SQL for Web Nerds

by Philip Greenspun

This is a draft manuscript. Comments (emailed to philg@mit.edu) would be welcome.

More:
the complete Oracle doc set is available online for registered (no charge) users, as a link from

Preface

to SQL for Web Nerds by Philip Greenspun

The last thing the world needs is another SQL tutorial. So here's ours ...

We could wallpaper all 500 rooms of the computer science building at MIT with different SQL tutorials from various publishers. Yet when it came time to sit down with our students and teach them this material, we couldn't bear to use any of the commercial texts.

The CS department at MIT doesn't offer a course in SQL (or in any other computer language *per se* for that matter). But lots of universities do and therefore you can choose from many textbooks designed to teach SQL to 20-year-old CS majors. Perfect, eh? Not for us. Part of the problem is that universities try to give students what they deem to be eternal knowledge. So rather than focus on the relational model and SQL, the overwhelmingly dominant system for the past 20 years, introductory database textbooks spend several chapters talking about database management systems that were used in the 1960s.

The second part of the problem with college-level texts is that these books are too dreary and long. In our class, the students face the tangible problem of building a sophisticated db-backed Web service in 12 weeks. This motivates them to learn whatever intricacies of SQL that they need. But what keeps a student going through 500 pages of SQL and RDBMS? We can't figure this out.

The third problem is that most of the college-level textbooks bring to mind the old question "What is the difference between a tenured professor of computer science and an ape?" (The ape doesn't think he can program.) Sure, the academic egghead author can learn the syntax of SQL but he or she won't have any personal experience with real-life interesting systems. That's because no real user of the RDBMS is stupid enough to hire a professor to write any SQL code.

In our computer-obsessed society, we need not be stuck with the dry theoretical offerings of computer scientists. Walk into any bookstore and you'll find SQL tutorials. Sadly, due to structural problems in the trade computer book industry (see http://photo.net/wtr/dead-trees/story.html), most of these books are written by authors who picked up SQL as they were writing. There are some good ones, however, our favorite being *The Practical SQL Handbook* (Bowman, Emerson, Darnovsky; Addison-Wesley). We truly do like this book so we can feel free to pick on it:

- though it is 450 pages long, by straining to offer complete coverage of simple boring stuff the authors run out of space and energy to cover the interesting and sophisticated stuff

- students have to learn SQL in the context of a data model; like its competitors *The Practical SQL Handbook* forces readers to live in the dessicated world of business data processing for several hundred pages. This is where the RDBMS started in the 1970s and if you want to get a job in a corporate IS department, this is not a bad place to live while learning SQL. However, we're trying to demonstrate how the concurrency control and transaction management capabilities of the RDBMS enable the construction of powerful reliable Web services. We can dazzle the students with much more interesting and relevant data models than *Practical SQL's* "bookbiz database about a fictitious publishing company".
- the authors write with the assumption that the reader is unaccustomed to thinking formally and using formal languages
- the authors avoid the ugly fact that SQL is not a standard. Most SQL queries involve dates and times. Yet there are only two pages out of 450 involving the date data type. Why? The authors don't explain but perhaps it is because they didn't want to say "here's what ANSI SQL date-time arithmetic looks like and, by the way, Oracle has completely different syntax and semantics."
- the authors assume that the Web doesn't exist, i.e., that the physical book must be self-contained and comprehensive. Anyone actually using an RDBMS is going to have full documentation on the Web or at least (shudder) on CD-ROM. This book started on the Web and therefore we assume that we can cover the interesting and pedagogically valuable stuff then link students to the full documentation.

Bashing other authors and publishers is fun but isn't pedagogically effective. Thus it is probably worth stating what this SQL tutorial tries to do.

First and foremost, we keep our readers in the world of Web services. Most often they are working within the data model for the ArsDigita Community System, an open-source toolkit for building collaborative Web sites. Sometimes we drag readers into the dreary world of commerce but at least it is the flashier-than-average corner of *ecommerce*.

Second, our examples are all drawn from real production Web sites that get close to 1 million requests per day. This should make the examples more interesting, particular as the sites are mostly still up and running so the students will be able to visit pages and see the queries in action on up-to-the-minute data sets.

Third, we assume that our readers are bright and accustomed to formal languages. We don't assume any experience with declarative languages, database query languages, or any specific programming language. But once we can assume that the reader has written code, it is possible to use more sophisticated examples and get to the interesting stuff more quickly. So if this book ends up being a bad choice for the office manager who wants to start building marketing reports, we hope that we make up for it by making it a great choice for the MIT student or the working programmer.

Fourth, we assume that our readers will be using Oracle. This is a safe assumption for our class because we set up the computing facility and, in fact, Oracle8 is the only RDBMS running on it! It is also a safe assumption for much of the world: Oracle is the most popular RDBMS system available. We find it burdensome to maintain. We wish it were open-source. We wish it were free. Yet if we

accept Oracle as part of the landscape, we don't have to waste a lot of ink pretending that SQL is a standard.

Introduction

by Philip Greenspun, part of SQL for Web Nerds

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.

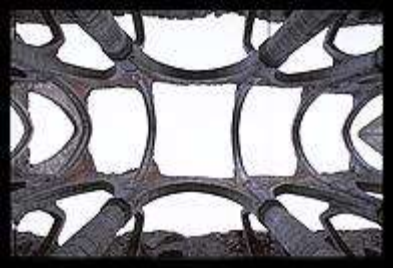
So what? Emacs may be ancient but it is still the best text editor in the world. You love using it so you might as well spend your weekends and evenings manually fixing up your flat file databases with Emacs. Who needs concurrency control?

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.

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? 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 (http://www.odi.com). 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.

Data Modeling

part of SQL for Web Nerds by Philip Greenspun

Data modeling is the hardest and most important activity in the RDBMS world. If you get the data model wrong, your application might not do what users need, it might be unreliable, it might fill up the database with garbage. Why then do we start a SQL tutorial with the most challenging part of the job? Because you can't do queries, inserts, and updates until you've defined some tables. And defining tables is *data modeling*.

When data modeling, you are telling the RDBMS the following:

- what elements of the data you will store
- how large each element can be
- what kind of information each element can contain
- what elements may be left blank
- which elements are constrained to a fixed range
- whether and how various tables are to be linked

Three-Valued Logic

Programmers in most computer languages are familiar with Boolean logic. A variable may be either true or false. Pervading SQL, however, is the alien idea of *three-valued logic*. A column can be true, false, or NULL. When building the data model you must affirmatively decide whether a NULL value will be permitted for a column and, if so, what it means.

For example, consider a table for recording user-submitted comments to a Web site. The publisher has made the following stipulations:

- comments won't go live until approved by an editor
- the admin pages will present editors with all comments that are pending approval, i.e., have been submitted but neither approved nor disapproved by an editor already

Here's the data model:

```
create table user_submitted_comments (
        comment_id              integer primary key,
        user_id                 not null references users,
        submission_time         date default sysdate not null,
        ip_address              varchar(50) not null,
        content                 clob,
        approved_p              char(1) check(approved_p in ('t','f'))
);
```

Implicit in this model is the assumption that `approved_p` can be NULL and that, if not explicitly set during the INSERT, that is what it will default to. What about the check constraint? It would seem to restrict `approved_p` to values of "t" or "f". NULL, however, is a special value and if we wanted to prevent `approved_p` from taking on NULL we'd have to add an explicit `not null` constraint.

How do NULLs work with queries? Let's fill `user_submitted_comments` with some sample data and see:

```
insert into user_submitted_comments
(comment_id, user_id, ip_address, content)
```

```
values
(1, 23069, '18.30.2.68', 'This article reminds me of Hemingway');

Table created.

SQL> select first_names, last_name, content, user_submitted_comments.approved_p
from user_submitted_comments, users
where user_submitted_comments.user_id = users.user_id;

FIRST_NAMES  LAST_NAME      CONTENT                                APPROVED_P
------------ -------------- ------------------------------------ ------------
Philip       Greenspun      This article reminds me of Hemingway
```

We've successfully JOINed the `user_submitted_comments` and `users` table to get both the comment content and the name of the user who submitted it. Notice that in the select list we had to explicitly request **`user_submitted_comments`**`.approved_p`. This is because the `users` table also has an `approved_p` column.

When we inserted the comment row we did not specify a value for the `approved_p` column. Thus we expect that the value would be NULL and in fact that's what it seems to be. Oracle's SQL*Plus application indicates a NULL value with white space.

For the administration page, we'll want to show *only* those comments where the `approved_p` column is NULL:

```
SQL> select first_names, last_name, content, user_submitted_comments.approved_p
from user_submitted_comments, users
where user_submitted_comments.user_id = users.user_id
and user_submitted_comments.approved_p = NULL;

no rows selected
```

"No rows selected"? That's odd. We know for a fact that we have one row in the comments table and that is `approved_p` column is set to NULL. How to debug the query? The first thing to do is simplify by removing the JOIN:

```
SQL> select * from user_submitted_comments where approved_p = NULL;

no rows selected
```

What is happening here is that any expression involving NULL evaluates to NULL, including one that effectively looks like "NULL = NULL". The WHERE clause is looking for expressions that evaluate to true. What you need to use is the special test IS NULL:

```
SQL> select * from user_submitted_comments where approved_p is NULL;

COMMENT_ID    USER_ID SUBMISSION_T IP_ADDRESS
---------- ---------- ------------ ----------
CONTENT                            APPROVED_P
---------------------------------- ------------
         1      23069 2000-05-27   18.30.2.68
This article reminds me of Hemingway
```

An adage among SQL programmers is that the only time you can use "= NULL" is in an UPDATE statement (to set a column's value to NULL). It never makes sense to use "= NULL" in a WHERE clause.

The bottom line is that as a data modeler you will have to decide which columns can be NULL and what that value will mean.

Back to the Mailing List

Let's return to the mailing list data model from the introduction:

```
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','beeper')),
        phone_number    varchar(20) not null
);
```

This data model locks you into some realities:

- You will not be sending out any physical New Year's cards to folks on your mailing list; you don't have any way to store their addresses.
- You will not be sending out any electronic mail to folks who work at companies with elaborate Lotus Notes configurations; sometimes Lotus Notes results in email addresses that are longer than 100 characters.
- You are running the risk of filling the database with garbage since you have not constrained phone numbers in any way. American users could add or delete digits by mistake. International users could mistype country codes.
- You are running the risk of not being able to serve rich people because the `number_type` column may be too constrained. Suppose William H. Gates the Third wishes to record some extra phone numbers with types of "boat", "ranch", "island", and "private_jet". The `check (number_type in ('work','home','cell','beeper'))` statement prevents Mr. Gates from doing this.
- You run the risk of having records in the database for people whose name you don't know, since the `name` column of `mailing_list` is free to be NULL.
- Changing a user's email address won't be the simplest possible operation. You're using `email` as a key in two tables and therefore will have to update both tables. The `references mailing_list` keeps you from making the mistake of only updating `mailing_list` and leaving orphaned rows in `phone_numbers`. But if users changed their email addresses frequently, you might not want to do things this way.
- Since you've no provision for storing a password or any other means of authentication, if you allow users to update their information, you run a minor risk of allowing a malicious change.

(The risk isn't as great as it seems because you probably won't be publishing the complete mailing list; an attacker would have to guess the names of people on your mailing list.)

These aren't necessarily bad realities in which to be locked. However, a good data modeler recognizes that every line of code in the .sql file has profound implications for the Web service.

Papering Over Your Mistakes with Triggers

Suppose that you've been using the above data model to collect the names of Web site readers who'd like to be alerted when you add new articles. You haven't sent any notices for two months. You want to send everyone who signed up in the last two months a "Welcome to my Web service; thanks for signing up; here's what's new" message. You want to send the older subscribers a simple "here's what's new" message. But you can't do this because you didn't store a registration date. It is easy enough to fix the table:

```
alter table mailing_list add (registration_date date);
```

But what if you have 15 different Web scripts that use this table? The ones that query it aren't a problem. If they don't ask for the new column, they won't get it and won't realize that the table has been changed (this is one of the big selling features of the RDBMS). But the scripts that update the table will all need to be changed. If you miss a script, you're potentially stuck with a table where various random rows are missing critical information.

Oracle has a solution to your problem: *triggers*. A trigger is a way of telling Oracle "any time anyone touches this table, I want you to execute the following little fragment of code". Here's how we define the trigger `mailing_list_registration_date`:

```
create trigger mailing_list_registration_date
before insert on mailing_list
for each row
when (new.registration_date is null)
begin
 :new.registration_date := sysdate;
end;
```

Note that the trigger only runs when someone is trying to insert a row with a NULL registration date. If for some reason you need to copy over records from another database and they have a registration date, you don't want this trigger overwriting it with the date of the copy.

A second point to note about this trigger is that it runs `for each row`. This is called a "row-level trigger" rather than a "statement-level trigger", which runs once per transaction, and is usually not what you want.

A third point is that we're using the magic Oracle procedure `sysdate`, which will return the current time. The Oracle `date` type is precise to the second even though the default is to display only the day.

A fourth point is that, starting with Oracle 8, we could have done this more cleanly by adding a `default sysdate` instruction to the column's definition.

The final point worth noting is the `:new.` syntax. This lets you refer to the new values being inserted. There is an analogous :old. feature, which is useful for update triggers:

```
create or replace trigger mailing_list_update
before update on mailing_list
for each row
when (new.name <> old.name)
begin
  -- user is changing his or her name
  -- record the fact in an audit table
  insert into mailing_list_name_changes
    (old_name, new_name)
  values
    (:old.name, :new.name);
end;
/
show errors
```

This time we used the `create or replace` syntax. This keeps us from having to `drop trigger mailing_list_update` if we want to change the trigger definition. We added a comment using the SQL comment shortcut "--". The syntax `new.` and `old.` is used in the trigger definition, limiting the conditions under which the trigger runs. Between the `begin` and `end`, we're in a PL/SQL block. This is Oracle's procedural language, described later, in which `new.name` would mean "the `name` element from the record in `new`". So you have to use `:new` instead. It is obscurities like this that lead to competent Oracle consultants being paid $200+ per hour.

The "/" and `show errors` at the end are instructions to Oracle's SQL*Plus program. The slash says "I'm done typing this piece of PL/SQL, please evaluate what I've typed." The "show errors" says "if you found anything to object to in what I just typed, please tell me".

The Discussion Forum -- philg's personal odyssey

Back in 1995, I built a threaded discussion forum, described *ad nauseum* in http://photo.net/wtr/dead-trees/53013.htm. Here's how I stored the postings:

```
create table bboard (
        msg_id          char(6) not null primary key,
        refers_to       char(6),
        email           varchar(200),
        name            varchar(200),
        one_line        varchar(700),
        message         clob,
        notify          char(1) default 'f' check (notify in ('t','f')),
        posting_time    date,
        sort_key        varchar(600)
);
```

German order reigns inside the system itself: messages are uniquely keyed with `msg_id`, refer to each other (i.e., say "I'm a response to msg X") with `refers_to`, and a thread can be displayed conveniently by using the `sort_key` column.

Italian chaos is permitted in the `email` and `name` columns; users could remain anonymous, masquerade as "president@whitehouse.gov" or give any name.

This seemed like a good idea when I built the system. I was concerned that it work reliably. I didn't care whether or not users put in bogus content; the admin pages made it really easy to remove such postings and, in any case, if someone had something interesting to say but needed to remain anonymous, why should the system reject their posting?

One hundred thousand postings later, as the moderator of the photo.net Q&A forum, I began to see the dimensions of my data modeling mistakes.

First, anonymous postings and fake email addresses didn't come from Microsoft employees revealing the dark truth about their evil bosses. They came from complete losers trying and failing to be funny or wishing to humiliate other readers. Some fake addresses came from people scared by the rising tide of spam email (not a serious problem back in 1995).

Second, I didn't realize how the combination of my email alert systems, fake email addresses, and Unix mailers would result in my personal mailbox filling up with messages that couldn't be delivered to "asdf@asdf.com" or "duh@duh.net".

Although the solution involved changing some Web scripts, fundamentally the fix was add a column to store the IP address from which a post was made:

```
alter table bboard add (originating_ipvarchar(16));
```

Keeping these data enabled me to see that most of the anonymous posters were people who'd been using the forum for some time, typically from the same IP address. I just sent them mail and asked them to stop, explaining the problem with bounced email.

After four years of operating the photo.net community, it became apparent that we needed ways to

- display site history for users who had changed their email addresses
- discourage problem users from burdening the moderators and the community
- carefully tie together user-contributed content in the various subsystems of photo.net

The solution was obvious to any experienced database nerd: a canonical users table and then content tables that reference it. Here's a simplified version of the data model, taken from the ArsDigita Community System:

```
create table users (
        user_id                 integer not null primary key,
        first_names             varchar(100) not null,
        last_name               varchar(100) not null,
        email                   varchar(100) not null unique,
        ..
);

create table bboard (
        msg_id          char(6) not null primary key,
        refers_to       char(6),
```

```
        topic           varchar(100) not null references bboard_topics,
        category        varchar(200),  -- only used for categorized Q&A forums
        originating_ip  varchar(16),   -- stored as string, separated by periods
        user_id         integer not null references users,
        one_line        varchar(700),
        message         clob,
        -- html_p - is the message in html or not
        html_p          char(1) default 'f' check (html_p in ('t','f')),
        ...
);

create table classified_ads (
        classified_ad_id       integer not null primary key,
        user_id                integer not null references users,
        ...
);
```

Note that a contributor's name and email address no longer appear in the `bboard` table. That doesn't mean we don't know who posted a message. In fact, this data model can't even represent an anonymous posting: `user_id integer not null references users` requires that each posting be associated with a user ID and that there actually be a row in the `users` table with that ID.

First, let's talk about how much fun it is to move a live-on-the-Web 600,000 hit/day service from one data model to another. In this case, note that the original `bboard` data model had a single `name` column. The community system has separate columns for first and last names. A conversion script can easily split up "Joe Smith" but what is it to do with William Henry Gates III?

How do we copy over anonymous postings? Remember that Oracle is not flexible or intelligent. We said that we wanted every row in the `bboard` table to reference a row in the `users` table. Oracle will abort any transaction that would result in a violation of this integrity constraint. So we either have to drop all those anonymous postings (and any non-anonymous postings that refer to them) or we have to create a user called "Anonymous" and assign all the anonymous postings to that person. The technical term for this kind of solution is *kludge*.

