



Chapter 12: Database Management Systems

by [Philip Greenspun](#), part of [Philip and Alex's Guide to Web Publishing](#)

Revised July 2003

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 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, it is covered in the most depth and the chapter includes 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 are kept in big unstructured named clumps called *files*. The great thing about the file system is its invisibility. You probably didn't purchase it separately, you might not be aware of its existence, you won't have to run an ad in the newspaper for a *file system administrator* with 5+ years of experience, and it will pretty much work as advertised. All you need to do with a file system is back it up to tape every day or two.

Despite its unobtrusiveness, the file system on a Macintosh, Unix, or Windows machine is capable of storing any data that may be represented in digital form. For example, suppose that you are storing a mailing list in a file system file. If you accept the limitation that no e-mail address or person's name can contain a newline character, 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 never to try storing a newline or vertical bar, you can keep your data in this "flat file." Searching can be slow and expensive, though. What if you want to see if "philg@mit.edu" is on the mailing list? Your computer must 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. For a personal Web site the consequences of a corrupted database aren't very serious: you spend your nights and weekends fixing up the flat files with a text editor. But imagine that you build an online bank with Perl scripts and flat files.



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 but also overwrites Joe's new lower balance with the old high balance. Where does that leave you?

\$10,000 poorer 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, let's suppose that this is an Internet business circa 2003 and therefore you're running it out of your mom's house with the server under the living room sofa. You're feeling sleepy so you heat up some coffee in the microwave and simultaneously toast a bagel in the toaster oven. The circuit breaker trips. You hear the sickening sound of disks spinning down. You scramble to get your server back up and, glancing at the logs 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. Maybe it is time to look at what the transaction processing experts have come up with over the last 50 years...



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 are.

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 of the database. If the audit table's disk is off-line and can't be written, 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? A "mere matter of programming" as our friend Jin likes to say. Well, 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 medium popularity sites getting 20 requests per second, it pays to be conscious of speed.

Flat files work okay 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 on-line 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 a 1000-processor supercomputer or build an index file. If you build an index file that maps account numbers to sequential transaction numbers, 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 so that they also keep the index current.

This works great until two years later when a brand new MBA arrives from Harvard. She asks 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 data base 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 and is available from <http://www.acm.org/classics/nov95/toc.html> (ACM claims to represent the best practices that a business person could expect from a computing professional; their online version of Codd's paper was half-finished in 1995 and features a "coming soon" note for the rest... is it any wonder that U.S. corporations are moving IT jobs to India and China? If a computer professional is going to do a half-assed job it doesn't make sense to pay him more than \$10/hour.).

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 (RDBMS). 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 meet 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 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."

RDBMSes 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 your hardware supplier and ask them to send you 16 more CPUs with maximum RAM. Alternatively, you can 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.**



How can multiple people use a spreadsheet such as Microsoft *Excel* at the same time? Would they take turns at the keyboard and mouse? Fight over the keyboard and mouse? Crowd their heads together in front of the display?

We need to get rid of the mouse/keyboard/windows idea of interaction. It doesn't make sense when there are many simultaneous users. Instead imagine that the database management system is sitting by itself in a dark closet. Users write down their requests on strips of paper and slip them under the door. A request might be "create a table", "insert a row into a table", "update an existing row in a table", "give me a report of the information contained in all the rows in a table that meet the following criteria...". If a request requires an answer, the database chews on the question for awhile, then pushes a paper report back out underneath the door.

Let's examine how this works in greater detail.

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 has two columns: *name* and *e-mail*. 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, e.g., 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 a spreadsheet program. If you do define a `row_number` column or some other unique identifier for rows in a table, it becomes 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          varchar(100) not null primary key,
    name          varchar(100)
);
```

The table will be called `mailing_list` and will have two columns, both variable length character strings. 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 there isn't much value in having 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 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, but 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 therefore 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 1000 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. (Note that many RDBMS implementations, including Oracle, automatically define an index on a column that is constrained to be unique.)

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 (phone_number varchar(20));
```

(the Oracle syntax for adding a column), 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 [newsgroups beginning with comp.databases](#) right now, you can probably find someone asking "How do I insert a string containing a single quote into an RDBMS?" Here's one harvested back in 1998:

demaagd@cs.hope.edu (David DeMaagd) wrote:

```
>hwo can I get around the fact that the ' is a reserved character in
>SQL Syntax? I need to be able to select/insert fields that have
>apostrophies in them. Can anyone help?
```

You can use two apostrophes '' and SQL will treat it as one.

```
=====
Pete Nelson      | Programmers are almost as good at reading
weasel@ecis.com | documentation as they are at writing it.
=====
```

We'll take Pete Nelson's advice and double the single quote in "O'Grady":

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

Having created a table and inserted some data, at last we are ready to experience the awesome power of the SQL SELECT. Want your data back?

```
select * from mailing_list;
```

If you typed this query into a standard shell-style RDBMS client program, for example Oracle's SQL*PLUS, you'd get ... a horrible mess. That's because you told Oracle that the columns could be as wide as 100 characters (`varchar(100)`). Very seldom will you need to store e-mail addresses or names that are anywhere near as long as 100 characters. However, the solution to the "ugly report" problem is not to cut down on the maximum allowed length in the database. You don't want your system failing for people who happen to have exceptionally long names or e-mail addresses. The solution is either to use a fancier tool for querying your database or to give SQL*Plus some hints for preparing a report:

```
SQL> column email format a25
SQL> column name format a25
SQL> column phone_number format a12
SQL> set feedback on
SQL> select * from mailing_list;
```

