','beeper')),
  number        text
);
```

Note that in this table the email column is *not* a primary key. That's because we want to allow multiple rows with the same e-mail address. If you are hanging around with a database nerd friend, you can say that there is a *relationship* between the rows in the phone_numbers table and the mailing_list table. In fact, you can say that it is a *many-to-one relation* because many rows in the phone_numbers table may correspond to only one row in the mailing_list table. If you spend enough time thinking about and talking about your database in these terms, two things will happen:

1. You'll get an A in an RDBMS course at a mediocre state university.
2. You'll pick up readers of *Psychology Today* who think you are sensitive and caring because you are always talking about relationships.

Another item worth noting about our two-table data model is that we do not store the user's name in the phone_numbers table. That would be redundant with the mailing_list table and potentially self-redundant as well, as, for example, if "robert.loser@fastbuck.com" says he is "Robert Loser" when he

types in his work phone and then "Rob Loser" when he puts in his beeper number, and "Bob Lsr" when he puts in his cell phone number while typing on his laptop's cramped keyboard. A database nerd would say that this data model is consequently in "Third Normal Form." Everything in each row in each table depends only on the primary key and nothing is dependent on only part of the key. The primary key for the phone_numbers table is the combination of email and number_type. If you had the user's name in this table, it would depend only on the email portion of the key.

Anyway, enough database nerdism. Let's populate the phone_numbers table:

```
insert into phone_numbers values ('ogrady@fastbuck.com','work','(800) 555-1212');
insert into phone_numbers values ('ogrady@fastbuck.com','home','(617) 495-6000');
insert into phone_numbers values ('philg@mit.edu','work','(617) 253-8574');
insert into phone_numbers values ('ogrady@fastbuck.com','beper','(617) 222-3456');
```

Note that we have used an evil SQL shortcut and not specified the columns into which we are inserting data. The system defaults to using all the columns in the order that they were defined. Except for prototyping and playing around, I don't recommend ever using this shortcut.

The first three INSERTs work fine, but what about the last one, where Mr. O'Grady misspelled "beeper?"

```
X23C00:integrity constraint violation:
 Constraint Name _CHECK_CONSTRAINT_ON_INSERT_TO_public_phone_numbers_
```

We asked Illustra at table definition time to check (number_type in ('work','home','cell','beeper')) and it did. The database cannot be left in an inconsistent state.

Let's say you want all of our data out. E-mail, full name, phone numbers. The most obvious query to try is a *join*.

```
select *
from mailing_list, phone_numbers;
-------------------------------------------------------------------------
|email              |number_type|number        |email        |name           |
-------------------------------------------------------------------------
|ogrady@fastbuck.com|work       |(800) 555-1212|philg@mit.edu|Philip Greenspun|
|ogrady@fastbuck.com|home       |(617) 495-6000|philg@mit.edu|Philip Greenspun|
|philg@mit.edu      |work       |(617) 253-8574|philg@mit.edu|Philip Greenspun|
|ogrady@fastbuck.com|work       |(800) 555-1212|ogrady@fastb*|Michael O'Grady |
|ogrady@fastbuck.com|home       |(617) 495-6000|ogrady@fastb*|Michael O'Grady |
|philg@mit.edu      |work       |(617) 253-8574|ogrady@fastb*|Michael O'Grady |
-------------------------------------------------------------------------
6 rows selected
```

Yow! What happened? There are only two rows in the mailing_list table and three in the phone_numbers table. Yet here we have six rows back. This is how joins work. They give you the *Cartesian product* of the two tables. Each row of one table is paired with all the rows of the other table in turn. So if you join an *n*-row table with an *m*-row table, you get back a result with *n* m* rows. In real databases, *n* and *m* can be up in the millions so it is worth being a little more specific as to which rows you want:

```
select *
from mailing_list, phone_numbers
where mailing_list.email = phone_numbers.email;
-------------------------------------------------------------------------
|email         |number_type|number         |email         |name          |
-------------------------------------------------------------------------
|philg@mit.edu|work        |(617) 253-8574|philg@mit.edu|Philip Greenspun|
|ogrady@fastb*|work        |(800) 555-1212|ogrady@fastb*|Michael O'Grady |
|ogrady@fastb*|home        |(617) 495-6000|ogrady@fastb*|Michael O'Grady |
-------------------------------------------------------------------------
3 rows selected
```

Probably more like what you had in mind. Refining your SQL statements in this manner can sometimes be more exciting. For example, let's say that you want to get rid of Philip Greenspun's phone numbers but aren't sure of the exact syntax.

```
delete from phone_numbers;
3 rows deleted
```

Oops. Yes, this does actually delete *all* the rows in the table. You probably wish you'd typed

```
delete from phone_numbers where email = 'philg@mit.edu';
```

but it is too late now. I guess there is one more SQL statement that is worth learning. Suppose that I move to Hollywood to realize my long-standing dream of becoming a major motion picture producer. Clearly a change of name is in order, though I'd be reluctant to give up the e-mail address I've had since 1976. Here's the SQL:

```
* update mailing_list set name = 'Phil-baby Greenspun' where email =
'philg@mit.edu';

one row updated
```

```
* select * from mailing_list;
-----------------------------
|email              |name     |
-----------------------------
|ogrady@fastbuck.com|Michael O'Grady|
|philg@mit.edu      |Phil-baby Gree*|
-----------------------------
2 rows selected
```

As with DELETE, I don't recommend playing around with UPDATE statements unless you have a WHERE clause at the end.

SQL THE HARD WAY

So you've gotten your pathetic Web site up and running and are proud of yourself for your wimpy little SELECTs. You had planned to live on the advertising revenue from your site, but find that $1.37 a month doesn't go very far in Manhattan. So you say, "At least I've become a database wizard; I can work in the financial industry." The next day, you're interviewing for that $200,000-a-year database hacking job at CitiCorp. Everything is going smoothly until they ask you how you'd investigate a performance problem with a "self-join with subquery." Self-join? Subquery? Maybe you have a little more to learn about RDBMS. Here's an example drawn from my real-life suffering with Illustra.

My site would hand out a unique session key to every new user who arrived at the site. These keys were just an integer sequence. I realized that I could keep track of how many users came to the site by simply inserting an audit row every day showing the value of the session key generator. Here's the history table I created:

```
* create table history (
    sample_time     timestamp,     -- when did the cookie have the value
    sample_value    integer
);

* select * from history order by sample_time;
---------------------------------------
|sample_time              |sample_value |
---------------------------------------
|1996-02-15 19:02:08.000000|75000       |
|1996-02-16 19:02:08.000000|76000       |
|1996-02-17 19:02:08.000000|77000       |
|1996-02-18 19:02:08.347617|77276       |
---------------------------------------
```