A more difficult problem than anonymous postings is presented by long-time users who have difficulty typing and or keeping a job. Consider a user who has identified himself as

1. Joe Smith; jsmith@ibm.com
2. Jo Smith; jsmith@ibm.com (typo in name)
3. Joseph Smith; jsmth@ibm.com (typo in email)
4. Joe Smith; cantuseworkaddr@hotmail.com (new IBM policy)
5. Joe Smith-Jones; joe_smithjones@hp.com (got married, changed name, changed jobs)
6. Joe Smith-Jones; jsmith@somedivision.hp.com (valid but not canonical corporate email address)
7. Josephina Smith; jsmith@somedivision.hp.com (sex change; divorce)
8. Josephina Smith; josephina_smith@hp.com (new corporate address)
9. Siddhartha Bodhisattva; josephina_smith@hp.com (change of philosophy)
10. Siddhartha Bodhisattva; thinkwaitfast@hotmail.com (traveling for awhile to find enlightenment)

Contemporary community members all recognize these postings as coming from the same person but it would be very challenging even to build a good semi-automated means of merging postings from this person into one user record.

Once we've copied everything into this new *normalized* data model, notice that we can't dig ourselves into the same hole again. If a user has contributed 1000 postings, we don't have 1000 different records of that person's name and email address. If a user changes jobs, we need only update one column in one row in one table.

The `html_p` column in the new data model is worth mentioning. In 1995, I didn't understand the problems of user-submitted data. Some users will submit plain text, which seems simple, but in fact you can't just spit this out as HTML. If user A typed < or > characters, they might get swallowed by user B's Web browser. Does this matter? Consider that "<g>" is interpreted in various online circles as an abbreviation for "grin" but by Netscape Navigator as an unrecognized (and therefore ignore) HTML tag. Compare the meaning of

"We shouldn't think it unfair that Bill Gates has more wealth than the 100 million poorest Americans *combined*. After all, he invented the personal computer, the graphical user interface, and the Internet."
with
"We shouldn't think it unfair that Bill Gates has more wealth than the 100 million poorest Americans *combined*. After all, he invented the personal computer, the graphical user interface, and the Internet. <g>"

It would have been easy enough for me to make sure that such characters never got interpreted as markup. In fact, with AOLserver one can do it with a single call to the built-in procedure `ns_quotehtml`. However, consider the case where a nerd posts some HTML. Other users would then see

"For more examples of my brilliant thinking and modesty, check out my home page."
I discovered that the only real solution is to ask the user whether the submission is an HTML fragment or plain text, show the user an approval page where the content may be previewed, and then remember what the user told us in an `html_p` column in the database.

Is this data model perfect? Permanent? Absolutely. It will last for at least... Whoa! Wait a minute. I didn't know that Dave Clark was replacing his original Internet Protocol, which the world has been running since around 1980, with IPv6 (http://info.internet.isi.edu/in-notes/rfc/files/rfc2460.txt). In the near future, we'll have IP addresses that are 128 bits long. That's 16 bytes, each of which takes two hex characters to represent. So we need 32 characters plus at least 7 more for periods that separate the hex digits. We might also need a couple of characters in front to say "this is a hex representation". Thus our brand new data model in fact has a crippling deficiency. How easy is it to fix? In Oracle:

```
alter table bboard modify (originating_ip varchar(50));
```

You won't always get off this easy. Oracle won't let you shrink a column from a maximum of 50 characters to 16, even if no row has a value longer than 16 characters. Oracle also makes it tough to add a column that is constrained `not null`.

Representing Web Site Core Content

Free-for-all Internet discussions can often be useful and occasionally are compelling, but the anchor of a good Web site is usually a set of carefully authored extended documents. Historically these have tended to be stored in the Unix file system and they don't change too often. Hence I refer to them as *static pages*. Examples of static pages on the photo.net server include this book chapter, the tutorial on light for photographers at http://photo.net/photo/tutorial/light.html.

We have some big goals to consider. We want the data in the database to

- help community experts figure out which articles need revision and which new articles would be most valued by the community at large.
- help contributors work together on a draft article or a new version of an old article.
- collect and organize reader comments and discussion, both for presentation to other readers but also to assist authors in keeping content up-to-date.
- collect and organize reader-submitted suggestions of related content out on the wider Internet (i.e., links).
- help point readers to new or new-to-them content that might interest them, based on what they've read before or based on what kind of content they've said is interesting.

The big goals lead to some more concrete objectives:

- We will need a table that holds the static pages themselves.
- Since there are potentially many comments per page, we need a separate table to hold the user-submitted comments.
- Since there are potentially many related links per page, we need a separate table to hold the user-submitted links.
- Since there are potentially many authors for one page, we need a separate table to register the author-page many-to-one relation.
- Considering the "help point readers to stuff that will interest them" objective, it seems that we need to store the category or categories under which a page falls. Since there are potentially many categories for one page, we need a separate table to hold the mapping between pages and categories.

```
create table static_pages (
        page_id         integer not null primary key,
        url_stub        varchar(400) not null unique,
        original_author integer references users(user_id),
        page_title      varchar(4000),
        page_body       clob,
        obsolete_p      char(1) default 'f' check (obsolete_p in ('t','f')),
        members_only_p  char(1) default 'f' check (members_only_p in ('t','f')),
        price           number,
```

```
        copyright_info varchar(4000),
        accept_comments_p      char(1) default 't' check (accept_comments_p in
('t','f')),
        accept_links_p         char(1) default 't' check (accept_links_p in
('t','f')),
        last_updated           date,
        -- used to prevent minor changes from looking like new content
        publish_date           date
);

create table static_page_authors (
        page_id         integer not null references static_pages,
        user_id         integer not null references users,
        notify_p        char(1) default 't' check (notify_p in ('t','f')),
        unique(page_id,user_id)
);
```

Note that we use a generated integer `page_id` key for this table. We could key the table by the `url_stub` (filename), but that would make it very difficult to reorganize files in the Unix file system (something that should actually happen very seldom on a Web server; it breaks links from foreign sites).

How to generate these unique integer keys when you have to insert a new row into `static_pages`? You could

- lock the table
- find the maximum `page_id` so far
- add one to create a new unique `page_id`
- insert the row
- commit the transaction (releases the table lock)

Much better is to use Oracle's built-in sequence generation facility:

```
create sequence page_id_sequence start with 1;
```

Then we can get new page IDs by using `page_id_sequence.nextval` in INSERT statements (see the Transactions chapter for a fuller discussion of sequences).

Reference

Here is a summary of the data modeling tools available to you in Oracle, each hyperlinked to the Oracle documentation. This reference section covers the following:

- data types
- statements for creating, altering, and dropping tables
- constraints

Data Types

For each column that you define for a table, you must specify the data type of that column. Here are your options:

Character Data

char(n) A fixed-length character string, e.g., `char(200)` will take up 200 bytes regardless of how long the string actually is. This works well when the data truly are of fixed size, e.g., when you are recording a user's sex as "m" or "f". This works badly when the data are of variable length. Not only does it waste space on the disk and in the memory cache, but it makes comparisons fail. For example, suppose you insert "rating" into a `comment_type` column of type `char(30)` and then your Tcl program queries the database. Oracle sends this column value back to procedural language clients padded with enough spaces to make up 30 total characters. Thus if you have a comparison within Tcl of whether `$comment_type == "rating"`, the comparison will fail because `$comment_type` is actually "rating" followed by 24 spaces.

The maximum length char in Oracle8 is 2000 bytes.

varchar(n) A variable-length character string, up to 4000 bytes long in Oracle8. These are stored in such a way as to minimize disk space usage, i.e., if you only put one character into a column of type `varchar(4000)`, Oracle only consumes two bytes on disk. The reason that you don't just make all the columns `varchar(4000)` is that the Oracle indexing system is limited to indexing keys of about 700 bytes.

clob A variable-length character string, up to 4 gigabytes long in Oracle8. The CLOB data type is useful for accepting user input from such applications as discussion forums. Sadly, Oracle8 has tremendous limitations on how CLOB data may be inserted, modified, and queried. Use `varchar(4000)` if you can and prepare to suffer if you can't.

In a spectacular demonstration of what happens when companies don't follow the lessons of *The Mythical Man Month*, the regular string functions don't work on CLOBs. You need to call identically named functions in the DBMS_LOB package. These functions take the same arguments but in different orders. You'll never be able to write a working line of code without first reading the DBMS_LOB section of the *Oracle8 Server Application Developer's Guide*.

nchar, nvarchar, nclob The n prefix stands for "national character set". These work like char, varchar, and clob but for multi-byte characters (e.g., Unicode; see http://www.unicode.org).

Numeric Data

number Oracle actually only has one internal data type that is used for storing numbers. It can handle 38 digits of precision and exponents from -130 to +126. If you want to get fancy, you can specify precision and scale limits. For example, `number(3,0)` says "round everything to an integer [scale 0] and accept numbers than range from -999 to +999". If you're American and commercially minded, `number(9,2)` will probably work well for storing prices in dollars and cents (unless you're selling stuff to Bill Gates, in which case the billion dollar limit imposed by the precision of 9 might prove constraining). If you

	are Bill Gates, you might not want to get distracted by insignificant numbers: Tell Oracle to round everything to the nearest million with `number(38,-6)`.
integer	In terms of storage consumed and behavior, this is not any different from `number(38)` but I think it reads better and it is more in line with ANSI SQL (which would be a standard if anyone actually implemented it).
	Dates and Date/Time Intervals (Version 9i and newer)
timestamp	A point in time, recorded with sub-second precision. When creating a column you specify the number of digits of precision beyond one second from 0 (single second precision) to 9 (nanosecond precision). Oracle's calendar can handle dates between between January 1, 4712 BC and December 31, 9999 AD. You can put in values with the `to_timestamp` function and query them out using the `to_char` function. Oracle offers several variants of this datatype for coping with data aggregated across multiple timezones.
interval year to month	An amount of time, expressed in years and months.
interval day to second	An amount of time, expressed in days, hours, minutes, and seconds. Can be precise down to the nanosecond if desired.
	Dates and Date/Time Intervals (Versions 8i and earlier)
date	Obsolete as of version 9i. A point in time, recorded with one-second precision, between January 1, 4712 BC and December 31, 4712 AD. You can put in values with the `to_date` function and query them out using the `to_char` function. If you don't use these functions, you're limited to specifying the date with the default system format mask, usually 'DD-MON-YY'. This is a good recipe for a Year 2000 bug since January 23, 2000 would be '23-JAN-00'. On better-maintained systems, this is often the ANSI default: 'YYYY-MM-DD', e.g., '2000-01-23' for January 23, 2000.
number	Hey, isn't this a typo? What's `number` doing in the date section? It is here because this is how Oracle versions prior to 9i represented date-time intervals, though their docs never say this explicitly. If you add numbers to dates, you get new dates. For example, tomorrow at exactly this time is `sysdate+1`. To query for stuff submitted in the last hour, you limit to `submitted_date > sysdate - 1/24`.
	Binary Data
blob	BLOB stands for "Binary Large OBject". It doesn't really have to be all that large, though Oracle will let you store up to 4 GB. The BLOB data type was set up to permit the storage of images, sound recordings, and other inherently binary data. In practice, it also gets used by fraudulent application software vendors. They spend a few years kludging together some nasty format of their own. Their MBA executive customers demand that the whole thing be RDBMS-based. The software vendor learns enough about Oracle to "stuff everything into a BLOB". Then all the marketing and sales folks are happy because the application is now running from Oracle instead of from the file system. Sadly, the programmers and users don't get much because you can't use SQL very effectively to query or update what's inside a BLOB.

bfile A binary file, stored by the operating system (typically Unix) and kept track of by Oracle. These would be useful when you need to get to information both from SQL (which is kept purposefully ignorant about what goes on in the wider world) and from an application that can only read from standard files (e.g., a typical Web server). The bfile data type is pretty new but to my mind it is already obsolete: Oracle 8.1 (8i) lets external applications view content in the database as though it were a file on a Windows NT server. So why not keep everything as a BLOB and enable Oracle's Internet File System?

Despite this plethora of data types, Oracle has some glaring holes that torture developers. For example, there is no Boolean data type. A developer who needs an `approved_p` column is forced to use `char(1) check(this_column in ('t','f'))` and then, instead of the clean query `where approved_p` is forced into `where approved_p = 't'`.

Oracle8 includes a limited ability to create your own data types. Covering these is beyond the scope of this book. See Oracle8 Server Concepts, User-Defined Datatypes.

Tables

The basics:

```
CREATE TABLE your_table_name (
       the_key_column       key_data_type PRIMARY KEY,
       a_regular_column             a_data_type,
       an_important_column   a_data_type NOT NULL,
       ... up to 996 intervening columns in Oracle8 ...
       the_last_column       a_data_type
);
```

Even in a simple example such as the one above, there are few items worth noting. First, I like to define the key column(s) at the very top. Second, the `primary key` constraint has some powerful effects. It forces `the_key_column` to be non-null. It causes the creation of an index on `the_key_column`, which will slow down updates to `your_table_name` but improve the speed of access when someone queries for a row with a particular value of `the_key_column`. Oracle checks this index when inserting any new row and aborts the transaction if there is already a row with the same value for `the_key_column`. Third, note that there is no comma following the definition of the last row. If you are careless and leave the comma in, Oracle will give you a very confusing error message.

If you didn't get it right the first time, you'll probably want to

```
alter table your_table_name add (new_column_name a_data_type any_constraints);
```

or

```
alter table your_table_name modify (existing_column_name new_data_type
new_constraints);
```

In Oracle 8i you can drop a column:

```
alter table your_table_name drop column existing_column_name;
```

(see http://www.oradoc.com/keyword/drop_column).

If you're still in the prototype stage, you'll probably find it easier to simply

```
drop table your_table_name;
```

and recreate it. At any time, you can see what you've got defined in the database by querying Oracle's *Data Dictionary*:

```
SQL> select table_name from user_tables order by table_name;

TABLE_NAME
------------------------------
ADVS
ADV_CATEGORIES
ADV_GROUPS
ADV_GROUP_MAP
ADV_LOG
ADV_USER_MAP
AD_AUTHORIZED_MAINTAINERS
AD_CATEGORIES
AD_DOMAINS
AD_INTEGRITY_CHECKS
BBOARD
...
STATIC_CATEGORIES
STATIC_PAGES
STATIC_PAGE_AUTHORS
USERS
...
```

after which you will typically type `describe table_name_of_interest` in SQL*Plus:

```
SQL> describe users;
 Name                           Null?    Type
 ------------------------------ -------- ----
 USER_ID                        NOT NULL NUMBER(38)
 FIRST_NAMES                    NOT NULL VARCHAR2(100)
 LAST_NAME                      NOT NULL VARCHAR2(100)
 PRIV_NAME                               NUMBER(38)
 EMAIL                          NOT NULL VARCHAR2(100)
 PRIV_EMAIL                              NUMBER(38)
 EMAIL_BOUNCING_P                        CHAR(1)
 PASSWORD                       NOT NULL VARCHAR2(30)
 URL                                     VARCHAR2(200)
 ON_VACATION_UNTIL                       DATE
 LAST_VISIT                              DATE
 SECOND_TO_LAST_VISIT                    DATE
 REGISTRATION_DATE                       DATE
 REGISTRATION_IP                         VARCHAR2(50)
 ADMINISTRATOR_P                         CHAR(1)
 DELETED_P                               CHAR(1)
 BANNED_P                                CHAR(1)
 BANNING_USER                            NUMBER(38)
 BANNING_NOTE                            VARCHAR2(4000)
```

Note that Oracle displays its internal data types rather than the ones you've given, e.g., `number(38)` rather than `integer` and `varchar2` instead of the specified `varchar`.

Constraints

When you're defining a table, you can constrain single rows by adding some magic words after the data type:

- `not null`; requires a value for this column
- `unique`; two rows can't have the same value in this column (side effect in Oracle: creates an index)
- `primary key`; same as `unique` except that no row can have a null value for this column and other tables can refer to this column
- `check`; limit the range of values for column, e.g., `rating integer check(rating > 0 and rating <= 10)`
- `references`; this column can only contain values present in another table's primary key column, e.g., `user_id not null references users` in the `bboard` table forces the `user_id` column to only point to valid users. An interesting twist is that you don't have to give a data type for `user_id`; Oracle assigns this column to whatever data type the foreign key has (in this case `integer`).

Constraints can apply to multiple columns:

```
create table static_page_authors (
        page_id         integer not null references static_pages,
        user_id         integer not null references users,
        notify_p        char(1) default 't' check (notify_p in ('t','f')),
        unique(page_id,user_id)
);
```

Oracle will let us keep rows that have the same `page_id` and rows that have the same `user_id` but not rows that have the same value in both columns (which would not make sense; a person can't be the author of a document more than once). Suppose that you run a university distinguished lecture series. You want speakers who are professors at other universities or at least PhDs. On the other hand, if someone controls enough money, be it his own or his company's, he's in. Oracle stands ready:

```
create table distinguished_lecturers (
        lecturer_id             integer primary key,
        name_and_title          varchar(100),
        personal_wealth         number,
        corporate_wealth        number,
        check (instr(upper(name_and_title),'PHD') <> 0
               or instr(upper(name_and_title),'PROFESSOR') <> 0
               or (personal_wealth + corporate_wealth) > 1000000000)
);

insert into distinguished_lecturers
values
(1,'Professor Ellen Egghead',-10000,200000);
```

```
1 row created.

insert into distinguished_lecturers
values
(2,'Bill Gates, innovator',75000000000,18000000000);

1 row created.

insert into distinguished_lecturers
values
(3,'Joe Average',20000,0);

ORA-02290: check constraint (PHOTONET.SYS_C001819) violated
```

As desired, Oracle prevented us from inserting some random average loser into the `distinguished_lecturers` table, but the error message was confusing in that it refers to a constraint given the name of "SYS_C001819" and owned by the PHOTONET user. We can give our constraint a name at definition time:

```
create table distinguished_lecturers (
        lecturer_id            integer primary key,
        name_and_title         varchar(100),
        personal_wealth        number,
        corporate_wealth       number,
        constraint ensure_truly_distinguished
        check (instr(upper(name_and_title),'PHD') <> 0
               or instr(upper(name_and_title),'PROFESSOR') <> 0
               or (personal_wealth + corporate_wealth) > 1000000000)
);

insert into distinguished_lecturers
values
(3,'Joe Average',20000,0);

ORA-02290: check constraint (PHOTONET.ENSURE_TRULY_DISTINGUISHED) violated
```

Now the error message is easier to understand by application programmers.