EMAIL	NAME	PHONE_NUMBER
-----	-----	-----
philg@mit.edu	Philip Greenspun	
ogrady@fastbuck.com	Michael O'Grady	

```
2 rows selected.
```

Note that there are no values in the `phone_number` column because we haven't set any. As soon as we do start to add phone numbers, we realize that our data model was inadequate. This is the Internet and Joe Typical User will have his pants hanging around his knees under the weight of a cell phone, beeper, and other personal communication accessories. One phone number column is clearly inadequate and even `work_phone` and `home_phone` columns won't accommodate the wealth of information users might want to give us. The clean database-y way to do this is to remove our `phone_number` column from the `mailing_list` table and define a helper table just for the phone numbers. Removing or renaming a column turns out to be impossible in Oracle 8, so we

```
drop table mailing_list;

create table mailing_list (
    email          varchar(100) not null primary key,
    name          varchar(100)
);

create table phone_numbers (
    email          varchar(100) not null references mailing_list,
    number_type    varchar(15) check (number_type in ('work', 'home', 'cell', 'b
    phone_number   varchar(20)
);
```

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. [see "Using the Internet to Pick up Babes and/or Hunks" at <http://philip.greenspun.com/wtr/getting-dates> before following any of the author's dating advice]

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, if, for example, "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 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 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:

```
SQL> insert into phone_numbers values ('ograde@fastbuck.com','work','(800) 555-121
ORA-02291: integrity constraint (SCOTT.SYS_C001080) violated - parent key not found
```

Oops! When we dropped the `mailing_list` table, we lost all the rows. The `phone_numbers` table has a referential integrity constraint ("references `mailing_list`") to make sure that we don't record e-mail addresses for people whose names we don't know. We have to first insert the two users into `mailing_list`:

```
insert into mailing_list (name, email)
values ('Philip Greenspun','philg@mit.edu');
insert into mailing_list (name, email)
values ('Michael O'Grady','ograde@fastbuck.com');

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

Note that the last four INSERTs use an evil SQL shortcut and don't specify 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"?

```
ORA-02290: check constraint (SCOTT.SYS_C001079) violated
```

We asked Oracle 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 we want all of our data out. Email, full name, phone numbers. The most obvious query to try is a *join*.

```
SQL> select * from mailing_list, phone_numbers;
```

EMAIL	NAME	EMAIL	TYPE	NUMBER
philg@mit.edu	Philip Greenspun	ogrady@fastbuck.	work	(800) 555-1212
ogrady@fastbuck.	Michael O'Grady	ogrady@fastbuck.	work	(800) 555-1212
philg@mit.edu	Philip Greenspun	ogrady@fastbuck.	home	(617) 495-6000
ogrady@fastbuck.	Michael O'Grady	ogrady@fastbuck.	home	(617) 495-6000
philg@mit.edu	Philip Greenspun	philg@mit.edu	work	(617) 253-8574
ogrady@fastbuck.	Michael O'Grady	philg@mit.edu	work	(617) 253-8574

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	NAME	EMAIL	TYPE	NUMBER
ogrady@fastbuck.	Michael O'Grady	ogrady@fastbuck.	work	(800) 555-1212
ogrady@fastbuck.	Michael O'Grady	ogrady@fastbuck.	home	(617) 495-6000
philg@mit.edu	Philip Greenspun	philg@mit.edu	work	(617) 253-8574

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.

```
SQL> delete from phone_numbers;
```

3 rows deleted.

Oops. Yes, this does actually delete *all* the rows in the table and is perhaps why Oracle makes the default in SQL*Plus that you're inside a transaction. I.e., nothing happens that other users can see until you type "commit". If we type "rollback", therefore, we have an opportunity to enter the statement that we wished we'd typed:

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

There is one more SQL statement that is worth learning. In the 20th century parents encouraged their children to become software engineers rather than trying their luck in Hollywood. In the 21st century, however, it might be safer to pitch scripts than to sit at a computer waiting for your job to be outsourced to a Third World country. Suppose therefore that Philip Greenspun gives up programming and heads out to Beverly Hills. Clearly a change of name is in order and here's the SQL:

```
SQL> update mailing_list set name = 'Phil-baby Greenspun' where email = 'philg@mit
```

1 row updated.

