Introduction to Database Management Systems

23 min. read · View original

After writing a preface lampooning academic eggheads who waste a lot of ink placing the relational database management system (RDBMS) in the context of 50 years of database management software, how does this book start? With a chapter placing the RDBMS in the context of other database management software.

Why? You ought to know why you're paying the huge performance, financial, and administration cost of an RDBMS. This chapter doesn't dwell on mainframe systems that people stopped using in the 1970s, but it does cover the alternative approaches to data management taken by Web sites that you've certainly visited and perhaps built.

The architect of any new information system must decide how much responsibility for data management the new custom software should take and how much should be left 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, the RDBMS, object-relational database management systems, and object databases. This chapter also introduces the SQL language.

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? You 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:

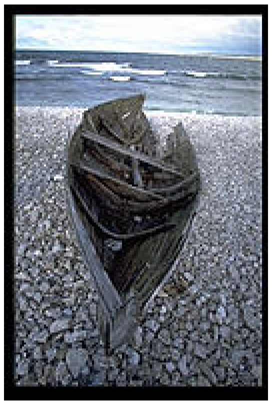
- Both inserts succeed.
- One of the inserts is lost.
- Information from the two inserts is mixed together so that both are corrupted.

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.

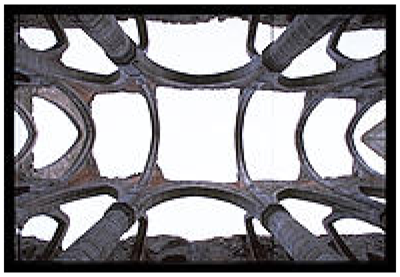
Yes, that's right, your stove. Suppose that you buy a $268,500 condo in Harvard Square. You think to yourself, "Now my friends will really be impressed with me" and invite them over for brunch. Not because you like them, but just to make them envious of your large lifestyle. Imagine your horror when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



A *Viking stove*?!? They cost $5000. The only way you are going to come up with this kind of cash is to join the growing ranks of on-line entrepreneurs. So you open an Internet bank. An experienced Perl script/flat-file wizard by now, you confidently build a system in which 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, 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 on-line 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 are. For a Web content management example, suppose that a user is editing a comment. A Web script tells the database to "copy the old comment value to an audit table and update the live table with the new text". If the hard drive fills up after the copy but before the update, the audit table insertion will be rolled back.

Consistency

Welcome to
UNIVERSAL CITY
For the comfort & security of all of our guests, we strictly enforce our Code of Conduct.

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 postings in a discussion forum table must be tied to a valid user ID. Then you hire Joe Novice to write some admin pages. Joe writes a delete-user page that doesn't bother to check whether or not the deletion will result in an orphaned discussion forum posting. The DBMS will check, though, and abort any transaction that would result in you having a discussion forum posting by a deleted user.

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 100 hits 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: (1) buy a 64-processor Sun E10000 server with 64 GB of RAM, or (2) 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 to also keep the index current.

This 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 building a cutting-edge Web service. You need the latest and greatest in computer technology. That's why you use, uh, Unix. Yeah. Anyway, even if your operating system was developed in 1969, 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.

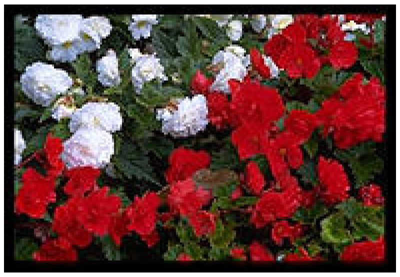
Yes, that's right, 1970. What you need to do is move your Web site into the '70s with one of these newfangled relational database management systems (RDBMS). Actually, as Codd notes in his paper, most of the problems we've encountered so far in this chapter were solved in the 1960s by off-the-shelf mainframe 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 1-800-USESUNX and ask them to send you a 16-processor Sun Enterprise 10000 with 32 GB of 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.

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 `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, 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 Visicalc or its derivatives such as Lotus 1-2-3 and Excel. 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 newsgroup `comp.databases` right now, I'll bet that you can find someone asking "How do I insert a string containing a single quote into an RDBMS?" Here's one harvested from AltaVista:

```
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 more sophisticated 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 (see the "Data Modeling" chapter for some ALTER TABLE commands that become possible starting with Oracle 8i), 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(email),
        number_type     varchar(15) check
(number_type in ('work','home','cell','beeper')),
        phone_number    varchar(20) not null
);
```

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 any 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.html before following any of my 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 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:

```
SQL> insert into phone_numbers values
('ogrady@fastbuck.com','work','(800) 555-1212');

ORA-02291: integrity constraint
(SCOTT.SYS_C001080) violated - parent key not
found
```

Ooops! 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','ogrady@fastbuck.com');


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 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, we 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. You probably wish you'd typed

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

but it is too late now.