Creating More Elaborate Constraints with Triggers

The default Oracle mechanisms for constraining data are not always adequate. For example, the ArsDigita Community System auction module has a table called `au_categories`. The `category_keyword` column is a unique shorthand way of referring to a category in a URL. However, this column may be NULL because it is not the primary key to the table. The shorthand method of referring to the category is optional.

```
create table au_categories (
        category_id             integer primary key,
        -- shorthand for referring to this category,
        -- e.g. "bridges", for use in URLs
        category_keyword        varchar(30),
        -- human-readable name of this category,
        -- e.g. "All types of bridges"
        category_name           varchar(128) not null
```

```
);
```

We can't add a UNIQUE constraint to the `category_keyword` column. That would allow the table to only have one row where `category_keyword` was NULL. So we add a trigger that can execute an arbitrary PL/SQL expression and raise an error to prevent an INSERT if necessary:

```
create or replace trigger au_category_unique_tr
before insert
on au_categories
for each row
declare
        existing_count integer;
begin
        select count(*) into existing_count from au_categories
            where category_keyword = :new.category_keyword;
        if existing_count > 0
        then
            raise_application_error(-20010, 'Category keywords must be unique if
used');
        end if;
end;
```

This trigger queries the table to find out if there are any matching keywords already inserted. If there are, it calls the built-in Oracle procedure `raise_application_error` to abort the transaction.

The True Oracle Religion

- Oracle8 Server Application Developer's Guide, Selecting a Datatype
- Oracle8 Server Concepts, Built-In Datatypes
- Oracle8 Server Concepts, User-Defined Datatypes

Queries

part of SQL for Web Nerds by Philip Greenspun

If you start up SQL*Plus, you can start browsing around immediately with the SELECT statement. You don't even need to define a table; Oracle provides the built-in `dual` table for times when you're interested in a constant or a function:

```
SQL> select 'Hello World' from dual;

'HELLOWORLD
-----------
Hello World

SQL> select 2+2 from dual;

       2+2
----------
         4

SQL> select sysdate from dual;

SYSDATE
----------
1999-02-14
```

... or to test your knowledge of three-valued logic (see the "Data Modeling" chapter):

```
SQL> select 4+NULL from dual;

    4+NULL
----------
```

(any expression involving NULL evaluates to NULL).

There is nothing magic about the `dual` table for these purposes; you can compute functions using the `bboard` table instead of `dual`:

```
select sysdate,2+2,atan2(0, -1) from bboard;

SYSDATE          2+2 ATAN2(0,-1)
---------- ---------- -----------
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
...

1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265

55010 rows selected.
```

but not everyone wants 55010 copies of the same result. The `dual` table is predefined during Oracle installation and, though it is just a plain old table, it is guaranteed to contain only one row because no user will have sufficient privileges to insert or delete rows from `dual`.

Getting beyond Hello World

To get beyond Hello World, pick a table of interest. As we saw in the introduction,

```
select * from users;
```

would retrieve all the information from every row of the `users` table. That's good for toy systems but in any production system, you'd be better off starting with

```
SQL> select count(*) from users;

  COUNT(*)
----------
      7352
```

You don't really want to look at 7352 rows of data, but you would like to see what's in the users table, start off by asking SQL*Plus to query Oracle's data dictionary and figure out what columns are available in the `users` table:

```
SQL> describe users
 Name                           Null?    Type
 ------------------------------ -------- ----
 USER_ID                        NOT NULL NUMBER(38)
 FIRST_NAMES                    NOT NULL VARCHAR2(100)
 LAST_NAME                      NOT NULL VARCHAR2(100)
 PRIV_NAME                               NUMBER(38)
 EMAIL                          NOT NULL VARCHAR2(100)
 PRIV_EMAIL                              NUMBER(38)
 EMAIL_BOUNCING_P                        CHAR(1)
 PASSWORD                       NOT NULL VARCHAR2(30)
 URL                                     VARCHAR2(200)
 ON_VACATION_UNTIL                       DATE
 LAST_VISIT                              DATE
 SECOND_TO_LAST_VISIT                    DATE
 REGISTRATION_DATE                       DATE
 REGISTRATION_IP                         VARCHAR2(50)
 ADMINISTRATOR_P                         CHAR(1)
 DELETED_P                               CHAR(1)
 BANNED_P                                CHAR(1)
 BANNING_USER                            NUMBER(38)
 BANNING_NOTE                            VARCHAR2(4000)
```

The data dictionary is simply a set of built-in tables that Oracle uses to store information about the objects (tables, triggers, etc.) that have been defined. Thus SQL*Plus isn't performing any black magic when you type `describe`; it is simply querying `user_tab_columns`, a view of some of the tables in Oracle's data dictionary. You could do the same explicitly, but it is a little cumbersome.

```
column fancy_type format a20
select column_name, data_type || '(' || data_length || ')' as fancy_type
from user_tab_columns
where table_name = 'USERS'
order by column_id;
```

Here we've had to make sure to put the table name ("USERS") in all-uppercase. Oracle is case-insensitive for table and column names in queries but the data dictionary records names in uppercase. Now that we know the names of the columns in the table, it will be easy to explore.

Simple Queries from One Table

A simple query from one table has the following structure:

- the select list (which columns in our report)
- the name of the table
- the where clauses (which rows we want to see)
- the order by clauses (how we want the rows arranged)

Let's see some examples. First, let's see how many users from MIT are registered on our site:

```
SQL> select email
from users
where email like '%mit.edu';

EMAIL
------------------------------
philg@mit.edu
andy@california.mit.edu
ben@mit.edu
...
wollman@lcs.mit.edu
ghomsy@mit.edu
hal@mit.edu
...
jpearce@mit.edu
richmond@alum.mit.edu
andy_roo@mit.edu
kov@mit.edu
fletch@mit.edu
lsandon@mit.edu
psz@mit.edu
philg@ai.mit.edu
philg@martigny.ai.mit.edu
andy@californnia.mit.edu
ty@mit.edu
teadams@mit.edu

68 rows selected.
```

The `email like '%mit.edu'` says "every row where the email column ends in 'mit.edu'". The percent sign is Oracle's wildcard character for "zero or more characters". Underscore is the wildcard for "exactly one character":

```
SQL> select email
from users
where email like '___@mit.edu';

EMAIL
------------------------------
kov@mit.edu
hal@mit.edu
```

```
...
ben@mit.edu
psz@mit.edu
```
Suppose that we notice in the above report some similar email addresses. It is perhaps time to try out the ORDER BY clause:

```
SQL> select email
from users
where email like '%mit.edu'
order by email;

EMAIL
-----------------------------
andy@california.mit.edu
andy@californnia.mit.edu
andy_roo@mit.edu
...
ben@mit.edu
...
hal@mit.edu
...
philg@ai.mit.edu
philg@martigny.ai.mit.edu
philg@mit.edu
```
Now we can see that this users table was generated by grinding over pre-ArsDigita Community Systems postings starting from 1995. In those bad old days, users typed their email address and name with each posting. Due to typos and people intentionally choosing to use different addresses at various times, we can see that we'll have to build some sort of application to help human beings merge some of the rows in the users table (e.g., all three occurrences of "philg" are in fact the same person (me)).

Restricting results

Suppose that you were featured on Yahoo in September 1998 and want to see how many users signed up during that month:

```
SQL> select count(*)
from users
where registration_date >= '1998-09-01'
and registration_date < '1998-10-01';

  COUNT(*)
----------
       920
```
We've combined two restrictions in the WHERE clause with an AND. We can add another restriction with another AND:

```
SQL> select count(*)
from users
where registration_date >= '1998-09-01'
and registration_date < '1998-10-01'
and email like '%mit.edu';
```

```
  COUNT(*)
----------
        35
```

OR and NOT are also available within the WHERE clause. For example, the following query will tell us how many classified ads we have that either have no expiration date or whose expiration date is later than the current date/time.

```
select count(*)
from classified_ads
where expires >= sysdate
or expires is null;
```

Subqueries

You can query one table, restricting the rows returned based on information from another table. For example, to find users who have posted at least one classified ad:

```
select user_id, email
from users
where 0 < (select count(*)
           from classified_ads
           where classified_ads.user_id = users.user_id);

   USER_ID EMAIL
---------- -----------------------------------
     42485 twm@meteor.com
     42489 trunghau@ecst.csuchico.edu
     42389 ricardo.carvajal@kbs.msu.edu
     42393 gon2foto@gte.net
     42399 rob@hawaii.rr.com
     42453 stefan9@ix.netcom.com
     42346 silverman@pon.net
     42153 gallen@wesleyan.edu
...
```

Conceptually, for each row in the `users` table Oracle is running the subquery against `classified_ads` to see how many ads are associated with that particular user ID. Keep in mind that this is only *conceptually*; the Oracle SQL parser may elect to execute this query in a more efficient manner.

Another way to describe the same result set is using EXISTS:

```
select user_id, email
from users
where exists (select 1
              from classified_ads
              where classified_ads.user_id = users.user_id);
```

This may be more efficient for Oracle to execute since it hasn't been instructed to actually count the number of classified ads for each user, but only to check and see if any are present. Think of EXISTS as a Boolean function that

1. takes a SQL query as its only parameter

2. returns TRUE if the query returns any rows at all, regardless of the contents of those rows (this is why we can use the constant 1 as the select list for the subquery)

JOIN

A professional SQL programmer would be unlikely to query for users who'd posted classified ads in the preceding manner. The SQL programmer knows that, inevitably, the publisher will want information from the classified ad table along with the information from the users table. For example, we might want to see the users and, for each user, the sequence of ad postings:

```
select users.user_id, users.email, classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id
order by users.email, posted;

   USER_ID EMAIL                               POSTED
---------- ----------------------------------- ----------
     39406 102140.1200@compuserve.com          1998-09-30
     39406 102140.1200@compuserve.com          1998-10-08
     39406 102140.1200@compuserve.com          1998-10-08
     39842 102144.2651@compuserve.com          1998-07-02
     39842 102144.2651@compuserve.com          1998-07-06
     39842 102144.2651@compuserve.com          1998-12-13
...
     41284 yme@inetport.com                    1998-01-25
     41284 yme@inetport.com                    1998-02-18
     41284 yme@inetport.com                    1998-03-08
     35389 zhupanov@usa.net                    1998-12-10
     35389 zhupanov@usa.net                    1998-12-10
     35389 zhupanov@usa.net                    1998-12-10
```

Because of the JOIN restriction, `where users.user_id = classified_ads.user_id`, we only see those users who have posted at least one classified ad, i.e., for whom a matching row may be found in the `classified_ads` table. This has the same effect as the subquery above.

The `order by users.email, posted` is key to making sure that the rows are lumped together by user and then printed in order of ascending posting time.

OUTER JOIN

Suppose that we want an alphabetical list of all of our users, with classified ad posting dates for those users who have posted classifieds. We can't do a simple JOIN because that will exclude users who haven't posted any ads. What we need is an OUTER JOIN, where Oracle will "stick in NULLs" if it can't find a corresponding row in the `classified_ads` table.

```
select users.user_id, users.email, classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
order by users.email, posted;

...
   USER_ID EMAIL                               POSTED
---------- ----------------------------------- ----------
```

```
     52790 dbrager@mindspring.com
     37461 dbraun@scdt.intel.com
     52791 dbrenner@flash.net
     47177 dbronz@free.polbox.pl
     37296 dbrouse@enter.net
     47178 dbrown@cyberhighway.net
     36985 dbrown@uniden.com                 1998-03-05
     36985 dbrown@uniden.com                 1998-03-10
     34283 dbs117@amaze.net
     52792 dbsikorski@yahoo.com
...
```

The plus sign after `classified_ads.user_id` is our instruction to Oracle to "add NULL rows if you can't meet this JOIN constraint".

Extending a simple query into a JOIN

Suppose that you have a query from one table returning almost everything that you need, except for one column that's in another table. Here's a way to develop the JOIN without risking breaking your application:

1. add the new table to your FROM clause
2. add a WHERE constraint to prevent Oracle from building a Cartesian product
3. hunt for ambiguous column names in the SELECT list and other portions of the query; prefix these with table names if necessary
4. test that you've not broken anything in your zeal to add additional info
5. add a new column to the SELECT list

Here's an example from Problem Set 2 of a course that we give at MIT (see http://photo.net/teaching/psets/ps2/ps2.adp). Students build a conference room reservation system. They generally define two tables: `rooms` and `reservations`. The top level page is supposed to show a user what reservations he or she is current holding:

```
select room_id, start_time, end_time
from reservations
where user_id = 37
```

This produces an unacceptable page because the rooms are referred to by an ID number rather than by name. The name information is in the `rooms` table, so we'll have to turn this into a JOIN.

Step 1: add the new table to the FROM clause

```
select room_id, start_time, end_time
from reservations, rooms
where user_id = 37
```

We're in a world of hurt because Oracle is now going to join every row in `rooms` with every row in `reservations` where the `user_id` matches that of the logged-in user.

Step 2: add a constraint to the WHERE clause

```
select room_id, start_time, end_time
from reservations, rooms
where user_id = 37
and reservations.room_id = rooms.room_id
```

Step 3: look for ambiguously defined columns

Both `reservations` and `rooms` contain columns called "room_id". So we need to prefix the `room_id` column in the SELECT list with "reservations.". Note that we don't have to prefix `start_time` and `end_time` because these columns are only present in `reservations`.

```
select reservations.room_id, start_time, end_time
from reservations, rooms
where user_id = 37
and reservations.room_id = rooms.room_id
```

Step 4: test

Test the query to make sure that you haven't broken anything. You should get back the same rows with the same columns as before.

Step 5: add a new column to the SELECT list

We're finally ready to do what we set out to do: add `room_name` to the list of columns for which we're querying.

```
select reservations.room_id, start_time, end_time, rooms.room_name
from reservations, rooms
where user_id = 37
and reservations.room_id = rooms.room_id
```

Reference

- Oracle8 Server SQL Reference, SELECT command section

Complex Queries

part of SQL for Web Nerds by Philip Greenspun

Suppose that you want to start lumping together information from multiple rows. For example, you're interested in JOINing users with their classified ads. That will give you one row per ad posted. But you want to mush all the rows together for a particular user and just look at the most recent posting time. What you need is the GROUP BY construct:

```
select users.user_id, users.email, max(classified_ads.posted)
from users, classified_ads
where users.user_id = classified_ads.user_id
group by users.user_id, users.email
order by upper(users.email);

   USER_ID EMAIL                                MAX(CLASSI
---------- ---------------------------------- ----------
     39406 102140.1200@compuserve.com         1998-10-08
     39842 102144.2651@compuserve.com         1998-12-13
     41426 50@seattle.va.gov                  1997-01-13
     37428 71730.345@compuserve.com           1998-11-24
     35970 aaibrahim@earthlink.net            1998-11-08
     36671 absolutsci@aol.com                 1998-10-06
     35781 alevy@agtnet.com                   1997-07-14
     40111 alexzorba@aol.com                  1998-09-25
     39060 amchiu@worldnet.att.net            1998-12-11
     35989 andrew.c.beckner@bankamerica.com    1998-08-13
     33923 andy_roo@mit.edu                   1998-12-10
```

The `group by users.user_id, users.email` tells SQL to "lump together all the rows that have the same values in these two columns." In addition to the grouped by columns, we can run aggregate functions on the columns that aren't being grouped. For example, the MAX above applies to the posting dates for the rows in a particular group. We can also use COUNT to see at a glance how active and how recently active a user has been:

```
select users.user_id, users.email, count(*), max(classified_ads.posted)
from users, classified_ads
where users.user_id = classified_ads.user_id
group by users.user_id, users.email
order by upper(users.email);

   USER_ID EMAIL                                COUNT(*) MAX(CLASSI
---------- ---------------------------------- ---------- ----------
```

```
     39406 102140.1200@compuserve.com                3 1998-10-08
     39842 102144.2651@compuserve.com                3 1998-12-13
     41426 50@seattle.va.gov                         1 1997-01-13
     37428 71730.345@compuserve.com                  3 1998-11-24
     35970 aaibrahim@earthlink.net                   1 1998-11-08
     36671 absolutsci@aol.com                        2 1998-10-06
     35781 alevy@agtnet.com                          1 1997-07-14
     40111 alexzorba@aol.com                         1 1998-09-25
     39060 amchiu@worldnet.att.net                   1 1998-12-11
     35989 andrew.c.beckner@bankamerica.com          1 1998-08-13
     33923 andy_roo@mit.edu                          1 1998-12-10
```

A publisher who was truly curious about this stuff probably wouldn't be interested in these results alphabetically. Let's find our most recently active users. At the same time, let's get rid of the unsightly "MAX(CLASSI" at the top of the report:

```
select users.user_id,
       users.email,
       count(*) as how_many,
       max(classified_ads.posted) as how_recent
from users, classified_ads
where users.user_id = classified_ads.user_id
group by users.user_id, users.email
order by how_recent desc, how_many desc;

   USER_ID EMAIL                                HOW_MANY HOW_RECENT
---------- ---------------------------------- ---------- ----------
     39842 102144.2651@compuserve.com                3 1998-12-13
     39968 mkravit@mindspring.com                    1 1998-12-13
     36758 mccallister@mindspring.com                1 1998-12-13
     38513 franjeff@alltel.net                       1 1998-12-13
     34530 nverdesoto@earthlink.net                  3 1998-12-13
     34765 jrl@blast.princeton.edu                   1 1998-12-13
     38497 jeetsukumaran@pd.jaring.my                1 1998-12-12
     38879 john.macpherson@btinternet.com            5 1998-12-12
     37808 eck@coastalnet.com                        1 1998-12-12
     37482 dougcan@arn.net                           1 1998-12-12
```

Note that we were able to use our *correlation names* of "how_recent" and "how_many" in the ORDER BY clause. The `desc` ("descending") directives in the ORDER BY clause instruct Oracle to put the largest values at the top. The default sort order is from smallest to largest ("ascending").

Upon close inspection, the results are confusing. We instructed Oracle to rank first by date and second by number of postings. Yet for 1998-12-13 we don't see both users with three total postings at the top. That's because Oracle dates are precise to the second even when the hour, minute, and second details are not displayed by SQL*Plus. A better query would include the clause

```
order by trunc(how_recent) desc, how_many desc
```

where the built-in Oracle function `trunc` truncates each date to midnight on the day in question.