```
SQL> select * from mailing_list;
```

EMAIL	NAME
-----	-----

philg@mit.edu Phil-baby Greenspun
 ogrady@fastbuck.com Michael O'Grady

2 rows selected.

As with DELETE, it is not a good idea to play around with UPDATE statements unless you have a WHERE clause at the end.

SQL the Hard Way

Problems that are difficult in imperative computer languages can be very simple in a declarative language such as SQL and vice versa. Let's look at an example of building a report of how many users come to a site every day. Back in 1995 we built a site that would hand out a unique session key to every new user. Since the keys were an ascending sequence of integers, we realized that we 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:



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

insert into history values (to_date('1998-03-01 23:59:00', 'YYYY-MM-DD HH24:MI:SS'))
insert into history values (to_date('1998-03-02 23:59:00', 'YYYY-MM-DD HH24:MI:SS'))
insert into history values (to_date('1998-03-03 23:59:00', 'YYYY-MM-DD HH24:MI:SS'))
insert into history values (to_date('1998-03-04 23:59:00', 'YYYY-MM-DD HH24:MI:SS'))

select * from history order by sample_time;
```

SAMPLE_TIM	SAMPLE_VALUE
1998-03-01	75000
1998-03-02	76000
1998-03-03	77000
1998-03-04	78000

Note: the Oracle DATE data type is able to represent time down to one-second precision, sort of like an ANSI TIMESTAMP(0); in Oracle 9i and newer you'd probably use the timestamp type.

Now that we are accumulating the data, it should be easy to write a report page, eh? It turns out to be very easy to extract the rows of a table in order, as in the last SELECT above. However, it is impossible for a row to refer to "the last row in this SELECT". You could have write a C#, Java, PL/SQL, or VB procedure to walk through the rows in order, just setting things up on the first pass through the loop and then doing the appropriate subtraction for subsequent rows. However, running into the arms of an imperative computer language is considered untasteful in the SQL world.

In SQL you start by thinking about what you want and working backward. If the history table has N rows, we want an interval table with N-1 rows. Each row should have the start time, end time, time interval, and cookie interval. Any time that you need information from two different rows in the database, the way to get it is with a JOIN. Since there is only one underlying table, though, this will have to be a self-join:

```
SQL> select h1.sample_time, h2.sample_time
from history h1, history h2;
```

```

SAMPLE_TIM SAMPLE_TIM
-----
1998-03-04 1998-03-04
1998-03-01 1998-03-04
1998-03-02 1998-03-04
1998-03-03 1998-03-04
1998-03-04 1998-03-01
1998-03-01 1998-03-01
1998-03-02 1998-03-01
1998-03-03 1998-03-01
1998-03-04 1998-03-02
1998-03-01 1998-03-02
1998-03-02 1998-03-02
1998-03-03 1998-03-02
1998-03-04 1998-03-03
1998-03-01 1998-03-03
1998-03-02 1998-03-03
1998-03-03 1998-03-03

```

16 rows selected.

A note about syntax is in order here. In an SQL FROM list, one can assign a correlation name to a table. In this case, h1 and h2 are assigned to the two copies of history from which we are selecting. Then we can refer to h1.sample_time and get "the sample_time column from the first copy of the history table."

The main problem with this query, though, has nothing to do with syntax. It is the fact that we have 13 rows too many. Instead of N-1 rows, we specified the Cartesian product and got NxN rows. We've successfully done a self-join and gotten all the pairings we need, but also all possible other pairings. Now it is time to specify which of those pairings we want in our final report:

```

select h1.sample_time as s1,
       h2.sample_time as s2
from history h1, history h2
where h2.sample_time > h1.sample_time;

```

```

S1          S2
-----
1998-03-01 1998-03-04
1998-03-02 1998-03-04
1998-03-03 1998-03-04
1998-03-01 1998-03-02
1998-03-01 1998-03-03
1998-03-02 1998-03-03

```

6 rows selected.

Note first that we've given correlation names to the columns as well, resulting in a report labelled with "s1" and "s2" (in a database with a smarter SQL parser than Oracle's, we could also use these as shorthand in the WHERE clause). The critical change here is the WHERE clause, which states that we only want intervals where s2 is later than s1. That kills off 10 of the rows from the Cartesian product but there are still three unwanted rows, e.g., the pairing of 1998-03-01 and 1998-03-04. We only want the pairing of 1998-03-01 and 1998-03-02. Let's refine the WHERE clause:

```

select h1.sample_time as s1,
       h2.sample_time as s2
from history h1, history h2
where h2.sample_time = (select min(h3.sample_time)
                       from history h3
                       where h3.sample_time > h1.sample_time)

```

```
order by h1.sample_time;
```

```
S1          S2
-----
1998-03-01 1998-03-02
1998-03-02 1998-03-03
1998-03-03 1998-03-04
```

```
3 rows selected.
```

Note that we are now asking the database, for each of the potential row pairings, to do a subquery:

```
select min(h3.sample_time)
from history h3
where h3.sample_time > h1.sample_time
```

This will scan the history table yet again to find the oldest sample that is still newer than s1. In the case of an unindexed history table, this query should probably take an amount of time proportional to the number of rows in the table cubed (N^3). If we'd done this procedurally, it would have taken time proportional to $N \cdot \log(N)$ (the limiting factor being the sort for the ORDER BY clause). There are a couple of lessons to be learned here: (1) Sometimes declarative languages can be difficult to use and vastly less efficient than procedural languages and (2) it is good to have a fast database server.