There is one more fundamental SQL statement to learn. Suppose that Philip moves to Hollywood to realize his long-standing dream of becoming a major motion picture producer. Clearly a change of name is in order, though he'd be reluctant to give up the e-mail address he's had since 1976. Here's the SQL:

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

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, don't play around with UPDATE statements unless you have a WHERE clause at the end.

Brave New World

The original mid-1970s RDBMS let companies store the following kinds of data: numbers, dates, and character strings. After more than twenty years of innovation, you can today run out to the store and spend $300,000 on an "enterprise-class" RDBMS that will let you store the following kinds of data: numbers, dates, and character strings.

With an *object-relational* database, you get to define your own data types. For example, you could define a data type called `url`...

http://www.postgresql.org.

Braver New World

If you really want to be on the cutting edge, you can use a bona fide object database, like Object Design's ObjectStore (acquired by Progress Software). These 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 past in his Acura NSX. The relational database management system should have been crushed long ago under the weight of this superior technology, introduced with tremendous hype in the mid-1980s.

After 10 years, the market for object database management systems is about $100 million a year, perhaps 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 coughing up aggregations based on attributes.

The critical difference between RDBMS and ODBMS is the extent to which the programmer is constrained in interacting with the data. With an RDBMS the application program--written in a procedural language such as C, COBOL, Fortran, Perl, or Tcl--can have all kinds of catastrophic bugs. However, these bugs generally won't affect the information in the database because all communication with the RDBMS is constrained through SQL statements. With an ODBMS, the application program is directly writing slots in objects stored in the database. A bug in the application program may translate directly into corruption of the database, one of an organization's most valuable assets.

More

- "A Relational Model of Data for Large Shared Data Banks", E.F. Codd's paper in the June 1970 *Communications of the ACM* is reprinted in *Readings in Database Systems* (Stonebraker and Hellerstein 1998; Morgan Kaufmann). You might be wondering why, in 1999, eight years after the world's physicists gave us the Web, I didn't hyperlink you over to Codd's paper at www.acm.org. However, the organization is so passionately dedicated to demonstrating simultaneously the greed and incompetence of academic computer scientists worldwide that they charge money to electronically distribute material that they didn't pay for themselves.
- 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* (above)

Reference

- If you want to sit down and drive Oracle, you'll find SQL*Plus User's Guide and Reference useful.
- If you're hungry for detail, you can get God's honest truth (well, Larry Ellison's honest truth anyway, which is pretty much the same thing in the corporate IT world) from Oracle8 Server Concepts.

Next: data modeling

---

philg@mit.edu

Reader's Comments

> Actually, the ACM do make A Relational Model of Data for Large Shared Data Banks freely available, but that's the exception rather than the rule.
>
> -- Tom L, November 26, 2003

> I'm using MySQL, and I wanted to comment on a snag I ran into while I was following the tutorial in this page. Maybe other newbies can benefit from this.
>
> As far as I can tell:
>
> a) MySQL supports different "storage engines" for tables. This is presumably a good thing. However, not all engines support referencial constraints.
>
> b) For a MySQL table to support a "references" constraint, it must be of type InnoDB. In my

installation (on SuSE Linux, right out of a standard RPM binary package), this is *not* the default. So you have to either change the server configuration to make this the default, or specify "ENGINE = InnoDB" after the closing parenthesis in the table definition.

c) Even for InnoDB, the syntax described by Phil above does not work, though it is not rejected, merely ignored. According to the manual, this is effectively just a comment to the developer that this column is supposed to reference another column, even if the constraint is not enforced by mysql.

d) So, the only way to make this kind of constraint work is to: 1. make the table InnoDB and 2. use the "FOREIGN KEY (email) REFERENCES mailing_list(email)" format as a separate entry inside the table definition.

[MySQL won't even give a warning! Not even a reminder that such reference clauses are merely "comments". It will just happily ignore them and allow any old value in that row. Ugh.]

-- Antonio Ramirez, March 19, 2007

Another addendum for MySQL is that "The CHECK clause is parsed but ignored by all storage engines" (http://dev.mysql.com/doc/refman/5.1/en/create-table.html). The CHECK can be accomplished, however, with an appropriate TRIGGER.

-- Eddie Marks, June 22, 2010

I'd avoid MySQL when learning about RDBMSes - it's philosophically a bit different, as evidenced by the silent errors and nondefault status of InnoDB. I use PostGres instead - it's free, and the command line tools are excellent. It's straightforward to configure and install, and has mature support pretty much anywhere you care to use it.

-- chris cooney, September 2, 2010

The default storage engine can be changed in the MySQL config file, which on Linux (e.g. RHEL, CentOS, Debian, SLES, etc.) is stored at **/etc/my.cnf**

Within that file there SHOULD be a section labeled [mysqld]. Add the following line immediately below that label so that the result is as follows:

```
[mysqld]
default-storage-engine = myisam
```

You will then need to restart the mysqld process. There are different ways to accomplish this task depending upon your version of Linux. One way is as follow:

```
/etc/init.d/mysql stop
/etc/init.d/mysql start
```

Enjoy!

-- Cai Black, June 9, 2011

Add a comment