Finding co-moderators: The HAVING Clause

The WHERE clause restricts which rows are returned. The HAVING clause operates analogously but on groups of rows. Suppose, for example, that we're interested in finding those users who've contributed heavily to our discussion forum:

```
select user_id, count(*) as how_many
from bboard
group by user_id
order by how_many desc;

   USER_ID   HOW_MANY
---------- ----------
     34474       1922
     35164        985
     41112        855
     37021        834
     34004        823
     37397        717
     40375        639
...
     33963          1
     33941          1
     33918          1

7348 rows selected.
```

Seventy three hundred rows. That's way too big considering that we are only interested in nominating a couple of people. Let's restrict to more recent activity. A posting contributed three years ago is not necessarily evidence of interest in the community right now.

```
select user_id, count(*) as how_many
from bboard
where posting_time + 60 > sysdate
group by user_id
order by how_many desc;

   USER_ID   HOW_MANY
---------- ----------
     34375         80
     34004         79
     37903         49
     41074         46
...

1120 rows selected.
```

We wanted to kill rows, not groups, so we did it with a WHERE clause. Let's get rid of the people who are already serving as maintainers.

```
select user_id, count(*) as how_many
from bboard
where not exists (select 1 from
                  bboard_authorized_maintainers bam
                  where bam.user_id = bboard.user_id)
```

```
and posting_time + 60 > sysdate
group by user_id
order by how_many desc;
```

The concept of User ID makes sense for both rows and groups, so we can restrict our results either with the extra WHERE clause above or by letting the relational database management system produce the groups and then we'll ask that they be tossed out using a HAVING clause:

```
select user_id, count(*) as how_many
from bboard
where posting_time + 60 > sysdate
group by user_id
having not exists (select 1 from
                   bboard_authorized_maintainers bam
                   where bam.user_id = bboard.user_id)
order by how_many desc;
```

This doesn't get to the root cause of our distressingly large query result: we don't want to see groups where `how_many` is less than 30. Here's the SQL for "show me users who've posted at least 30 messages in the past 60 days, ranked in descending order of volubility":

```
select user_id, count(*) as how_many
from bboard
where posting_time + 60 > sysdate
group by user_id
having count(*) >= 30
order by how_many desc;

   USER_ID   HOW_MANY
---------- ----------
     34375         80
     34004         79
     37903         49
     41074         46
     42485         46
     35387         30
     42453         30

7 rows selected.
```

We had to do this in a HAVING clause because the number of rows in a group is a concept that doesn't make sense at the per-row level on which WHERE clauses operate.

Oracle 8's SQL parser is too feeble to allow you to use the `how_many` correlation variable in the HAVING clause. You therefore have to repeat the `count(*)` incantation.

Set Operations: UNION, INTERSECT, and MINUS

Oracle provides set operations that can be used to combine rows produced by two or more separate SELECT statements. UNION pools together the rows returned by two queries, removing any duplicate rows. INTERSECT combines the result sets of two queries by removing any rows that are not present in both. MINUS combines the results of two queries by taking the the first result set and subtracting from it any rows that are also found in the second. Of the three, UNION is the most useful in practice.

In the ArsDigita Community System ticket tracker, people reporting a bug or requesting a feature are given a menu of potential deadlines. For some projects, common project deadlines are stored in the

`ticket_deadlines` table. These should appear in an HTML SELECT form element. We also, however, want some options like "today", "tomorrow", "next week", and "next month". The easiest way to handle these is to query the `dual` table to perform some date arithmetic. Each of these queries will return one row and if we UNION four of them together with the query from `ticket_deadlines`, we can have the basis for a very simple Web script to present the options:

```
select
  'today - ' || to_char(trunc(sysdate),'Mon FMDDFM'),
  trunc(sysdate) as deadline
from dual
UNION
select
  'tomorrow - '|| to_char(trunc(sysdate+1),'Mon FMDDFM'),
  trunc(sysdate+1) as deadline
from dual
UNION
select
  'next week - '|| to_char(trunc(sysdate+7),'Mon FMDDFM'),
  trunc(sysdate+7) as deadline
from dual
UNION
select
  'next month - '|| to_char(trunc(ADD_MONTHS(sysdate,1)),'Mon FMDDFM'),
  trunc(ADD_MONTHS(sysdate,1)) as deadline
from dual
UNION
select
  name || ' - ' || to_char(deadline, 'Mon FMDDFM'),
  deadline
from ticket_deadlines
where project_id = :project_id
and deadline >= trunc(sysdate)
order by deadline
```

will produce something like

The INTERSECT and MINUS operators are seldom used. Here are some contrived examples. Suppose that you collect contest entries by Web users, each in a separate table:

```
create table trip_to_paris_contest (
        user_id         references users,
        entry_date      date not null
);

create table camera_giveaway_contest (
        user_id         references users,
        entry_date      date not null
);
```

Now let's populate with some dummy data:

```
-- all three users love to go to Paris
insert into trip_to_paris_contest values (1,'2000-10-20');
insert into trip_to_paris_contest values (2,'2000-10-22');
insert into trip_to_paris_contest values (3,'2000-10-23');

-- only User #2 is a camera nerd
insert into camera_giveaway_contest values (2,'2000-05-01');
```

Suppose that we've got a new contest on the site. This time we're giving away a trip to Churchill, Manitoba to photograph polar bears. We assume that the most interested users will be those who've entered both the travel and the camera contests. Let's get their user IDs so that we can notify them via email (spam) about the new contest:

```
select user_id from trip_to_paris_contest
intersect
select user_id from camera_giveaway_contest;

   USER_ID
----------
         2
```

Or suppose that we're going to organize a personal trip to Paris and want to find someone to share the cost of a room at the Crillon. We can assume that anyone who entered the Paris trip contest is interested in going. So perhaps we should start by sending them all email. On the other hand, how can one enjoy a quiet evening with the absinthe bottle if one's companion is constantly blasting away with an electronic flash? We're interested in people who entered the Paris trip contest but who *did not* enter the camera giveaway:

```
select user_id from trip_to_paris_contest
minus
select user_id from camera_giveaway_contest;

   USER_ID
----------
         1
         3
```

Next: Transactions

---

philg@mit.edu

Reader's Comments

In less trivial uses of UNION, you can use UNION ALL, instructing Oracle not to remove duplicates and saving the sort if you know there aren't going to be any duplicate rows(or maybe don't care)

-- Neal Sidhwaney, December 10, 2002

Another example of using MINUS is shown in the following crazy-looking (and Oracle-specific [1]) query which selects the 91st through 100th rows of a subquery.

```
with subq as (select * from my_table order by my_id)

select * from subq
where rowid in (select rowid from subq
                where rownum <= 100
                    MINUS
                select rowid from subq
                where rownum <= 90)
```

[1] The Oracle dependencies in this query are rowid and rownum. Other databases have other means of limiting query results by row position.

-- Kevin Murphy, February 10, 2003

And in PostgreSQL (and MySQL too for that matter) it is as simple as:

select * from my_table order by my_id limit 90,10

An easier way for Oracle (according to a random post in a devshed.com forum I googled) would be like this:

select * from my_table order by my_id where rownum between 90,100

(Though the whole point about how to use MINUS is well taken)

-- Gabriel Ricard, February 26, 2003

Oops. I was wrong. Phil emailed me and explained that my rownum example won't work (just goes to show that not everything you find on the internet is right!).

-- Gabriel Ricard, March 17, 2003

Transactions

part of SQL for Web Nerds by Philip Greenspun

In the introduction we covered some examples of inserting data into a database by typing at SQL*Plus:

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

Generally, this is not how it is done. As a programmer, you write code that gets executed every time a user submits a discussion forum posting or classified ad. The structure of the SQL statement remains fixed but not the string literals after the `values`.

The simplest and most direct interface to a relational database involves a procedural program in C, Java, Lisp, Perl, or Tcl putting together a string of SQL that is then sent to to the RDBMS. Here's how the ArsDigita Community System constructs a new entry in the clickthrough log:

```
insert into clickthrough_log
 (local_url, foreign_url, entry_date, click_count)
values
 ('$local_url', '$foreign_url', trunc(sysdate), 1)"
```

The INSERT statement adds one row, filling in the four list columns. Two of the values come from local variables set within the Web server, `$local_url` and `$foreign_url`. Because these are strings, they must be surrounded by single quotes. One of the values is dynamic and comes straight from Oracle: `trunc(sysdate)`. Recall that the `date` data type in Oracle is precise to the second. We only want one of these rows per day of the year and hence truncate the date to midnight. Finally, as this is the first clickthrough of the day, we insert a constant value of 1 for `click_count`.

Atomicity

Each SQL statement executes as an atomic transaction. For example, suppose that you were to attempt to purge some old data with

```
delete from clickthrough_log where entry_date + 120 < sysdate;
```

(delete clickthrough records more than 120 days old) and that 3500 rows in `clickthrough_log` are older than 120 days. If your computer failed halfway through the execution of this DELETE, i.e., before the transaction committed, you would find that none of the rows had been deleted. Either all 3500 rows will disappear or none will.

More interestingly, you can wrap a transaction around multiple SQL statements. For example, when a user is editing a comment, the ArsDigita Community System keeps a record of what was there before:

```
ns_db dml $db "begin transaction"

# insert into the audit table
ns_db dml $db "insert into general_comments_audit
 (comment_id, user_id, ip_address, audit_entry_time, modified_date, content)
select comment_id,
       user_id,
       '[ns_conn peeraddr]',
       sysdate,
       modified_date,
       content from general_comments
where comment_id = $comment_id"

# change the publicly viewable table
ns_db dml $db "update general_comments
set content = '$QQcontent',
    html_p = '$html_p'
```

```
where comment_id = $comment_id"

# commit the transaction
ns_db dml $db "end transaction"
```

This is generally referred to in the database industry as *auditing*. The database itself is used to keep track of what has been changed and by whom.

Let's look at these sections piece by piece. We're looking at a Tcl program calling AOLserver API procedures when it wants to talk to Oracle. We've configured the system to reverse the normal Oracle world order in which everything is within a transaction unless otherwise committed. The `begin transaction` and `end transaction` statements never get through to Oracle; they are merely instructions to our Oracle driver to flip Oracle out and then back into autocommit mode.

The transaction wrapper is imposed around two SQL statements. The first statement inserts a row into `general_comments_audit`. We could simply query the `general_comments` table from Tcl and then use the returned data to create a standard-looking INSERT. However, if what you're actually doing is moving data from one place within the RDBMS to another, it is extremely bad taste to drag it all the way out to an application program and then stuff it back in. Much better to use the "INSERT ... SELECT" form.

Note that two of the columns we're querying from `general_comments` don't exist in the table: `sysdate` and `'[ns_conn peeraddr]'`. It is legal in SQL to put function calls or constants in your select list, just as you saw at the beginning of the Queries chapter where we discussed Oracle's one-row system table: `dual`. To refresh your memory:

```
select sysdate from dual;

SYSDATE
----------
1999-01-14
```

You can compute multiple values in a single query:

```
select sysdate, 2+2, atan2(0, -1) from dual;

SYSDATE          2+2 ATAN2(0,-1)
---------- ---------- -----------
1999-01-14          4  3.14159265
```

This approach is useful in the transaction above, where we combine information from a table with constants and function calls. Here's a simpler example:

```
select posting_time, 2+2
from bboard
where msg_id = '000KWj';

POSTING_TI       2+2
---------- ----------
1998-12-13          4
```

Let's get back to our comment editing transaction and look at the basic structure:

- open a transaction
- insert into an audit table whatever comes back from a SELECT statement on the comment table
- update the comment table
- close the transaction

Suppose that something goes wrong during the INSERT. The tablespace in which the audit table resides is full and it isn't possible to add a row. Putting the INSERT and UPDATE in the same RDBMS transactions ensures that if there is a problem with one, the other won't be applied to the database.

Consistency

Suppose that we've looked at a message on the bulletin board and decide that its content is so offensive we wish to delete the user from our system:

```
select user_id from bboard where msg_id = '000KWj';

   USER_ID
----------
     39685

delete from users where user_id = 39685;
*
ERROR at line 1:
ORA-02292: integrity constraint (PHOTONET.SYS_C001526) violated - child record
found
```

Oracle has stopped us from deleting user 39685 because to do so would leave the database in an inconsistent state. Here's the definition of the bboard table:

```
create table bboard (
        msg_id          char(6) not null primary key,
        refers_to       char(6),
         ...
        user_id         integer not null references users,
        one_line        varchar(700),
        message         clob,
         ...
);
```

The `user_id` column is constrained to be not null. Furthermore, the value in this column must correspond to some row in the `users` table (`references users`). By asking Oracle to delete the author of msg_id 000KWj from the `users` table before we deleted all of his or her postings from the `bboard` table, we were asking Oracle to leave the RDBMS in an inconsistent state.

Mutual Exclusion

When you have multiple simultaneously executing copies of the same program, you have to think about *mutual exclusion*. If a program has to

- read a value from the database
- perform a computation based on that value

- update the value in the database based on the computation

Then you want to make sure only one copy of the program is executing at a time through this segment.
The /bboard module of the ArsDigita Community System has to do this. The sequence is

- read the last message ID from the `msg_id_generator` table
- increment the message ID with a bizarre collection of Tcl scripts
- update the `last_msg_id` column in the `msg_id_generator` table

First, anything having to do with locks only makes sense when the three operations are grouped together in a transaction. Second, to avoid deadlocks a transaction must acquire all the resources (including locks) that it needs at the start of the transaction. A SELECT in Oracle does not acquire any locks but a SELECT .. FOR UPDATE does. Here's the beginning of the transaction that inserts a message into the `bboard` table (from /bboard/insert-msg.tcl):

```
select last_msg_id
from msg_id_generator
for update of last_msg_id
```

Mutual Exclusion (the Big Hammer)

The `for update` clause isn't a panacea. For example, in the Action Network (described in Chapter 16 of Philip and Alex's Guide to Web Publishing), we need to make sure that a double-clicking user doesn't generate duplicate FAXes to politicians. The test to see if the user has already responded is

```
select count(*) from an_alert_log
where member_id = $member_id
and entry_type = 'sent_response'
and alert_id = $alert_id
```

By default, Oracle locks one row at a time and doesn't want you to throw a FOR UPDATE clause into a SELECT COUNT(*). The implication of that would be Oracle recording locks on every row in the table. Much more efficient is simply to start the transaction with

```
lock table an_alert_log in exclusive mode
```

This is a big hammer and you don't want to hold a table lock for more than an instant. So the structure of a page that gets a table lock should be

- open a transaction
- lock table
- select count(*)
- if the count was 0, insert a row to record the fact that the user has responded
- commit the transaction (releases the table lock)
- proceed with the rest of the script
- ...

What if I just want some unique numbers?

Does it really have to be this hard? What if you just want some unique integers, each of which will be used as a primary key? Consider a table to hold news items for a Web site:

```
create table news (
        title           varchar(100) not null,
        body            varchar(4000) not null,
        release_date    date not null,
        ...
);
```

You might think you could use the `title` column as a key, but consider the following articles:

```
insert into news (title, body, release_date)
values
('French Air Traffic Controllers Strike',
 'A walkout today by controllers left travelers stranded..',
 '1995-12-14');

insert into news (title, body, release_date)
values
('French Air Traffic Controllers Strike',
 'Passengers at Orly faced 400 canceled flights ...',
 '1997-05-01');

insert into news (title, body, release_date)
values
('Bill Clinton Beats the Rap',
 'Only 55 senators were convinced that President Clinton obstructed justice ...',
 '1999-02-12');

insert into news (title, body, release_date)
values
('Bill Clinton Beats the Rap',
 'The sexual harassment suit by Paula Jones was dismissed ...',
 '1998-12-02);
```

It would seem that, at least as far as headlines are concerned, little of what is reported is truly new. Could we add

```
        primary key (title, release_date)
```

at the end of our table definition? Absolutely. But keying by title and date would result in some unwieldy URLs for editing or approving news articles. If your site allows public suggestions, you might find submissions from multiple users colliding. If you accept comments on news articles, a standard feature of the ArsDigita Community System, each comment must reference a news article. You'd have to be sure to update both the comments table and the news table if you needed to correct a typo in the `title` column or changed the `release_date`.

The traditional database design that gets around all of these problems is the use of a generated key. If you've been annoyed by having to carry around your student ID at MIT or your patient ID at a hospital, now you understand the reason why: the programmers are using generated keys and making their lives a bit easier by exposing this part of their software's innards.

Here's how the news module of the ArsDigita Community System works, in an excerpt from http://software.arsdigita.com/www/doc/sql/news.sql:

```
create sequence news_id_sequence start with 1;

create table news (
        news_id         integer primary key,
        title           varchar(100) not null,
        body            varchar(4000) not null,
        release_date    date not null,
        ...
);
```

We're taking advantage of the nonstandard but very useful Oracle *sequence* facility. In almost any Oracle SQL statement, you can ask for a sequence's current value or next value.

```
SQL> create sequence foo_sequence;

Sequence created.

SQL> select foo_sequence.currval from dual;

ERROR at line 1:
ORA-08002: sequence FOO_SEQUENCE.CURRVAL is not yet defined in this session
```

Oops! Looks like we can't ask for the current value until we've asked for at least one key in our current session with Oracle.

```
SQL> select foo_sequence.nextval from dual;

   NEXTVAL
----------
         1

SQL> select foo_sequence.nextval from dual;

   NEXTVAL
----------
         2

SQL> select foo_sequence.nextval from dual;

   NEXTVAL
----------
         3

SQL> select foo_sequence.currval from dual;

   CURRVAL
----------
         3
```

You can use the sequence generator directly in an insert, e.g.,

```
insert into news (news_id, title, body, release_date)
```

```
values
(news_id_sequence.nextval,
 'Tuition Refund at MIT',
 'Administrators were shocked and horrified ...',
 '1998-03-12);
```

Background on this story: http://philip.greenspun.com/school/tuition-free-mit.html

In the ArsDigita Community System implementation, the `news_id` is actually generated in /news/post-new-2.tcl:

```
set news_id [database_to_tcl_string $db "select news_id_sequence.nextval from
dual"]
```

This way the page that actually does the database insert, /news/post-new-3.tcl, can be sure when the user has inadvertently hit submit twice:

```
if [catch { ns_db dml $db "insert into news
(news_id, title, body, html_p, approved_p,
 release_date, expiration_date, creation_date, creation_user,
 creation_ip_address)
values
($news_id, '$QQtitle', '$QQbody', '$html_p', '$approved_p',
 '$release_date', '$expiration_date', sysdate, $user_id,
 '$creation_ip_address')" } errmsg] {
    # insert failed; let's see if it was because of duplicate submission
    if { [database_to_tcl_string $db "select count(*)
                                      from news
                                      where news_id = $news_id"] == 0 } {
        # some error other than dupe submission
        ad_return_error "Insert Failed" "The database ..."
        return
    }
    # we don't bother to handle the cases where there is a dupe submission
    # because the user should be thanked or redirected anyway
}
```

In our experience, the standard technique of generating the key at the same time as the insert leads to a lot of duplicate information in the database.