Given the correct row pairings, it is easy to add syntax to the SELECT list to subtract the times and cookie values.

```
select h1.sample_time as s1,
       h2.sample_time as s2,
       h2.sample_time - h1.sample_time as gap_time,
       h2.sample_value - h1.sample_value as gap_cookie
from history h1, history h2
where h2.sample_time = (select min(h3.sample_time)
                       from history h3
                       where h3.sample_time > h1.sample_time)
order by h1.sample_time;
```

```
S1          S2          GAP_TIME GAP_COOKIE
-----
1998-03-01 1998-03-02          1         1000
1998-03-02 1998-03-03          1         1000
1998-03-03 1998-03-04          1         1000
```

```
3 rows selected.
```

Formulating SQL queries can be an art and most programmers need time and experience to get good at thinking declaratively.

Brave New World

Training an African Grey parrot to function as an information systems manager can be very rewarding. The key sentence is "We're pro-actively leveraging our object-oriented client/server database to target customer service during reengineering." In the 1980s db world, the applicable portion of this sentence was "client/server" (see next chapter). In the Brave New World of database management systems, the key phrase is "object-oriented."

Object systems contribute to software reliability and compactness by allowing programmers to factor their code into



chunks that are used as widely as possible. For example, suppose that you are building a catalog Web site to sell magazines, videos, books, and CDs. It might be worth thinking about the data and functions that are common to all of these and encapsulating them in a product class. At the product level, you'd define characteristics such as `product_id`, `short_name`, and `description`. Then you'd define a magazine subclass that inherited all the behavior of product and added things like `issues_per_year`.

Programmers using modern computer languages like Smalltalk and Lisp have been doing this since the mid-1970s, but the idea has only recently caught on in the RDBMS world. Here are some table definitions for the Illustra system, a derivative of the U.C. Berkeley Postgres research project:

```
create table products of new type product_t
(
    product_id          integer not null primary key,
    short_name          text not null,
    description         text
);
```

Then we define new types and tables that inherit from products...

```
create table magazines of new type magazine_t (
    issues              integer not null,
    foreign_postage    decimal(7,2),
    canadian_postage   decimal(7,2)
)
under products;
create table videos of new type video_t (
    length_in_minutes  integer
)
under products;
```

Having defined our data model, we can load some data.

```
* insert into magazines (product_id,short_name,description,issues)
values (0,'Dissentary','The result of merging Dissent and Commentary',12);
* insert into videos (product_id,short_name,description,length_in_minutes)
values (1,'Sense and Sensibility','Chicks dig it',110);
* select * from products;
```

```
-----
|product_id  |short_name          |description      |
-----
|1           |Sense and Sensibility|Chicks dig it  |
|0           |Dissentary          |The result o*  |
-----
```

Suppose that our pricing model is that magazines cost \$1.50/issue and videos cost 25 cents a minute. We want to hide these decisions from programs using the data.

```
create function find_price(product_t) returns numeric with (late)
as
return 5.50;
```

So a generic product will cost \$5.50.

```
create function find_price(magazine_t) returns numeric
as
return $1.issues * 1.50;
create function find_price(video_t) returns numeric
as
return $1.length_in_minutes * 0.25;
```

The appropriate version of the function `find_price` will be invoked depending on the type of the row.

```
* select short_name, find_price(products) from products;
```

short_name	find_price
Sense and Sensibility	27.50
Dissentary	18.00

This doesn't sound so impressive, but suppose you also wanted a function to prepare a special order code by concatenating `product_id`, price, and the first five characters of the title.

```
create function order_code(product_t) returns text
as
return $1.product_id::text ||
    '-' ||
    trim(leading from find_price($1)::text) ||
    '-' ||
    substring($1.short_name from 1 for 5);
* select order_code(products) from products;
```

order_code
1--27.50--Sense
0--18.00--Disse

This function, though trivial, is already plenty ugly. The fact that the `find_price` function dispatches according to the type of its argument allows a single `order_code` to be used for all products.

This Brave New World sounds great in DBMS vendor brochures, but the actual experience isn't always wonderful. Back in 1995, we were using Illustra and built ourselves a beautiful table hierarchy more or less as described above. Six months later we needed to add a column to the `products` table. E.F. Codd understood back in 1970 that data models had to grow as business needs change. But the Illustra folks were so excited by their object extensions that they forgot. The system couldn't add a column to a table with dependent subclasses. What should we do, we asked the support folks? "Dump the data out of all of your tables, drop all of them, rebuild them with the added column, then load all of your data back into your tables."

Uh, thanks...

Braver New World

If you really want to be on the cutting edge, you can use a bona fide object database, like ObjectStore (<http://www.objectstore.net>). These systems persistently store the sorts of object and pointer structures that you create in a Smalltalk, Common Lisp, C++, or Java program. Chasing pointers and certain kinds of transactions can be 10 to 100 times faster than in a relational database. If you believed everything in the object database vendors' literature, then you'd be surprised that Larry Ellison still has \$100 bills to fling to peasants as he roars overhead in his Gulfstream jet. The relational database management system should have been crushed long ago under the weight of this superior technology, introduced with tremendous hype in the early 1980s.

After 20 years, the market for object database management systems is about \$100 million a year, less than 1 percent the size of the relational



database market. Why the fizzle? Object databases bring back some of the bad features of 1960s pre-relational database management systems. The programmer has to know a lot about the details of data storage. If you know the identities of the objects you're interested in, then the query is fast and simple. But it turns out that most database users don't care about object *identities*; they care about object *attributes*. Relational databases tend to be faster and better at producing aggregations based on attributes.