Sequence Caveats

Oracle sequences are optimized for speed. Hence they offer the minimum guarantees that Oracle thinks are required for primary key generation and no more.

If you ask for a few nextvals and roll back your transaction, the sequence will not be rolled back.

You can't rely on sequence values to be, uh, sequential. They will be unique. They will be monotonically increasing. But there might be gaps. The gaps arise because Oracle pulls, by default, 20 sequence values into memory and records those values as used on disk. This makes nextval very fast since the new value need only be marked use in RAM and not on disk. But suppose that someone pulls the plug on your database server after only two sequence values have been handed out. If your database administrator and system administrator are working well together, the computer will come back to life

running Oracle. But there will be a gap of 18 values in the sequence (e.g., from 2023 to 2041). That's because Oracle recorded 20 values used on disk and only handed out 2.

More

- Oracle8 Server Application Developer's Guide, Controlling Transactions
- Orace8 Server SQL Reference, CREATE SEQUENCE section

Triggers

part of SQL for Web Nerds by Philip Greenspun

A trigger is a fragment of code that you tell Oracle to run before or after a table is modified. A trigger has the power to

- make sure that a column is filled in with default information
- make sure that an audit row is inserted into another table
- after finding that the new information is inconsistent with other stuff in the database, raise an error that will cause the entire transaction to be rolled back

Consider the `general_comments` table:

```
create table general_comments (
        comment_id              integer primary key,
        on_what_id              integer not null,
        on_which_table          varchar(50),
        user_id                 not null references users,
        comment_date            date not null,
        ip_address              varchar(50) not null,
        modified_date  date not null,
        content                 clob,
        -- is the content in HTML or plain text (the default)
        html_p                  char(1) default 'f' check(html_p in ('t','f')),
        approved_p              char(1) default 't' check(approved_p in ('t','f'))
);
```

Users and administrators are both able to edit comments. We want to make sure that we know when a comment was last modified so that we can offer the administrator a "recently modified comments

page". Rather than painstakingly go through all of our Web scripts that insert or update comments, we can specify an invariant in Oracle that "after every time someone touches the `general_comments` table, make sure that the `modified_date` column is set equal to the current date-time." Here's the trigger definition:

```
create trigger general_comments_modified
before insert or update on general_comments
for each row
begin
 :new.modified_date := sysdate;
end;
/
show errors
```

We're using the PL/SQL programming language, discussed in the procedural language chapter. In this case, it is a simple `begin-end` block that sets the `:new` value of `modified_date` to the result of calling the `sysdate` function.

When using SQL*Plus, you have to provide a / character to get the program to evaluate a trigger or PL/SQL function definition. You then have to say "show errors" if you want SQL*Plus to print out what went wrong. Unless you expect to write perfect code all the time, it can be convenient to leave these SQL*Plus incantations in your .sql files.

An Audit Table Example

The canonical trigger example is the stuffing of an audit table. For example, in the data warehouse section of the ArsDigita Community System, we keep a table of user queries. Normally the SQL code for a query is kept in a `query_columns` table. However, sometimes a user might hand edit the generated SQL code, in which case we simply store that in the `query_sql`queries table. The SQL code for a query might be very important to a business and might have taken years to evolve. Even if we have good RDBMS backups, we don't want it getting erased because of a careless mouse click. So we add a `queries_audit` table to keep historical values of the `query_sql` column:

```
create table queries (
        query_id        integer primary key,
        query_name      varchar(100) not null,
        query_owner     not null references users,
        definition_time date not null,
        -- if this is non-null, we just forget about all the query_columns
        -- stuff; the user has hand edited the SQL
        query_sql       varchar(4000)
);

create table queries_audit (
        query_id        integer not null,
        audit_time      date not null,
        query_sql       varchar(4000)
);
```

Note first that `queries_audit` has no primary key. If we were to make `query_id` the primary key, we'd only be able to store one history item per query, which is not our intent.

How to keep this table filled? We could do it by making sure that every Web script that might update the `query_sql` column inserts a row in `queries_audit` when appropriate. But how to enforce this after we've handed off our code to other programmers? Much better to let the RDBMS enforce the auditing:

```
create or replace trigger queries_audit_sql
before update on queries
for each row
when (old.query_sql is not null and (new.query_sql is null or old.query_sql <>
new.query_sql))
begin
  insert into queries_audit (query_id, audit_time, query_sql)
  values
  (:old.query_id, sysdate, :old.query_sql);
end;
```

The structure of a row-level trigger is the following:

```
CREATE OR REPLACE TRIGGER ***trigger name***
***when*** ON ***which table***
FOR EACH ROW
***conditions for firing***
begin
  ***stuff to do***
end;
```

Let's go back and look at our trigger:

- It is named `queries_audit_sql`; this is really of no consequence so long as it doesn't conflict with the names of other triggers.
- It will be run `before update`, i.e., only when someone is executing an SQL UPDATE statement.
- It will be run only when someone is updating the table `queries`.
- It will be run only when the old value of `query_sql` is not null; we don't want to fill our audit table with NULLs.
- It will be run only when the new value of `query_sql` is different from the old value; we don't want to fill our audit table with rows because someone happens to be updating another column in `queries`. Note that SQL's three-valued logic forces us to put in an extra test for `new.query_sql is null` because `old.query_sql <> new.query_sql` will not evaluate to true when `new.query_sql` is NULL (a user wiping out the custom SQL altogether; a very important case to audit).

Reference

- Oracle Application Developer's Guide, Using Database Triggers

Views

part of SQL for Web Nerds by Philip Greenspun

The relational database provides programmers with a high degree of abstraction from the physical world of the computer. You can't tell where on the disk the RDBMS is putting each row of a table. For all you know, information in a single row might be split up and spread out across multiple disk drives. The RDBMS lets you add a column to a billion-row table. Is the new information for each row going to be placed next to the pre-existing columns or will a big new block of disk space be allocated to hold the new column value for all billion rows? You can't know and shouldn't really care.

A view is a way of building even greater abstraction.

Suppose that Jane in marketing says that she wants to see a table containing the following information:

- user_id
- email address
- number of static pages viewed
- number of bboard postings made
- number of comments made

This information is spread out among four tables. However, having read the preceding chapters of this book, you're perfectly equipped to serve Jane's needs with the following query:

```
select u.user_id,
       u.email,
       count(ucm.page_id) as n_pages,
       count(bb.msg_id) as n_msgs,
       count(c.comment_id) as n_comments
from users u, user_content_map ucm, bboard bb, comments c
where u.user_id = ucm.user_id(+)
and u.user_id = bb.user_id(+)
and u.user_id = c.user_id(+)
group by u.user_id, u.email
```

```
order by upper(email)
```
Then Jane adds "I want to see this every day, updated with the latest information. I want to have a programmer write me some desktop software that connects directly to the database and looks at this information; I don't want my desktop software breaking if you reorganize the data model."

```
create or replace view janes_marketing_view
as
select u.user_id,
       u.email,
       count(ucm.page_id) as n_pages,
       count(bb.msg_id) as n_msgs,
       count(c.comment_id) as n_comments
from users u, user_content_map ucm, bboard bb, comments c
where u.user_id = ucm.user_id(+)
and u.user_id = bb.user_id(+)
and u.user_id = c.user_id(+)
group by u.user_id, u.email
order by upper(u.email)
```
To Jane, this will look and act just like a table when she queries it:

```
select * from janes_marketing_view;
```
Why should she need to be aware that information is coming from four tables? Or that you've reorganized the RDBMS so that the information subsequently comes from six tables?

Protecting Privacy with Views

A common use of views is protecting confidential data. For example, suppose that all the people who work in a hospital collaborate by using a relational database. Here is the data model:

```
create table patients (
       patient_id     integer primary key,
       patient_name   varchar(100),
       hiv_positive_p char(1),
       insurance_p    char(1),
       ...
);
```
If a bunch of hippie idealists are running the hospital, they'll think that the medical doctors shouldn't be aware of a patient's insurance status. So when a doc is looking up a patient's medical record, the looking is done through

```
create view patients_clinical
as
select patient_id, patient_name, hiv_positive_p from patients;
```
The folks over in accounting shouldn't get access to the patients' medical records just because they're trying to squeeze money out of them:

```
create view patients_accounting
as
select patient_id, patient_name, insurance_p from patients;
```
Relational databases have elaborate permission systems similar to those on time-shared computer systems. Each person in a hospital has a unique database user ID. Permission will be granted to view or modify certain tables on a per-user or per-group-of-users basis. Generally the RDBMS permissions

facilities aren't very useful for Web applications. It is the Web server that is talking to the database, not a user's desktop computer. So the Web server is responsible for figuring out who is requesting a page and how much to show in response.

Protecting Your Own Source Code

The ArsDigita Shoppe system, described in http://photo.net/wtr/thebook/ecommerce.html, represents all orders in one table, whether they were denied by the credit card processor, returned by the user, or voided by the merchant. This is fine for transaction processing but you don't want your accounting or tax reports corrupted by the inclusion of failed orders. You can make a decision in one place as to what constitutes a reportable order and then have all of your report programs query the view:

```
create or replace view sh_orders_reportable
as
select * from sh_orders
where order_state not in ('confirmed','failed_authorization','void');
```

Note that in the privacy example (above) we were using the view to leave unwanted columns behind whereas here we are using the view to leave behind unwanted rows.

If we add some order states or otherwise change the data model, the reporting programs need not be touched; we only have to keep this view definition up to date. Note that you can define every view with "create or replace view" rather than "create view"; this saves a bit of typing when you have to edit the definition later.

If you've used `select *` to define a view and subsequently alter any of the underlying tables, you have to redefine the view. Otherwise, your view won't contain any of the new columns. You might consider this a bug but Oracle has documented it, thus turning the behavior into a feature.

Views-on-the-fly and OUTER JOIN

Let's return to our first OUTER JOIN example, from the simple queries chapter:

```
select users.user_id, users.email, classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
order by users.email, posted;

...
   USER_ID EMAIL                              POSTED
---------- ---------------------------------- ----------
     52790 dbrager@mindspring.com
     37461 dbraun@scdt.intel.com
     52791 dbrenner@flash.net
     47177 dbronz@free.polbox.pl
     37296 dbrouse@enter.net
     47178 dbrown@cyberhighway.net
     36985 dbrown@uniden.com                  1998-03-05
     36985 dbrown@uniden.com                  1998-03-10
     34283 dbs117@amaze.net
     52792 dbsikorski@yahoo.com
...
```

The plus sign after `classified_ads.user_id` is our instruction to Oracle to "add NULL rows if you can't meet this JOIN constraint".

Suppose that this report has gotten very long and we're only interested in users whose email addresses start with "db". We can add a WHERE clause constraint on the `email` column of the `users` table:

```
select users.user_id, users.email, classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
and users.email like 'db%'
order by users.email, posted;

   USER_ID EMAIL                          POSTED
---------- ------------------------------ ----------
     71668 db-designs@emeraldnet.net
    112295 db1@sisna.com
    137640 db25@umail.umd.edu
     35102 db44@aol.com                   1999-12-23
     59279 db4rs@aol.com
     95190 db@astro.com.au
     17474 db@hotmail.com
    248220 db@indianhospitality.com
     40134 db@spindelvision.com           1999-02-04
    144420 db_chang@yahoo.com
     15020 dbaaru@mindspring.com
...
```

Suppose that we decide we're only interested in classified ads since January 1, 1999. Let's try the naive approach, adding another WHERE clause constraint, this time on a column from the `classified_ads` table:

```
select users.user_id, users.email, classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
and users.email like 'db%'
and classified_ads.posted > '1999-01-01'
order by users.email, posted;

   USER_ID EMAIL                          POSTED
---------- ------------------------------ ----------
     35102 db44@aol.com                   1999-12-23
     40134 db@spindelvision.com           1999-02-04
     16979 dbdors@ev1.net                 2000-10-03
     16979 dbdors@ev1.net                 2000-10-26
    235920 dbendo@mindspring.com          2000-08-03
    258161 dbouchar@bell.mma.edu          2000-10-26
     39921 dbp@agora.rdrop.com            1999-06-03
     39921 dbp@agora.rdrop.com            1999-11-05

8 rows selected.
```

Hey! This completely wrecked our outer join! All of the rows where the user had not posted any ads have now disappeared. Why? They didn't meet the `and classified_ads.posted > '1999-01-01'` constraint. The outer join added NULLs to every column in the report where there was no corresponding row in the `classified_ads` table. The new constraint is comparing NULL to January 1, 1999 and the answer is... NULL. That's three-valued logic for you. Any computation involving a NULL turns out NULL. Each WHERE clause constraint must evaluate to true for a row to be kept in the result set of the SELECT. What's the solution? A "view on the fly". Instead of OUTER JOINing the `users` table to the `classified_ads`, we will OUTER JOIN `users` to a *view* of `classified_ads` that contains only those ads posted since January 1, 1999:

```
select users.user_id, users.email, ad_view.posted
from
  users,
  (select *
   from classified_ads
   where posted > '1999-01-01') ad_view
where users.user_id = ad_view.user_id(+)
and users.email like 'db%'
order by users.email, ad_view.posted;

   USER_ID EMAIL                          POSTED
---------- ------------------------------ ----------
     71668 db-designs@emeraldnet.net
    112295 db1@sisna.com
    137640 db25@umail.umd.edu
     35102 db44@aol.com                   1999-12-23
     59279 db4rs@aol.com
     95190 db@astro.com.au
     17474 db@hotmail.com
    248220 db@indianhospitality.com
     40134 db@spindelvision.com           1999-02-04
    144420 db_chang@yahoo.com
     15020 dbaaru@mindspring.com
...
174 rows selected.
```

Note that we've named our "view on the fly" `ad_view` for the duration of this query.

How Views Work

Programmers aren't supposed to have to think about how views work. However, it is worth noting that the RDBMS merely stores the view definition and not any of the data in a view. Querying against a view versus the underlying tables does not change the way that data are retrieved or cached. Standard RDBMS views exist to make programming more convenient or to address security concerns, not to make data access more efficient.

How *Materialized* Views Work

Starting with Oracle 8.1.5, introduced in March 1999, you can have a *materialized view*, also known as a *summary*. Like a regular view, a materialized view can be used to build a black-box abstraction for the programmer. In other words, the view might be created with a complicated JOIN, or an expensive GROUP BY with sums and averages. With a regular view, this expensive operation would be done

every time you issued a query. With a materialized view, the expensive operation is done when the view is created and thus an individual query need not involve substantial computation.

Materialized views consume space because Oracle is keeping a copy of the data or at least a copy of information derivable from the data. More importantly, a materialized view does not contain up-to-the-minute information. When you query a regular view, your results includes changes made up to the last committed transaction before your SELECT. When you query a materialized view, you're getting results as of the time that the view was created or refreshed. Note that Oracle lets you specify a refresh interval at which the materialized view will automatically be refreshed.

At this point, you'd expect an experienced Oracle user to say "Hey, these aren't new. This is the old CREATE SNAPSHOT facility that we used to keep semi-up-to-date copies of tables on machines across the network!" What is new with materialized views is that you can create them with the ENABLE QUERY REWRITE option. This authorizes the SQL parser to look at a query involving aggregates or JOINs and go to the materialized view instead. Consider the following query, from the ArsDigita Community System's /admin/users/registration-history.tcl page:

```
select
  to_char(registration_date,'YYYYMM') as sort_key,
  rtrim(to_char(registration_date,'Month')) as pretty_month,
  to_char(registration_date,'YYYY') as pretty_year,
  count(*) as n_new
from users
group by
  to_char(registration_date,'YYYYMM'),
  to_char(registration_date,'Month'),
  to_char(registration_date,'YYYY')
order by 1;
```

```
SORT_K PRETTY_MO PRET     N_NEW
------ --------- ---- ----------
199805 May       1998        898
199806 June      1998        806
199807 July      1998        972
199808 August    1998        849
199809 September 1998       1023
199810 October   1998       1089
199811 November  1998       1005
199812 December  1998       1059
199901 January   1999       1488
199902 February  1999       2148
```

For each month, we have a count of how many users registered at photo.net. To execute the query, Oracle must sequentially scan the `users` table. If the users table grew large and you wanted the query to be instant, you'd sacrifice some timeliness in the stats with

```
create materialized view users_by_month
   enable query rewrite
   refresh complete
   start with 1999-03-28
```

```
next sysdate + 1
as
select
  to_char(registration_date,'YYYYMM') as sort_key,
  rtrim(to_char(registration_date,'Month')) as pretty_month,
  to_char(registration_date,'YYYY') as pretty_year,
  count(*) as n_new
from users
group by
  to_char(registration_date,'YYYYMM'),
  to_char(registration_date,'Month'),
  to_char(registration_date,'YYYY')
order by 1
```

Oracle will build this view just after midnight on March 28, 1999. The view will be refreshed every 24 hours after that. Because of the `enable query rewrite` clause, Oracle will feel free to grab data from the view even when a user's query does not mention the view. For example, given the query

```
select count(*)
from users
where rtrim(to_char(registration_date,'Month')) = 'January'
and to_char(registration_date,'YYYY') = '1999'
```

Oracle would ignore the `users` table altogether and pull information from `users_by_month`. This would give the same result with much less work. Suppose that the current month is March 1999, though. The query

```
select count(*)
from users
where rtrim(to_char(registration_date,'Month')) = 'March'
and to_char(registration_date,'YYYY') = '1999'
```

will also hit the materialized view rather than the `users` table and hence will miss anyone who has registered since midnight (i.e., the query rewriting will cause a different result to be returned).

More:

Reference

- High level: Oracle8 Server Concepts, View section. Oracle Application Developer's Guide, Managing Views section.
- Low level: Oracle8 Server SQL Reference, Create View syntax, and Create Materialized View section

Style

part of SQL for Web Nerds by Philip Greenspun

Here's a familiar simple example from the complex queries chapter:

```
select user_id,
  count(*) as how_many from bboard
where not exists (select 1 from bboard_authorized_maintainers
bam where bam.user_id =
bboard.user_id) and
posting_time + 60 > sysdate group by user_id order
by how_many desc;
```

Doesn't seem so simple, eh? How about if we rewrite it:

```
select user_id, count(*) as how_many
from bboard
where not exists (select 1 from
                  bboard_authorized_maintainers bam
                  where bam.user_id = bboard.user_id)
and posting_time + 60 > sysdate
group by user_id
order by how_many desc;
```

If your code isn't properly indented then you will never be able to debug it. How can we justify using the word "properly"? After all, the SQL parser doesn't take extra spaces or newlines into account. **Software is indented properly when the structure of the software is revealed *and* when the indentation style is familiar to a community of programmers.**

Rules for All Queries

If it fits on one line, it is okay to leave on one line:

```
select email from users where user_id = 34;
```

If it doesn't fit nicely on one line, give each clause a separate line:

```
select *
from news
where sysdate > expiration_date
and approved_p = 't'
order by release_date desc, creation_date desc
```

If the stuff for a particular clause won't fit on one line, put a newline immediately after the keyword that opens the clause. Then indent the items underneath. Here's an example from the ArsDigita Community System's static .html page administration section. We're querying the `static_pages` table, which holds a copy of any .html files in the Unix file system:

```
select
```

```
  to_char(count(*),'999G999G999G999G999') as n_pages,
  to_char(sum(dbms_lob.getlength(page_body)),'999G999G999G999G999') as n_bytes
from static_pages;
```

In this query, there are two items in the select list, a count of all the rows and a sum of the bytes in the `page_body` column (of type CLOB, hence the requirement to use `dbms_lob.getlength` rather than simply `length`). We want Oracle to format these numbers with separation characters between every three digits. For this, we have to use the `to_char` function and a mask of `'999G999G999G999G999'` (the "G" tells Oracle to use the appropriate character depending on the country where it is installed, e.g., comma in the U.S. and period in Europe). Then we have to give the results correlation names so that they will be easy to use as Tcl variables. By the time we're done with all of this, it would be confusing to put both items on the same line.

Here's another example, this time from the top-level comment administation page for the ArsDigita Community System. We're going to get back a single row with a count of each type of user-submitted comment:

```
select
  count(*) as n_total,
  sum(decode(comment_type,'alternative_perspective',1,0)) as
n_alternative_perspectives,
  sum(decode(comment_type,'rating',1,0)) as n_ratings,
  sum(decode(comment_type,'unanswered_question',1,0)) as n_unanswered_questions,
  sum(decode(comment_type,'private_message_to_page_authors',1,0)) as
n_private_messages
from comments
```

Notice the use of `sum(decode` to count the number of each type of comment. This gives us similar information to what we'd get from a GROUP BY, but we get a sum total as well as category totals. Also, the numbers come out with the column names of our choice. Of course, this kind of query only works when you know in advance the possible values of `comment_type`.

- The Oracle docs on DECODE
- for an explanation of the number formatting wizardry, see Oracle8 Server SQL Reference section on format conversion

Rules for GROUP BY queries

When you're doing a GROUP BY, put the columns that determine the group identity first in the select list. Put the aggregate columns that compute a function for that group afterwards:

```
select links.user_id, first_names, last_name, count(links.page_id) as n_links
from links, users
where links.user_id = users.user_id
group by links.user_id, first_names, last_name
order by n_links desc
```

Next: procedural

---

philg@mit.edu

Reader's Comments

The where clause is in two lines in the example above though the text suggests one line:
"If it doesn't fit nicely on one line, give each clause a separate line:
select *
from news
where sysdate > expiration_date
and approved_p = 't'
order by release_date desc, creation_date desc"
In this case, one line would be the better, of course.

-- Peter Tury, June 12, 2002

Escaping to the procedural world

part of SQL for Web Nerds by Philip Greenspun

Declarative languages can be very powerful and reliable, but sometimes it is easier to think about things procedurally. One way to do this is by using a procedural language in the database client. For example, with AOLserver we generally program in Tcl, a procedural language, and read the results of SQL queries. For example, in the /news module of the ArsDigita Community System, we want to

- query for the current news
- loop through the rows that come back and display one line for each row (with a link to a page that will show the full story)
- for the first three rows, see if the news story is very short. If so, just display it on this page

The words above that should give a SQL programmer pause are in the last bullet item: *if* and *for the first three rows*. There are no clean ways in standard SQL to say "do this just for the first N rows" or "do something special for a particular row if its data match a certain pattern".

Here's the AOLserver Tcl program. Note that depending on the contents of an item in the `news` table, the Tcl program may execute an SQL query (to figure out if there are user comments on a short news item).

```
set selection [ns_db select $db "select *
from news
where sysdate between release_date and expiration_date
and approved_p = 't'
order by release_date desc, creation_date desc"]
```

```
while { [ns_db getrow $db $selection] } {
    set_variables_after_query
    # we use the luxury of Tcl to format the date nicely
    ns_write "<li>[util_AnsiDatetoPrettyDate $release_date]: "
    if { $counter <= 3 && [string length $body] < 300 } {
        # it is one of the top three items and it is rather short
        # so, let's consider displaying it right here
        # first, let's go back to Oracle to find out if there are any
        # comments on this item
        set n_comments [database_to_tcl_string $db_sub "select count(*) from
general_comments where on_what_id = $news_id and on_which_table = 'news'"]
        if { $n_comments > 0 } {
            # there are some comments; just show the title
            ns_write "<a href=\"item.tcl?news_id=$news_id\">$title</a>\n"
        } else {
            # let's show the whole news item
            ns_write "$title\n<blockquote>\n[util_maybe_convert_to_html $body
$html_p]\n"
            if [ad_parameter SolicitCommentsP news 1] {
                ns_write "<br><br>\n<A HREF=\"comment-
add.tcl?news_id=$news_id\">comment</a>\n"
            }
            ns_write "</blockquote>\n"
        }
    } else {
        ns_write "<a href=\"item.tcl?news_id=$news_id\">$title</a>\n"
    }
}
```

Suppose that you have a million rows in your news table, you want five, but you can only figure out which five with a bit of procedural logic. Does it really make sense to drag those million rows of data all the way across the network from the database server to your client application and then throw out 999,995 rows?

Or suppose that you're querying a million-row table and want the results back in a strange order. Does it make sense to build a million-row data structure in your client application, sort them in the client program, then return the sorted rows to the user?

Visit http://www.scorecard.org/chemical-profiles/ and search for "benzene". Note that there are 328 chemicals whose names contain the string "benzene":

```
select count(*)
from chemical
where upper(edf_chem_name) like upper('%benzene%');

  COUNT(*)
----------
       328
```

The way we want to display them is

- exact matches on top

- line break
- chemicals that start with the query string
- line break
- chemicals that contain the query string

Within each category of chemicals, we want to sort alphabetically. However, if there are numbers or special characters in front of a chemical name, we want to ignore those for the purposes of sorting.

Can you do all of that with one query? And have them come back from the database in the desired order?

You could if you could make a procedure that would run inside the database. For each row, the procedure would compute a score reflecting goodness of match. To get the order correct, you need only ORDER BY this score. To get the line breaks right, you need only have your application program watch for changes in score. For the fine tuning of sorting equally scored matches alphabetically, just write another procedure that will return a chemical name stripped of leading special characters, then sort by the result. Here's how it looks:

```
select edf_chem_name,
       edf_substance_id,
       score_chem_name_match_score(upper(edf_chem_name),upper('%benzene%'))
          as match_score
from chemical
where upper(edf_chem_name) like upper('%benzene%');
order by score_chem_name_match_score(upper(edf_chem_name),upper('benzene')),
         score_chem_name_for_sorting(edf_chem_name)
```

We specify the procedure `score_chem_name_match_score` to take two arguments: one the chemical name from the current row, and one the query string from the user. It returns 0 for an exact match, 1 for a chemical whose name begins with the query string, and 2 in all other cases (remember that this is only used in queries where a LIKE clause ensures that every chemical name at least contains the query string. Once we defined this procedure, we'd be able to call it from a SQL query, the same way that we can call built-in SQL functions such as `upper`.

So is this possible? Yes, in all "enterprise-class" relational database management systems. Historically, each DBMS has had a proprietary language for these *stored procedures*. Starting in 1997, DBMS companies began to put Java byte-code interpreters into the database server. Oracle added Java-in-the-server capability with its 8.1 release in February 1999. If you're looking at old systems such as Scorecard, though, you'll be looking at procedures in Oracle's venerable PL/SQL language:

```
create or replace function score_chem_name_match_score
 (chem_name IN varchar, query_string IN varchar)
return integer
AS
BEGIN
  IF chem_name = query_string THEN
     return 0;
  ELSIF instr(chem_name,query_string) = 1 THEN
```

```
     return 1;
  ELSE
     return 2;
  END IF;
END score_chem_name_match_score;
```

Notice that PL/SQL is a strongly typed language. We say what arguments we expect, whether they are IN or OUT, and what types they must be. We say that `score_chem_name_match_score` will return an integer. We can say that a PL/SQL variable should be of the same type as a column in a table:

```
create or replace function score_chem_name_for_sorting (chem_name IN varchar)
return varchar
AS
  stripped_chem_name chem_hazid_ref.edf_chem_name%TYPE;
BEGIN
  stripped_chem_name := ltrim(chem_name,'1234567890-+()[],'' #');
  return stripped_chem_name;
END score_chem_name_for_sorting;
```

The local variable `stripped_chem_name` is going to be the same type as the `edf_chem_name` column in the `chem_hazid_ref` table.

If you are using the Oracle application SQL*Plus to define PL/SQL functions, you have to terminate each definition with a line containing only the character "/". If SQL*Plus reports that there were errors in evaluating your definition, you then have to type "show errors" if you want more explanation. Unless you expect to write perfect code all the time, it can be convenient to leave these SQL*Plus incantations in your .sql files. Here's an example:

```
  -- note that we prefix the incoming arg with v_ to keep it

  -- distinguishable from the database column of the same name

  -- this is a common PL/SQL convention

  create or replace function user_group_name_from_id (v_group_id IN integer)

  return varchar

  IS

    -- instead of worrying about how many characters to

    -- allocate for this local variable, we just tell

    -- Oracle "make it the same type as the group_name

    -- column in the user_groups table"

    v_group_name    user_groups.group_name%TYPE;
```

```
BEGIN

  if v_group_id is null

     then return '';

  end if;

  -- note the usage of INTO below, which pulls a column

  -- from the table into a local variable

  select group_name into v_group_name

   from user_groups

   where group_id = v_group_id;

  return v_group_name;

END;

/

show errors
```

Choosing between PL/SQL and Java

How to choose between PL/SQL and Java? Easy: you don't get to choose. In lots of important places, e.g., triggers, Oracle forces you to specify blocks of PL/SQL. So you have to learn at least the rudiments of PL/SQL. If you're going to build major packages, Java is probably a better long-term choice.

Reference

- Overview: Oracle8 Server Application Developer's Guide, "Using Procedures and Packages" at http://www.oradoc.com/keyword/using_procedures_and_packages
- PL/SQL User's Guide and Reference at http://www.oradoc.com/keyword/plsql
- Java Stored Procedures Developer's Guide at http://www.oradoc.com/keyword/java_stored_procedures

Trees in Oracle SQL

part of SQL for Web Nerds by Philip Greenspun

On its face, the relational database management system would appear to be a very poor tool for representing and manipulating trees. This chapter is designed to accomplish the following things:

- show you that a row in an SQL database can be thought of as an object
- show you that a pointer from one object to another can be represented by storing an integer key in a regular database column
- demonstrate the Oracle tree extensions (CONNECT BY ... PRIOR)
- show you how to work around the limitations of CONNECT BY with PL/SQL

The canonical example of trees in Oracle is the org chart.

```
create table corporate_slaves (
       slave_id              integer primary key,
       supervisor_id         references corporate_slaves,
       name                  varchar(100)
);

insert into corporate_slaves values (1, NULL, 'Big Boss Man');
insert into corporate_slaves values (2, 1, 'VP Marketing');
insert into corporate_slaves values (3, 1, 'VP Sales');
insert into corporate_slaves values (4, 3, 'Joe Sales Guy');
insert into corporate_slaves values (5, 4, 'Bill Sales Assistant');
```

```
insert into corporate_slaves values (6, 1, 'VP Engineering');
insert into corporate_slaves values (7, 6, 'Jane Nerd');
insert into corporate_slaves values (8, 6, 'Bob Nerd');

SQL> column name format a20
SQL> select * from corporate_slaves;

  SLAVE_ID SUPERVISOR_ID NAME
---------- ------------- --------------------
         1               Big Boss Man
         2             1 VP Marketing
         3             1 VP Sales
         4             3 Joe Sales Guy
         6             1 VP Engineering
         7             6 Jane Nerd
         8             6 Bob Nerd
         5             4 Bill Sales Assistant

8 rows selected.
```

The integers in the `supervisor_id` are actually pointers to other rows in the `corporate_slaves` table. Need to display an org chart? With only standard SQL available, you'd write a program in the client language (e.g., C, Lisp, Perl, or Tcl) to do the following:

1. query Oracle to find the employee `where supervisor_id is null`, call this `$big_kahuna_id`
2. query Oracle to find those employees whose `supervisor_id = $big_kahuna_id`
3. for each subordinate, query Oracle again to find their subordinates.
4. repeat until no subordinates found, then back up one level

With the Oracle CONNECT BY clause, you can get all the rows out at once:

```
select name, slave_id, supervisor_id
from corporate_slaves
connect by prior slave_id = supervisor_id;

NAME                 SLAVE_ID SUPERVISOR_ID
-------------------- ---------- -------------
Big Boss Man                  1
VP Marketing                  2             1
VP Sales                      3             1
Joe Sales Guy                 4             3
Bill Sales Assistant          5             4
VP Engineering                6             1
Jane Nerd                     7             6
Bob Nerd                      8             6

VP Marketing                  2             1

VP Sales                      3             1
Joe Sales Guy                 4             3
Bill Sales Assistant          5             4

Joe Sales Guy                 4             3
```

```
Bill Sales Assistant         5            4

VP Engineering               6            1
Jane Nerd                    7            6
Bob Nerd                     8            6

Jane Nerd                    7            6

Bob Nerd                     8            6

Bill Sales Assistant         5            4

20 rows selected.
```

This seems a little strange. It looks as though Oracle has produced all possible trees and subtrees. Let's add a START WITH clause:

```
select name, slave_id, supervisor_id
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id in (select slave_id
                        from corporate_slaves
                        where supervisor_id is null);

NAME                 SLAVE_ID SUPERVISOR_ID
-------------------- ---------- -------------
Big Boss Man                 1
VP Marketing                 2            1
VP Sales                     3            1
Joe Sales Guy                4            3
Bill Sales Assistant         5            4
VP Engineering               6            1
Jane Nerd                    7            6
Bob Nerd                     8            6

8 rows selected.
```

Notice that we've used a subquery in the START WITH clause to find out who is/are the big kahuna(s). For the rest of this example, we'll just hard-code in the `slave_id` 1 for brevity.

Though these folks are in the correct order, it is kind of tough to tell from the preceding report who works for whom. Oracle provides a magic pseudo-column that is meaningful only when a query includes a CONNECT BY. The pseudo-column is `level`:

```
select name, slave_id, supervisor_id, level
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id = 1;

NAME                 SLAVE_ID SUPERVISOR_ID     LEVEL
-------------------- ---------- ------------- ----------
Big Boss Man                 1                        1
VP Marketing                 2            1           2
VP Sales                     3            1           2
Joe Sales Guy                4            3           3
```

```
Bill Sales Assistant         5            4          4
VP Engineering               6            1          2
Jane Nerd                    7            6          3
Bob Nerd                     8            6          3

8 rows selected.
```

The `level` column can be used for indentation. Here we will use the concatenation operator (`||`) to add spaces in front of the name column:

```
column padded_name format a30

select
  lpad(' ', (level - 1) * 2) || name as padded_name,
  slave_id,
  supervisor_id,
  level
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id = 1;

PADDED_NAME                    SLAVE_ID SUPERVISOR_ID      LEVEL
------------------------------ ---------- ------------- ----------
Big Boss Man                            1                       1
  VP Marketing                          2             1         2
  VP Sales                              3             1         2
    Joe Sales Guy                       4             3         3
      Bill Sales Assistant              5             4         4
  VP Engineering                        6             1         2
    Jane Nerd                           7             6         3
    Bob Nerd                            8             6         3

8 rows selected.
```

If you want to limit your report, you can use standard WHERE clauses:

```
select
  lpad(' ', (level - 1) * 2) || name as padded_name,
  slave_id,
  supervisor_id,
  level
from corporate_slaves
where level <= 3
connect by prior slave_id = supervisor_id
start with slave_id = 1;

PADDED_NAME                    SLAVE_ID SUPERVISOR_ID      LEVEL
------------------------------ ---------- ------------- ----------
Big Boss Man                            1                       1
  VP Marketing                          2             1         2
  VP Sales                              3             1         2
    Joe Sales Guy                       4             3         3
  VP Engineering                        6             1         2
    Jane Nerd                           7             6         3
    Bob Nerd                            8             6         3
```

```
7 rows selected.
```

Suppose that you want people at the same level to sort alphabetically. Sadly, the ORDER BY clause doesn't work so great in conjunction with CONNECT BY:

```
select
  lpad(' ', (level - 1) * 2) || name as padded_name,
  slave_id,
  supervisor_id,
  level
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id = 1
order by level, name;

PADDED_NAME                    SLAVE_ID SUPERVISOR_ID      LEVEL
------------------------------ ---------- ------------- ----------
Big Boss Man                            1                        1
  VP Engineering                        6             1          2
  VP Marketing                          2             1          2
  VP Sales                              3             1          2
    Bob Nerd                            8             6          3
    Jane Nerd                           7             6          3
    Joe Sales Guy                       4             3          3
      Bill Sales Assistant              5             4          4

select
  lpad(' ', (level - 1) * 2) || name as padded_name,
  slave_id,
  supervisor_id,
  level
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id = 1
order by name;

PADDED_NAME                    SLAVE_ID SUPERVISOR_ID      LEVEL
------------------------------ ---------- ------------- ----------
Big Boss Man                            1                        1
      Bill Sales Assistant              5             4          4
    Bob Nerd                            8             6          3
    Jane Nerd                           7             6          3
    Joe Sales Guy                       4             3          3
  VP Engineering                        6             1          2
  VP Marketing                          2             1          2
  VP Sales                              3             1          2
```

SQL is a set-oriented language. In the result of a CONNECT BY query, it is precisely the order that has value. Thus it doesn't make much sense to also have an ORDER BY clause.