Keep in mind that if object databases ever *did* become popular, it would make using Java or C# as a page scripting language much more attractive. Currently the fancy type systems of advanced computer languages aren't very useful for Internet application development. The only things that can be stored persistently, i.e., from page load to page load, in an RDBMS are numbers, dates, and strings. In theory it is possible to write a complex Java program that does an SQL query, creates a huge collection of interlinked objects, serves a page to the user, then exits. But why would you? That collection of interlinked objects must be thrown out as soon as the page is served and then built up again the next time that user requests a page, which could be 1 second later or 2 days later. If on the other hand the objects created by a C# or Java program were efficiently made persistent it would make applying these big hammers to Web scripting a lot less ridiculous.

Choosing an RDBMS Vendor

You'd think that it wouldn't matter which RDBMS you use behind a Web site because every RDBMS comes with a standard SQL front end. In fact the vendors' flavors of SQL are different enough that porting from one RDBMS to another is generally a nightmare. Plan to live with your choice for 5 or 10 years.

Here are some factors that are important in choosing an RDBMS to sit behind a Web site:

1. cost/complexity to administer
2. lock management system
3. full-text indexing option
4. maximum length of VARCHAR data type
5. support



Cost/Complexity to Administer

In the bad old days you might install Oracle on a \$500,000 computer and accept all the defaults, then find that the rollback segment was about 15 MB. That means that you couldn't do a transaction updating more than 15 MB of data at a time on a computer with 200 GB of free disk space. Oracle would not, by default, just grab some more disk space.

Sloppy RDBMS administration is one of the most common causes of downtime at sophisticated sites. If you aren't sure that you'll have an experienced staff of database administrators to devote to your site, then you might want to consider whether an "Enterprise RDBMS" is really for you. The Big Three RDBMSes are IBM's DB2, Microsoft SQL Server, and Oracle. All offer tremendous flexibility in configuration with an eye towards building 64-CPU systems that process absurd numbers of simultaneous transactions. All claim that their latest and greatest management tools make installing and administering the system is so easy a 10-year-old could do it. If these claims were true the local Starbucks would be clogged



with unemployed database administrators. Midget competition is provided by the open-source PostgreSQL, which can't be scaled up to the large sizes and performance of the commercial RDBMSes but is inherently fairly simple to install and maintain (and it is free!). MySQL is popular among Linux users and because it has a SQL front-end it is often confused with an RDBMS. In fact it does not provide the ACID transaction guarantees and therefore would more properly be lumped in with Microsoft *Access* and similar desktop tools.

Lock Management System

Relational database management systems exist to support concurrent users. If you don't have people simultaneously updating information, you are probably better off with a simple Perl script, Microsoft *Access*, or MySQL rather than a commercial RDBMS (i.e., 100 MB of someone else's C code).

All database management systems handle concurrency problems with locks. Before an executing statement can modify some data, it must grab a lock. While this lock is held, no other simultaneously executing SQL statement can update the same data. In order to prevent another user from reading half-updated data, while this lock is held, no simultaneously executing SQL statement can even *read* the data.

Readers must wait for writers to finish writing. Writers must wait for readers to finish reading.



This kind of system, called *pessimistic locking*, is simple to implement, works great in the research lab, and can be proven correct mathematically. The only problem with this approach is that it often doesn't work in the real world of hurried developers and multiple simultaneous users. What can happen is that an admin page on an ecommerce site, for example, contains a reporting query that takes an hour to run and touches all the rows in the users and orders tables. While this query is running none of the users of the public pages can register or place orders.

With the Oracle RDBMS, *readers never wait for writers and writers never wait for readers*. If a SELECT starts reading at 9:01 and encounters a row that was updated (by another session) at 9:02, Oracle reaches into a rollback segment and digs up the pre-update value for the SELECT (this preserves the *Isolation* requirement of the ACID test). A transaction does not need to take locks unless it is modifying a table and, even then, only takes locks on the specific rows that are to be modified.

This is the kind of RDBMS locking architecture that you want for a Web site and, as of 2003, it is provided only by Oracle and Postgres.

Full-text Indexing Option

Suppose that a user says he wants to find out information on "dogs". If you had a bunch of strings in the database, you'd have to search them with a query like

```
select * from magazines where description like '%dogs%';
```

This requires the RDBMS to read every row in the table, which is slow. Also, this won't turn up magazines whose description includes the word "dog".

A full-text indexer builds a data structure (the index) on disk so that the RDBMS no longer has to scan the entire table to find rows containing a particular word or combination of words. The software is smart enough to



be able to think in terms of word stems rather than words. So "running" and "run" or "dog" and "dogs" can be interchanged in queries. Full-text indexers are also generally able to score a user-entered phrase against a database table of documents for relevance so that you can query for the most relevant matches.



Finally, the modern text search engines are very smart about how words relate. So they might deliver a document that did *not* contain the word "dog" but did contain "Golden Retriever". This makes services like classified ads, discussion forums, etc., much more useful to users.

Relational database management system vendors are gradually incorporating full-text indexing into their products. Sadly, there is no standard for querying using this index. Thus, if you figure out how to query Oracle Text for "rows relating to 'running' or its synonyms", the SQL syntax will not be useful for asking the same question of Microsoft SQL Server with its full-text indexing option.