JOIN doesn't work with CONNECT BY

If we try to build a report showing each employee and his or her supervisor's name, we are treated to one of Oracle's few informative error messages:

```
select
  lpad(' ', (level - 1) * 2) || cs1.name as padded_name,
  cs2.name as supervisor_name
from corporate_slaves cs1, corporate_slaves cs2
where cs1.supervisor_id = cs2.slave_id(+)
connect by prior cs1.slave_id = cs1.supervisor_id
start with cs1.slave_id = 1;

ERROR at line 4:
ORA-01437: cannot have join with CONNECT BY
```

We can work around this particular problem by creating a view:

```
create or replace view connected_slaves
as
select
  lpad(' ', (level - 1) * 2) || name as padded_name,
  slave_id,
  supervisor_id,
  level as the_level
from corporate_slaves
connect by prior slave_id = supervisor_id
start with slave_id = 1;
```

Notice that we've had to rename `level` so that we didn't end up with a view column named after a reserved word. The view works just like the raw query:

```
select * from connected_slaves;

PADDED_NAME                    SLAVE_ID SUPERVISOR_ID  THE_LEVEL
------------------------------ ---------- ------------- ----------
Big Boss Man                            1                       1
  VP Marketing                          2             1         2
  VP Sales                              3             1         2
    Joe Sales Guy                       4             3         3
      Bill Sales Assistant              5             4         4
  VP Engineering                        6             1         2
    Jane Nerd                           7             6         3
    Bob Nerd                            8             6         3

8 rows selected.
```

but we can JOIN now

```
select padded_name, corporate_slaves.name as supervisor_name
from connected_slaves, corporate_slaves
where connected_slaves.supervisor_id = corporate_slaves.slave_id(+);

PADDED_NAME                    SUPERVISOR_NAME
------------------------------ --------------------
Big Boss Man
  VP Marketing                 Big Boss Man
  VP Sales                     Big Boss Man
    Joe Sales Guy              VP Sales
      Bill Sales Assistant      Joe Sales Guy
  VP Engineering               Big Boss Man
```

```
    Jane Nerd                 VP Engineering
    Bob Nerd                  VP Engineering

8 rows selected.
```

If you have sharp eyes, you'll notice that we've actually OUTER JOINed so that our results don't exclude the big boss.

Select-list subqueries *do* work with CONNECT BY

Instead of the VIEW and JOIN, we could have added a subquery to the select list:

```
select
  lpad(' ', (level - 1) * 2) || name as padded_name,
  (select name
   from corporate_slaves cs2
   where cs2.slave_id = cs1.supervisor_id) as supervisor_name
from corporate_slaves cs1
connect by prior slave_id = supervisor_id
start with slave_id = 1;

PADDED_NAME                   SUPERVISOR_NAME
----------------------------- --------------------
Big Boss Man
  VP Marketing                Big Boss Man
  VP Sales                    Big Boss Man
    Joe Sales Guy             VP Sales
      Bill Sales Assistant     Joe Sales Guy
  VP Engineering              Big Boss Man
    Jane Nerd                 VP Engineering
    Bob Nerd                  VP Engineering

8 rows selected.
```

The general rule in Oracle is that you can have a subquery that returns a single row anywhere in the select list.

Does this person work for me?

Suppose that you've built an intranet Web service. There are things that your software should show to an employee's boss (or boss's boss) that it shouldn't show to a subordinate or peer. Here we try to figure out if the VP Marketing (#2) has supervisory authority over Jane Nerd (#7):

```
select count(*)
from corporate_slaves
where slave_id = 7
and level > 1
start with slave_id = 2
connect by prior slave_id = supervisor_id;

  COUNT(*)
----------
         0
```

Apparently not. Notice that we start with the VP Marketing (#2) and stipulate `level > 1` to be sure that we will never conclude that someone supervises him or herself. Let's ask if the Big Boss Man (#1) has authority over Jane Nerd:

```
select count(*)
from corporate_slaves
where slave_id = 7
and level > 1
start with slave_id = 1
connect by prior slave_id = supervisor_id;

  COUNT(*)
----------
         1
```

Even though Big Boss Man isn't Jane Nerd's direct supervisor, asking Oracle to compute the relevant subtree yields us the correct result. In the ArsDigita Community System Intranet module, we decided that this computation was too important to be left as a query in individual Web pages. We centralized the question in a PL/SQL procedure:

```
create or replace function intranet_supervises_p
  (query_supervisor IN integer, query_user_id IN integer)
return varchar
is
  n_rows_found integer;
BEGIN
  select count(*) into n_rows_found
   from intranet_users
   where user_id = query_user_id
   and level > 1
   start with user_id = query_supervisor
   connect by supervisor = PRIOR user_id;
  if n_rows_found > 0 then
        return 't';
  else
        return 'f';
  end if;
END intranet_supervises_p;
```

Family trees

What if the graph is a little more complicated than employee-supervisor? For example, suppose that you are representing a family tree. Even without allowing for divorce and remarriage, exotic South African fertility clinics, etc., we still need more than one pointer for each node:

```
create table family_relatives (
        relative_id     integer primary key,
        spouse          references family_relatives,
        mother          references family_relatives,
        father          references family_relatives,
        -- in case they don't know the exact birthdate
        birthyear       integer,
        birthday        date,
        -- sadly, not everyone is still with us
```

```
        deathyear       integer,
        first_names     varchar(100) not null,
        last_name       varchar(100) not null,
        sex             char(1) check (sex in ('m','f')),
        -- note the use of multi-column check constraints
        check ( birthyear is not null or birthday is not null)
);

-- some test data


insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(1, 'Nick', 'Gittes', 'm', NULL, NULL, NULL, 1902);

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(2, 'Cecile', 'Kaplan', 'f', 1, NULL, NULL, 1910);

update family_relatives
set spouse = 2
where relative_id = 1;

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(3, 'Regina', 'Gittes', 'f', NULL, 2, 1, 1934);

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(4, 'Marjorie', 'Gittes', 'f', NULL, 2, 1, 1936);

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(5, 'Shirley', 'Greenspun', 'f', NULL, NULL, NULL, 1901);

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(6, 'Jack', 'Greenspun', 'm', 5, NULL, NULL, 1900);

update family_relatives
set spouse = 6
where relative_id = 5;

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(7, 'Nathaniel', 'Greenspun', 'm', 3, 5, 6, 1930);
```

```
update family_relatives
set spouse = 7
where relative_id = 3;

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(8, 'Suzanne', 'Greenspun', 'f', NULL, 3, 7, 1961);

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(9, 'Philip', 'Greenspun', 'm', NULL, 3, 7, 1963);

insert into family_relatives
(relative_id, first_names, last_name, sex, spouse, mother, father, birthyear)
values
(10, 'Harry', 'Greenspun', 'm', NULL, 3, 7, 1965);
```

In applying the lessons from the employee examples, the most obvious problem that we face now is whether to follow the mother or the father pointers:

```
column full_name format a25

-- follow patrilineal (start with my mom's father)
select lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name
from family_relatives
connect by prior relative_id = father
start with relative_id = 1;

FULL_NAME
-------------------------
Nick Gittes
  Regina Gittes
  Marjorie Gittes

-- follow matrilineal (start with my mom's mother)
select lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name
from family_relatives
connect by prior relative_id = mother
start with relative_id = 2;

FULL_NAME
-------------------------
Cecile Kaplan
  Regina Gittes
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
  Marjorie Gittes
```

Here's what the official Oracle docs have to say about CONNECT BY:

specifies the relationship between parent rows and child rows of the hierarchy. condition can be any condition as described in "Conditions". However, some part of the condition must use the PRIOR

operator to refer to the parent row. The part of the condition containing the PRIOR operator must have one of the following forms:

```
PRIOR expr comparison_operator expr
expr comparison_operator PRIOR expr
```

There is nothing that says `comparison_operator` has to be merely the equals sign. Let's start again with my mom's father but CONNECT BY more than one column:

```
-- follow both
select lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name
from family_relatives
connect by prior relative_id in (mother, father)
start with relative_id = 1;

FULL_NAME
-------------------------
Nick Gittes
  Regina Gittes
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
  Marjorie Gittes
```

Instead of arbitrarily starting with Grandpa Nick, let's ask Oracle to show us all the trees that start with a person whose parents are unknown:

```
select lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name
from family_relatives
connect by prior relative_id in (mother, father)
start with relative_id in (select relative_id from family_relatives
                           where mother is null
                           and father is null);

FULL_NAME
-------------------------
Nick Gittes
  Regina Gittes
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
  Marjorie Gittes
Cecile Kaplan
  Regina Gittes
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
  Marjorie Gittes
Shirley Greenspun
  Nathaniel Greenspun
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
Jack Greenspun
  Nathaniel Greenspun
    Suzanne Greenspun
```

```
    Philip Greenspun
    Harry Greenspun

22 rows selected.
```

PL/SQL instead of JOIN

The preceding report is interesting but confusing because it is hard to tell where the trees meet in marriage. As noted above, you can't do a JOIN with a CONNECT BY. We demonstrated the workaround of burying the CONNECT BY in a view. A more general workaround is using PL/SQL:

```
create or replace function family_spouse_name
  (v_relative_id family_relatives.relative_id%TYPE)
return varchar
is
  v_spouse_id integer;
  spouse_name varchar(500);
BEGIN
  select spouse into v_spouse_id
    from family_relatives
    where relative_id = v_relative_id;
  if v_spouse_id is null then
    return null;
  else
    select (first_names || ' ' || last_name) into spouse_name
      from family_relatives
      where relative_id = v_spouse_id;
    return spouse_name;
  end if;
END family_spouse_name;
/
show errors

column spouse format a20

select
  lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name,
  family_spouse_name(relative_id) as spouse
from family_relatives
connect by prior relative_id in (mother, father)
start with relative_id in (select relative_id from family_relatives
                           where mother is null
                           and father is null);

FULL_NAME                SPOUSE
------------------------ --------------------
Nick Gittes              Cecile Kaplan
  Regina Gittes          Nathaniel Greenspun
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
  Marjorie Gittes
Cecile Kaplan            Nick Gittes
  Regina Gittes          Nathaniel Greenspun
```

```
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
  Marjorie Gittes
Shirley Greenspun         Jack Greenspun
  Nathaniel Greenspun     Regina Gittes
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
Jack Greenspun            Shirley Greenspun
  Nathaniel Greenspun     Regina Gittes
    Suzanne Greenspun
    Philip Greenspun
    Harry Greenspun
```

PL/SQL instead of JOIN and GROUP BY

Suppose that in addition to displaying the family tree in a Web page, we also want to show a flag when a story about a family member is available. First we need a way to represent stories:

```
create table family_stories (
        family_story_id        integer primary key,
        story                  clob not null,
        item_date              date,
        item_year              integer,
        access_control         varchar(20)
             check (access_control in ('public', 'family', 'designated')),
        check (item_date is not null or item_year is not null)
);

-- a story might be about more than one person
create table family_story_relative_map (
        family_story_id        references family_stories,
        relative_id            references family_relatives,
        primary key (relative_id, family_story_id)
);

-- put in a test story
insert into family_stories
(family_story_id, story, item_year, access_control)
values
(1, 'After we were born, our parents stuck the Wedgwood in a cabinet
and bought indestructible china.  Philip and his father were sitting at
the breakfast table one morning.  Suzanne came downstairs and, without
saying a word, took a cereal bowl from the cupboard, walked over to
Philip and broke the bowl over his head.  Their father immediately
started laughing hysterically.', 1971, 'public');

insert into family_story_relative_map
(family_story_id, relative_id)
values
(1, 8);

insert into family_story_relative_map
```

```
(family_story_id, relative_id)
values
(1, 9);

insert into family_story_relative_map
(family_story_id, relative_id)
values
(1, 7);
```

To show the number of stories alongside a family member's listing, we would typically do an OUTER JOIN and then GROUP BY the columns other than the `count(family_story_id)`. In order not to disturb the CONNECT BY, however, we create another PL/SQL function:

```
create or replace function family_n_stories (v_relative_id
family_relatives.relative_id%TYPE)
return integer
is
  n_stories integer;
BEGIN
  select count(*) into n_stories
    from family_story_relative_map
    where relative_id = v_relative_id;
  return n_stories;
END family_n_stories;
/
show errors

select
  lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name,
  family_n_stories(relative_id) as n_stories
from family_relatives
connect by prior relative_id in (mother, father)
start with relative_id in (select relative_id from family_relatives
                           where mother is null
                           and father is null);

FULL_NAME                 N_STORIES
------------------------- ----------
Nick Gittes                        0
...
Shirley Greenspun                  0
  Nathaniel Greenspun              1
    Suzanne Greenspun              1
    Philip Greenspun               1
    Harry Greenspun                0
...
```

Working Backwards

What does it look like to start at the youngest generation and work back?

```
select
  lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name,
  family_spouse_name(relative_id) as spouse
from family_relatives
connect by relative_id in (prior mother, prior father)
start with relative_id = 9;

FULL_NAME                 SPOUSE
------------------------- --------------------
Philip Greenspun
  Regina Gittes           Nathaniel Greenspun
    Nick Gittes           Cecile Kaplan
    Cecile Kaplan         Nick Gittes
  Nathaniel Greenspun     Regina Gittes
    Shirley Greenspun     Jack Greenspun
    Jack Greenspun        Shirley Greenspun
```

We ought to be able to view all the trees starting from all the leaves but Oracle seems to be exhibiting strange behavior:

```
select
  lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name,
  family_spouse_name(relative_id) as spouse
from family_relatives
connect by relative_id in (prior mother, prior father)
start with relative_id not in (select mother from family_relatives
                               union
                               select father from family_relatives);

no rows selected
```

What's wrong? If we try the subquery by itself, we get a reasonable result. Here are all the `relative_id`s that appear in the `mother` or `father` column at least once.

```
select mother from family_relatives
union
select father from family_relatives

    MOTHER
----------
         1
         2
         3
         5
         6
         7


7 rows selected.
```

The answer lies in that extra blank line at the bottom. There is a NULL in this result set. Experimentation reveals that Oracle behaves asymmetrically with NULLs and IN and NOT IN:

```
SQL> select * from dual where 1 in (1,2,3,NULL);

D
```

```
-
X

SQL> select * from dual where 1 not in (2,3,NULL);

no rows selected
```

The answer is buried in the Oracle documentation of NOT IN: "Evaluates to FALSE if any member of the set is NULL." The correct query in this case?

```
select
  lpad(' ', (level - 1) * 2) || first_names || ' ' || last_name as full_name,
  family_spouse_name(relative_id) as spouse
from family_relatives
connect by relative_id in (prior mother, prior father)
start with relative_id not in (select mother
                               from family_relatives
                               where mother is not null
                               union
                               select father
                               from family_relatives
                               where father is not null);

FULL_NAME                 SPOUSE
------------------------- --------------------
Marjorie Gittes
  Nick Gittes             Cecile Kaplan
  Cecile Kaplan           Nick Gittes
Suzanne Greenspun
  Regina Gittes           Nathaniel Greenspun
    Nick Gittes           Cecile Kaplan
    Cecile Kaplan         Nick Gittes
  Nathaniel Greenspun     Regina Gittes
    Shirley Greenspun     Jack Greenspun
    Jack Greenspun        Shirley Greenspun
Philip Greenspun
  Regina Gittes           Nathaniel Greenspun
    Nick Gittes           Cecile Kaplan
    Cecile Kaplan         Nick Gittes
  Nathaniel Greenspun     Regina Gittes
    Shirley Greenspun     Jack Greenspun
    Jack Greenspun        Shirley Greenspun
Harry Greenspun
  Regina Gittes           Nathaniel Greenspun
    Nick Gittes           Cecile Kaplan
    Cecile Kaplan         Nick Gittes
  Nathaniel Greenspun     Regina Gittes
    Shirley Greenspun     Jack Greenspun
    Jack Greenspun        Shirley Greenspun

24 rows selected.
```

Performance and Tuning

Oracle is not getting any help from the Tree Fairy in producing results from a CONNECT BY. If you don't want tree queries to take O(N^2) time, you need to build indices that let Oracle very quickly answer questions of the form "What are all the children of Parent X?"
For the corporate slaves table, you'd want two concatenated indices:

```
create index corporate_slaves_idx1
  on corporate_slaves (slave_id, supervisor_id);
create index corporate_slaves_idx2
  on corporate_slaves (supervisor_id, slave_id);
```

Reference

- SQL Reference section on CONNECT BY
- PL/SQL User's Guide and Reference

Gratuitous Photos

Next: dates

---

philg@mit.edu

Related Links

- representing an m-ary tree in sql- This method allows for very fast retrieval of descendants and modification of an m-ary tree. no self-referencing or nested select statements are necessary to retrieve some or all descendants. the labelling of nodes is such that it allows very simple and fast querying for DFS order of nodes. it was partially inspired by huffman encoding.
  (contributed by Anthony D'Auria)

Dates

part of SQL for Web Nerds by Philip Greenspun, updated June 13, 2003

When representing date-time information in Oracle it becomes absolutely critical to know with which version of the Oracle server you are working. From version 9 onwards it is possible to represent points in time and time intervals using ANSI SQL data types such as `timestamp` and `interval`. Earlier versions of Oracle represented points in time with the `date` datatype, which is precise to within one second, and time intervals as numbers (where 1 = one day).

We strongly recommend that you use the newly available ANSI data types when building new applications. These are cleaner and more powerful than the older Oracle-specific way of doing things and further will make it easier to port your application to another RDBMS if necessary.

If you are stuck using an older version of Oracle or are writing queries and transactions to an older data model, please visit http://philip.greenspun.com/sql/dates-pre-9.