Maximum Length of VARCHAR Data Type

You might naively expect a relational database management system to provide abstraction for data storage. After defining a column to hold a character string, you'd expect to be able to give the DBMS a ten-character string or a million-character string and have each one stored as efficiently as possible.

In practice, current commercial systems are very bad at storing unexpectedly long data, e.g., Oracle only lets you have 4,000 characters in a VARCHAR. This is Okay if you're building a corporate accounting system but bad for a public Web site. You can't really be sure how long a user's classified ad or bulletin board posting is going to be. The SQL standard provides for a LONG data type to handle this kind of situation and modern database vendors often provide character large-objects (CLOB). These types theoretically allow you to store arbitrarily large data. However, in practice there are so many restrictions on these columns that they aren't very useful. For example, you can't use them in a SQL WHERE clause and thus the above "LIKE '%dogs%'" would be illegal. You can't build a standard index on a LONG column. You may also have a hard time getting strings into or out of LONG columns. The Oracle SQL parser only accepts string literals up to 4,000 characters in length. After that you have to use special C API calls.



Currently, PostgreSQL seems to be the leader in this area. You can declare a column to be character varying or text and store a string up to about 1 GB in size.

Paying an RDBMS Vendor

This is the part that hurts. The basic pricing strategy of database management system vendors is to hang the user up by his heels, see how much money falls out, take it all and then ask for another \$50,000 for "support". Ideally, they'd like to know how much your data are worth and how much profit you expect from making them available and then extract all of that profit from you. In this respect, they behave like the classical price-discriminating profit-maximizing monopoly from Microeconomics 101.



Classically an RDBMS license was priced per user. Big insurance companies with 1000 claims processors would pay more than small companies with 5. The Web confused the RDBMS vendors. On the one hand, the server was accessible to anyone anywhere in the world. Thus, the fair arrangement would be a \$64,000 per CPU unlimited user license. On the other hand, not too many Web publishers actually had \$64,000 per CPU lying around in their checking accounts. So the RDBMS vendors would settle for selling a 5-user or 8-user license.



If you can't stomach the prices of commercial systems, take a long hard look at PostgreSQL. Developing a credible threat to use PostgreSQL may result in some pricing flexibility from commercial RDBMS salesmen.

Performance

Every RDBMS vendor claims to have the world's fastest system. No matter what you read in the brochures, be assured that any RDBMS product will be plenty slow. Back in the 1990s we had 70,000 rows of data to insert into Oracle8. Each row contained six numbers. It turned out that the data wasn't in the most convenient format for importation so we wrote a one-line Perl script to reformat it. It took less than one second to read all 70,000 rows, reformat them, and write them back to disk in one file. Then we started inserting them into an Oracle 8 table, one SQL insert at a time. It took about 20 minutes (60 rows/second). This despite the fact that the table wasn't indexed and therefore Oracle did not have to update multiple locations on the disk.



There are several ways to achieve high performance. One is to buy a huge multi-processor computer with enough RAM to hold the entire data model at once. Unfortunately, unless you are using PostgreSQL, your RDBMS vendor will probably give your bank account a reaming that it will not soon forget. The license fee will be four times as much for a four-CPU machine as for a one-CPU machine. Thus it might be best to try to get hold of the fastest possible single-CPU computer.

If you are processing a lot of transactions, all those CPUs bristling with RAM won't help you. Your bottleneck will be disk spindle contention. The solution to this is to chant "Oh what a friend I have in Seagate." Disks are slow. Very slow. Literally almost one million times slower than the computer. Therefore the computer spends a lot of time waiting for the disk(s). You can speed up SQL SELECTs simply by buying so much RAM that the entire database is in memory. However, the Durability requirement in the ACID test for transactions means that some record of a transaction will have to be written to a medium that won't be erased in the event of a power failure. If a disk can only do 100 seeks a second and you only have one disk, your RDBMS is going to be hard pressed to do more than about 100 updates a second.

The first thing you should do is mirror all of your disks. If you don't have the entire database in RAM, this speeds up SELECTs because the disk controller can read from whichever disk is closer to the desired track. The opposite effect can be achieved if you use "RAID level 5" where data is striped across multiple disks. Then the RDBMS has to wait for five disks to seek before it can cough up a few rows. Straight mirroring, or "RAID level 1", is what you want.

The next decision that you must make is "How many disks?" The [Oracle9i DBA Handbook](#) (Loney 2001; Oracle Press) recommends a 7x2 disk configuration as a *minimum compromise* for a machine doing nothing but database service. Their *solutions* start at 9x2 disks and go up to 22x2. The idea is to keep files that might be written in parallel on separate disks so that one can do 2200 seeks/second instead of 100.

Here's Kevin Loney's 17-disk (mirrored X2) solution for avoiding spindle contention:

Disk	Contents
1	Oracle software
2	SYSTEM tablespace
3	RBS tablespace (roll-back segment in case a transaction goes badly)
4	DATA tablespace
5	INDEXES tablespace (changing data requires changing indices; this allows those changes to proceed in parallel)
6	TEMP tablespace
7	TOOLS tablespace
8	Online Redo log 1, Control file 1 (these would be separated on a 22-disk machine)
9	Online Redo log 2, Control file 2
10	Online Redo log 3, Control file 3
11	Application Software
12	RBS_2
13	DATA_2 (tables that tend to be grabbed in parallel with those in DATA)
14	INDEXES_2
15	TEMP_USER
16	Archived redo log destination disk
17	Export dump file destination disk

Now that you have lots of disks, you finally have to be very thoughtful about how you lay your data out across them. "Enterprise" relational database management systems force you to think about where your data files should go. On a computer with one disk, this is merely annoying and keeps you from doing development; you'd probably get similar performance with a zero-administration RDBMS like PostgreSQL. But the flexibility is there in enterprise databases because you know which of your data areas tend to be accessed simultaneously and the computer doesn't. So if you do have a proper database server with a rack of disk drives, an intelligent manual layout can improve performance fivefold.

[Another area in which the commercial databases can be much faster than PostgreSQL is dealing with 50 simultaneous transactions from 50 different users. In a naive RDBMS implementation these would be lined up in order, a log of transaction 1 would be written to disk and then the database user who requested it would be notified of success, transaction 2 would then be processed and, after another disk write, be declared a success. Oracle, however, is smart enough to say "about 50 transactions came in at nearly the same moment so I'll write one big huge block of info to disk and then get back to all 50 users informing them of the success of their transaction." Thus performance in theory could 50 times better with Oracle (1 disk write) than with PostgreSQL (50 disk writes).]

Don't forget to back up

Be afraid. Be very afraid. Standard Unix or Windows file system backups will not leave you with a consistent and therefore



restoreable database on tape. Suppose that your RDBMS is storing your database in two separate filesystem files, foo.db and bar.db. Each of these files is 200 MB in size. You start your backup program running and it writes the file foo.db to tape. As the backup is proceeding, a transaction comes in that requires changes to foo.db and bar.db. The RDBMS makes those changes, but the ones to foo.db occur to a portion of the file that has already been written out to tape. Eventually the backup program gets around to writing bar.db to tape and it writes the new version with the change. Your system administrator arrives at 9:00 am and sends the tapes via courier to an off-site storage facility.



At noon, an ugly mob of users assembles outside your office, angered by your addition of a Flash animation splash page. You send one of your graphic designers out to explain how "cool" it looked when run off a local disk in a demo to the vice-president. The mob stones him to death and then burns your server farm to the ground. You manage to pry your way out of the rubble with one of those indestructible HP Unix box keyboards. You manage to get the HP disaster support people to let you use their machines for awhile and confidently load your backup tape. To your horror, the RDBMS chokes up blood following the restore. It turned out that there were linked data structures in foo.db and bar.db. Half of the data structures (the ones from foo.db) are the "old pre-transaction version" and half are the "new post-transaction version" (the ones from bar.db). One transaction occurring during your backup has resulted in a complete loss of availability for all of your data. Maybe you think that isn't the world's most robust RDBMS design but there is nothing in the SQL standard or manufacturer's documentation that says Oracle or Microsoft SQL Server can't work this way.

Full mirroring keeps you from going off-line due to media failure. But you still need snapshots of your database in case someone gets a little excited with a DELETE FROM statement or in the situation described above.

There are two ways to back up a relational database: off-line and on-line. For an off-line backup, you shut down the databases, thus preventing transactions from occurring. Most vendors would prefer that you use their utility to make a dump file of your off-line database, but in practice it will suffice just to back up the Unix or Windows filesystem files. Off-line backup is typically used by insurance companies and other big database users who only need to do transactions for eight hours a day.

Each RDBMS vendor has an advertised way of doing on-line backups. It can be as simple as "call this function and we'll grind away for a couple of hours building you a dump file that contains a consistent database but minus all the transactions that occurred after you called the function. Here's an Oracle example:

```
exp DBUSER/DBPASSWD file=/exportdest/foo.980210.dmp owner=DBUSER consistent=Y
```

This exports all the tables owned by DBUSER, pulling old rows from a rollback segment if a table has undergone transactions since the dump started.

What if your database is too large to be exported to a disk and can't be taken offline? Here's a technique that is used on the Oracle installation at Boston Children's Hospital:

- Break the mirror.
- Back up from the disks that are off-line as far as the database is concerned.
- Reestablish the mirror.

What if one of the on-line disks fails during backup? Are transactions lost? No. The redo log is

on a separate disk from the rest of the database. This increases performance in day-to-day operation and ensures that it is possible to recover transactions that occur when the mirror is broken, albeit with some off-line time.

The lessons here are several. First, whatever your backup procedure, make sure you test it with periodic restores. Second, remember that the backup and maintenance of an RDBMS is done by a full-time staffer at most companies, called "the dba", short for "database administrator". If the software worked as advertised, you could expect a few days of pain during the install and then periodic recurring pain to keep current with improved features. However, dba's earn their moderately lavish salaries. No amount of marketing hype suffices to make a C program work as advertised. That goes for an RDBMS just as much as for a word processor. Coming to terms with bugs can be a full-time job at a large installation. Most often this means finding workarounds since vendors are notoriously sluggish with fixes. Another full-time job is hunting down users who are doing queries that are taking 1000 times longer than necessary because they forgot to build indices or don't know SQL very well. Children's Hospital has three full-time dbas and they work hard.

Let's close by quoting Perrin Harkins. A participant in a discussion forum asked whether caching db queries in Unix files would speed up his Web server. Here's Perrin's response:

"Modern databases use buffering in RAM to speed up access to often requested data. You don't have to do anything special to make this happen, except tune your database well (which could take the rest of your life)."

Summary

Here's what you should have learned from reading this chapter:

- You and your programmers will make mistakes implementing transaction processing systems. You are better off focusing your energies on the application and leaving indexing, transactions, and concurrency to a database management system.
- The most practical database management software for Web sites is a relational database management system with a full-text indexer.
- If you can program a spreadsheet, you can program an RDBMS in SQL.
- RDBMSes are slow. Prepare to buy a big machine with a lot of disks.
- RDBMSes, though much more reliable than most user-written transaction processing code, are not nearly as reliable as a basic Web server pulling static files out a file system. Prepare to hire a half- or full-time database administrator if your database will grow to any significant size.

In the next chapter, we'll see how to best integrate an RDBMS with a Web server.

More

- [SQL for Web Nerds](#) is the author's short, sweet, free, and Web-available SQL tutorial
- [The Practical SQL Handbook](#) (Bowman, Emerson, Darnovsky 1996; Addison-Wesley) is my favorite introductory SQL tutorial. If you want to see how the language is used in real installations, get [SQL for Smarties: Advanced SQL Programming](#) (Joe Celko 1999; Morgan Kaufmann).
- For some interesting history about the first relational database implementation, visit http://www.mcjones.org/System_R/
- For a look under the hoods of a variety of database management systems, get [Readings in Database Systems](#) (Stonebraker and Hellerstein 1998; Morgan Kaufmann)
- [Building an Object-Oriented Database System](#) will help you take off your RDBMS

blindens as it describes the O_2 system.



[Get Hardcopy](#) or [move on to Chapter 13: Interfacing a Relational Database to the Web](#)

philg@mit.edu

Reader's Comments

It is unfortunate this page helps perpetuate myths and misinformation about databases.

For example:

It says there are RDBMSes in the market today. There is only one, Alphora Dataphor. All the others are SQL DBMSes, and SQL violates too many fundamentals of the relational model to be called a relational language. Therefore, SQL DBMSes aren't RDBMSes.

As a consequence, it says RDBMSes are slow. SQL DBMSes are slow, but RDBMSes needn't be, as the relational model defines that the physical model can be quite different than the logical model of the database. Thus, one can keep a clean logical model yet implement all kinds of optimisations in the physical level. That is what SQL optimisers try to do, but due to SQL complexity and confusion of the logical and physical model they fail.

-- [Leandro Guimarães Faria Corcete Dutra](#), May 18, 2004

I agree with the above poster in that the author is confusing SQL DBMS or a relational database. However I can forgive the author for this because this is so common in the industry (even if he is from MIT).

The slowness of most databases while inserting is because most database systems force the data to disk after each insert. This is a safety 'feature' not a performance 'bug' and has nothing to do with performance. Simply wrapping the entire load process in a transaction should have fixed the problem.

The Author's comments about PostgreSQL not being scalable is commercial systems are incorrect. PostgreSQL implemented write-ahead-logging (which is what he is talking about) with version 7.2 in 2002. Also, PostgreSQL has MVCC which makes it very, very scabale...some commerical implementations do not even have this, for example MS SQL server.

Merlin

-- [Merlin Moncure](#), March 18, 2005

Phil is somewhat incorrect about vendors supporting row-level locking support. MS has supported row-level locking since SQL Server 7.0. In practice, its transaction manager decides how granular a lock will be according to the parameters of the query. If too many individual rows are going to be locked in a transaction (according to some calculated "lock escalation threshold"), the entire table will be locked.

-- [pete lai](#), June 8, 2005

It is just terribly wrong to recommend MS Access database for any internet website, server-side or backoffice use. [1] Extract from: "INFO: Considerations for Server-Side Automation of Office" <http://support.microsoft.com/kb/257757/EN-US/> "Current licensing guidelines prevent Office Applications from being used on a server to service client requests, unless those clients themselves have licensed copies of Office. Using server-side Automation to provide Office functionality to unlicensed workstations is not covered by the End User License Agreement (EULA)." My 2 coins. Guennadi Vanine

-- [Guennadi Vanine](#), July 10, 2005

The reference to MySQL is very out of date. It seems that most people's understanding of MySQL's capabilities is mired in MySQL 3.x. Philip mentions that his information is current "circa 2003" but quite a lot has changed since then. For OLTP and enterprise grade applications, the stable now includes:

- MySQL 4.0, with InnoDB ACID-compliant storage engine
- [MySQL 5.0](#), with InnoDB and standards compliant stored procedures, triggers, views, etc.
- [MaxDB](#), formerly SAP DB, "a heavy-duty, SAP-certified open source database for OLTP and OLAP usage".

(Those who follow such things will note that Oracle now owns [Innobase Oy](#). However MySQL's license to sell InnoDB has been [renewed](#). They have several other OLTP options in the works, including a recent announcement of partnership with [Solid Information Technology](#).)

-- [Toby Thain](#), April 17, 2006

[Add a comment](#) | [Add a link](#)