Querying by Date

Suppose that we have the following table to record user registrations:

```
create table users (
        user_id                 integer primary key,
        first_names             varchar(50),
        last_name               varchar(50) not null,
        email                   varchar(100) not null unique,
        -- we encrypt passwords using operating system crypt function
        password                varchar(30) not null,
        -- we only need precision to within one second
        registration_date       timestamp(0)
);

-- add some sample data
insert into users
(user_id, first_names, last_name, email, password, registration_date)
values
(1,'schlomo','mendelowitz','schlomo@mendelowitz.com','67xui2',
to_timestamp('2003-06-13 09:15:00','YYYY-MM-DD HH24:MI:SS'));

insert into users
(user_id, first_names, last_name, email, password, registration_date)
values
(2,'George Herbert Walker','Bush','former-president@whitehouse.gov','kl88q',
to_timestamp('2003-06-13 15:18:22','YYYY-MM-DD HH24:MI:SS'));
```

Let's query for people who registered during the last day:

```
column email format a35
column registration_date format a25

select email, registration_date
from users
where registration_date > current_date - interval '1' day;

EMAIL                               REGISTRATION_DATE
----------------------------------- -------------------------
```

```
schlomo@mendelowitz.com             13-JUN-03 09.15.00 AM
former-president@whitehouse.gov     13-JUN-03 03.18.22 PM
```

Note how the registration date comes out in a non-standard format that won't sort lexicographically and that does not have a full four digits for the year. You should curse your database administrator at this point for not configuring Oracle with a more sensible default. You can fix the problem for yourself right now, however:

```
alter session
set nls_timestamp_format =
'YYYY-MM-DD HH24:MI:SS';

select email, registration_date
from users
where registration_date > current_date - interval '1' day;

EMAIL                               REGISTRATION_DATE
----------------------------------- ---------------------
schlomo@mendelowitz.com             2003-06-13 09:15:00
former-president@whitehouse.gov     2003-06-13 15:18:22
```

You can query for shorter time intervals:

```
select email, registration_date
from users
where registration_date > current_date - interval '1' hour;

EMAIL                               REGISTRATION_DATE
----------------------------------- -------------------------
former-president@whitehouse.gov     2003-06-13 15:18:22

select email, registration_date
from users
where registration_date > current_date - interval '1' minute;

no rows selected

select email, registration_date
from users
where registration_date > current_date - interval '1' second;

no rows selected
```

You can be explicit about how you'd like the timestamps formatted:

```
select email, to_char(registration_date,'Day, Month DD, YYYY') as reg_day
from users
order by registration_date;

EMAIL                               REG_DAY
----------------------------------- ------------------------------
schlomo@mendelowitz.com             Friday   , June      13, 2003
former-president@whitehouse.gov     Friday   , June      13, 2003
```

Oops. Oracle pads some of these fields by default so that reports will be lined up and neat. We'll have to trim the strings ourselves:

```
select
  email,
  trim(to_char(registration_date,'Day')) || ', ' ||
   trim(to_char(registration_date,'Month')) || ' ' ||
   trim(to_char(registration_date,'DD, YYYY')) as reg_day
from users
order by registration_date;

EMAIL                              REG_DAY
---------------------------------- ----------------------------
schlomo@mendelowitz.com            Friday, June 13, 2003
former-president@whitehouse.gov    Friday, June 13, 2003
```

Some Very Weird Things

One reason that Oracle may have resisted ANSI date-time datatypes and arithmetic is that they can make life very strange for the programmer.

```
alter session set nls_date_format = 'YYYY-MM-DD';

-- old
select add_months(to_date('2003-07-31','YYYY-MM-DD'),-1) from dual;

ADD_MONTHS
----------
2003-06-30

-- new
select to_timestamp('2003-07-31','YYYY-MM-DD') - interval '1' month from dual;

ERROR at line 1:
ORA-01839: date not valid for month specified

-- old
select to_date('2003-07-31','YYYY-MM-DD') - 100 from dual;

TO_DATE('2
----------
2003-04-22

-- new (broken)
select to_timestamp('2003-07-31','YYYY-MM-DD') - interval '100' day from dual;

ERROR at line 1:
ORA-01873: the leading precision of the interval is too small

-- new (note the extra "(3)")
select to_timestamp('2003-07-31','YYYY-MM-DD') - interval '100' day(3) from dual;

TO_TIMESTAMP('2003-07-31','YYYY-MM-DD')-INTERVAL'100'DAY(3)
---------------------------------------------------------
2003-04-22 00:00:00
```

Some Profoundly Painful Things

Calculating time intervals between rows in a table can be very painful because there is no way in standard SQL to refer to "the value of this column from the previous row in the report". You can do this easily enough in an imperative computer language, e.g., C#, Java, or Visual Basic, that is reading rows from an SQL database but doing it purely in SQL is tough.

Let's add a few more rows to our users table to see how this works.

```
insert into users
(user_id, first_names, last_name, email, password, registration_date)
values
(3,'Osama','bin Laden','50kids@aol.com','dieusa',
to_timestamp('2003-06-13 17:56:03','YYYY-MM-DD HH24:MI:SS'));

insert into users
(user_id, first_names, last_name, email, password, registration_date)
values
(4,'Saddam','Hussein','livinlarge@saudi-online.net','wmd34',
to_timestamp('2003-06-13 19:12:43','YYYY-MM-DD HH24:MI:SS'));
```

Suppose that we're interested in the average length of time between registrations. With so few rows we could just query all the data out and eyeball it:

```
select registration_date
from users
order by registration_date;

REGISTRATION_DATE
-------------------------
2003-06-13 09:15:00
2003-06-13 15:18:22
2003-06-13 17:56:03
2003-06-13 19:12:43
```

If we have a lot of data, however, we'll need to do a self-join.

```
column r1 format a21
column r2 format a21

select
  u1.registration_date as r1,
  u2.registration_date as r2
from users u1, users u2
where u2.user_id = (select min(user_id) from users
                    where registration_date > u1.registration_date)
order by r1;

R1                    R2
--------------------- ---------------------
2003-06-13 09:15:00   2003-06-13 15:18:22
2003-06-13 15:18:22   2003-06-13 17:56:03
2003-06-13 17:56:03   2003-06-13 19:12:43
```

Notice that to find the "next row" for the pairing we are using the `user_id` column, which we know to be sequential and unique, rather than the registration_date column, which may not be unique because two users could register at exactly the same time.

Now that we have information from adjacent rows paired up in the same report we can begin to calculate intervals:

```
column reg_gap format a21

select
  u1.registration_date as r1,
  u2.registration_date as r2,
  u2.registration_date-u1.registration_date as reg_gap
from users u1, users u2
where u2.user_id = (select min(user_id) from users
                    where registration_date > u1.registration_date)
order by r1;

R1                    R2                    REG_GAP
--------------------- --------------------- ---------------------
2003-06-13 09:15:00   2003-06-13 15:18:22   +000000000 06:03:22
2003-06-13 15:18:22   2003-06-13 17:56:03   +000000000 02:37:41
2003-06-13 17:56:03   2003-06-13 19:12:43   +000000000 01:16:40
```

The interval for each row of the report has come back as days, hours, minutes, and seconds. At this point you'd expect to be able to average the intervals:

```
select avg(reg_gap)
from
(select
  u1.registration_date as r1,
  u2.registration_date as r2,
  u2.registration_date-u1.registration_date as reg_gap
from users u1, users u2
where u2.user_id = (select min(user_id) from users
                    where registration_date > u1.registration_date))

ERROR at line 1:
ORA-00932: inconsistent datatypes: expected NUMBER got INTERVAL
```

Oops. Oracle isn't smart enough to aggregate time intervals. And sadly there doesn't seem to be an easy way to turn a time interval into a number of seconds, for example, that would be amenable to averaging. If you figure how out to do it, please let me know!

Should we give up? If you have a strong stomach you can convert the timestamps to old-style Oracle dates through character strings before creating the intervals. This will give us a result as a fraction of a day:

```
select avg(reg_gap)
from
(select
  u1.registration_date as r1,
  u2.registration_date as r2,
```

```
  to_date(to_char(u2.registration_date,'YYYY-MM-DD HH24:MI:SS'),'YYYY-MM-DD
HH24:MI:SS')
   - to_date(to_char(u1.registration_date,'YYYY-MM-DD HH24:MI:SS'),'YYYY-MM-DD
HH24:MI:SS')
   as reg_gap
from users u1, users u2
where u2.user_id = (select min(user_id) from users
                    where registration_date > u1.registration_date))

AVG(REG_GAP)
------------
   .13836034
```

If we're going to continue using this ugly query we ought to create a view:

```
create view registration_intervals
as
select
  u1.registration_date as r1,
  u2.registration_date as r2,
  to_date(to_char(u2.registration_date,'YYYY-MM-DD HH24:MI:SS'),'YYYY-MM-DD
HH24:MI:SS')
   - to_date(to_char(u1.registration_date,'YYYY-MM-DD HH24:MI:SS'),'YYYY-MM-DD
HH24:MI:SS')
   as reg_gap
from users u1, users u2
where u2.user_id = (select min(user_id) from users
                    where registration_date > u1.registration_date)
```

Now we can calculate the average time interval in minutes:

```
select 24*60*avg(reg_gap) as avg_gap_minutes from registration_intervals;

AVG_GAP_MINUTES
---------------
     199.238889
```

Reporting

Here's an example of using the `to_char` function an GROUP BY to generate a report of sales by calendar quarter:

```
select to_char(shipped_date,'YYYY') as shipped_year,
       to_char(shipped_date,'Q') as shipped_quarter,
       sum(price_charged) as revenue
from sh_orders_reportable
where product_id = 143
and shipped_date is not null
group by to_char(shipped_date,'YYYY'), to_char(shipped_date,'Q')
order by to_char(shipped_date,'YYYY'), to_char(shipped_date,'Q');

SHIPPED_YEAR         SHIPPED_QUARTER        REVENUE
-------------------- -------------------- ----------
1998                 2                        1280
1998                 3                        1150
1998                 4                         350
```

```
1999                    1                              210
```

This is a hint that Oracle has all kinds of fancy date formats (covered in their online documentation). We're using the "Q" mask to get the calendar quarter. We can see that this product started shipping in Q2 1998 and that revenues trailed off in Q4 1998.

More

- "New Datatypes, New Possibilities" by Steven Feuerstein

Limits

part of SQL for Web Nerds by Philip Greenspun

The most painful limit in Oracle is the 4000-byte maximum length of a VARCHAR. For most Web applications, this turns out to be long enough to hold 97% of user-entered data. The remaining 3% are sufficiently important that you might be unable to use VARCHAR.

Oracle8 includes a Character Large Objects (CLOB) data type, which can be up to 4 GB in size.

```
create table foobar
( mykey integer,
  moby clob );

insert into foobar values ( 1, 'foo');
```

The first time I used CLOBs was for an Oracle 8 port of my Q&A forum software (what you see running at photo.net). I use it for the MESSAGE column. In my Illustra implementation, it turned out that only 46 out of the 12,000 rows in the table had messages longer than 4000 bytes (the VARCHAR limit in Oracle 8). But those 46 messages were very interesting and sometimes contained quoted material from other works. So it seemed desirable to allow users to post extremely long messages if they want.

Minor Caveats:

- CLOBs don't work like strings. You can't ask for the LENGTH of a CLOB column, for example. You can work around this with PL/SQL calls but it isn't much fun.

- LOBs are not allowed in GROUP BY, ORDER BY, SELECT DISTINCT, aggregates and JOINS
- `alter table bboard modify message clob;` does not work (even with no rows in the table). If you want to use CLOBs, you apparently have to say so at table creation time.

If you thought that these limitations were bad, you haven't gotten to the big one: the Oracle SQL parser can only handle string literals up to 4000 characters in length. SQL*Plus is even more restricted. It can handle strings up to 2500 characters in length. From most Oracle clients, there will in fact be no way to insert data longer than 4000 characters long into a table. A statement/system that works perfectly with a 4000-character string will fail completely with a 4001-character string. You have to completely redesign the way you're doing things to work with strings that *might* be long.

My partner Cotton was custom writing me an Oracle 8 driver for AOLserver (my preferred RDBMS client). So he decided to use C bind variables instead. It turned out that these didn't work either for strings longer than 4000 chars. There is some special CLOB type for C that you could use if you knew in advance that the column was CLOB. But of course Cotton's C code was just taking queries from my AOLserver Tcl code. Without querying the database before every INSERT or SELECT, he had no way of knowing which columns were going to be CLOB.

One of the most heavily touted features of Oracle 8 is that you can partition tables, e.g., say "every row with an order_date column less than January 1, 1997 goes in tablespace A; every row with order_date less than January 1, 1998 goes in tablespace B; all the rest go in tablespace C." Assuming that these tablespaces are all on separate disk drives, this means that when you do a GROUP BY on orders during the last month or two, Oracle isn't sifting through many years worth of data; it only has to scan the partition of the table that resides in tablespace C. Partitioning seems to be a good idea, or at least the Informix customers who've been using it over the years seem to think so. But if you are a CLOB Achiever, you won't be using partitioning.

Right now, my take on CLOBs is that they are so unpleasant to use as to be almost not worth it. It has taken Cotton longer to get this one feature working than everything else he did with his driver. Informix Universal Server lets you have 32,000-character long VARCHARs and I'm looking longingly at that system right now...

Reference

- Oracle Server Reference, "Limits", http://www.oradoc.com/keyword/limits

Indexing and Tuning

part of SQL for Web Nerds by Philip Greenspun

One of the great dividends of investing in an RDBMS is that you don't have to think too much about the computer's inner life. You're the programmer and say what kinds of data you want. The computer's job is to fetch it and you don't really care how.

Maybe you'll start caring after that $500,000 database server has been grinding away on one of your queries for two solid hours...

While software is being developed, it is rare for tables to contain more than a handful of rows. Nobody wants to sit in SQL*Plus or at Web forms and type in test data. After the application launches and tables begin to fill up, people eventually notice that a particular section of the site is slow. Here are the steps that you must take

1. Find a URL that is running too slowly.
2. If possible, enable query logging from your Web or application server. What you want is for the Web server to write every SQL query and transaction into a single file so that you can see exactly what the database management system was told to do and when. This the kind of feature that makes a Web programming environment truly productive that it is tough to advertise it to the Chief Technology Officer types who select Web programming environment (i.e., if you're stuck using some closed-source Web connectivity middleware/junkware you might not be able to do this).

   With AOLserver, enable query logging by setting `Verbose=On` in the `[ns/db/pool/**poolname**]` section of your .ini file. The queries will show up in the error log ("/home/nsadmin/log/server.log" by default).

3. Request the problematic URL from a Web browser.
4. fire up Emacs and load the query log into a buffer; spawn a shell and run sqlplus from the shell, logging in with the same username/password as used by the Web server
5. you can now cut (from server.log) and paste (into sqlplus) the queries performed by the script backing the slow URL. However, first you must turn on tracing so that you can see what Oracle is doing.

```
6.  SQL> set autotrace on
7.  Unable to verify PLAN_TABLE format or existence
8.  Error enabling EXPLAIN report
```

Oops! It turns out that Oracle is unhappy about just writing to standard output. For each user that wants to trace queries, you need to feed sqlplus the file $ORACLE_HOME/rdbms/admin/utlxplan.sql which contains a single table definition:

```
create table PLAN_TABLE (
  statement_id   varchar2(30),
  timestamp      date,
  remarks        varchar2(80),
  operation      varchar2(30),
  options        varchar2(30),
  object_node    varchar2(128),
  object_owner   varchar2(30),
  object_name    varchar2(30),
  object_instance numeric,
  object_type     varchar2(30),
  optimizer       varchar2(255),
  search_columns  number,
  id             numeric,
  parent_id      numeric,
  position       numeric,
  cost           numeric,
  cardinality    numeric,
  bytes          numeric,
  other_tag       varchar2(255),
  partition_start varchar2(255),
        partition_stop  varchar2(255),
        partition_id    numeric,
  other          long);
```

9. Type "set autotrace on" again (it should work now; if you get an error about the PLUSTRACE role then tell your dbadmin to run $ORACLE_HOME/sqlplus/admin/plustrce.sql as SYS then GRANT your user that role).
10. Type "set timing on" (you'll get reports of elapsed time)
11. cut and paste the query of interest.

Now that we're all set up, let's look at a few examples.

A simple B-Tree Index

Suppose that we want to ask "Show me the users who've requested a page within the last few minutes". This can support a nice "Who's online now?" page, like what you see at http://photo.net/shared/whos-online. Here's the source code to find users who've requested a page within the last 10 minutes (600 seconds):

```
select user_id, first_names, last_name, email
from users
where last_visit > sysdate - 600/86400
order by upper(last_name), upper(first_names), upper(email)
```

We're querying the users table:

```
create table users (
```

```
        user_id                 integer primary key,
        first_names             varchar(100) not null,
        last_name               varchar(100) not null,
        ...
        email                   varchar(100) not null unique,
        ...
        -- set when user reappears at site
        last_visit              date,
        -- this is what most pages query against (since the above column
        -- will only be a few minutes old for most pages in a session)
        second_to_last_visit    date,
        ...
);
```

Suppose that we ask for information about User #37. Oracle need not scan the entire table because the declaration that `user_id` be the table's primary key implicitly causes an index to be constructed. The `last_visit` column, however, is not constrained to be unique and therefore Oracle will not build an index on its own. Searching for the most recent visitors at photo.net will require scanning all 60,000 rows in the `users` table. We can add a B-Tree index, for many years the only kind available in any database management system, with the following statement:

```
create index users_by_last_visit on users (last_visit);
```

Now Oracle can simply check the index first and find pointers to rows in the `users` table with small values of `last_visit`.

Tracing/Tuning Case 1: did we already insert the message?

The SQL here comes from an ancient version of the bulletin board system in the ArsDigita Community System (see http://photo.net/bboard/ for an example). In the bad old days when we were running the Illustra relational database management system, it took so long to do an INSERT that users would keep hitting "Reload" on their browsers. When they were all done, there were three copies of a message in the bulletin board. So we modified the insertion script to check the `bboard` table to see if there was already a message with exactly the same values in the `one_line` and `message` columns. Because `message` is a CLOB column, you can't just do the obvious "=" comparison and need to call the PL/SQL function `dbms_lob.instr`, part of Oracle's built-in DBMS_LOB package.

Here's a SQL*Plus session looking for an already-posted message with a subject line of "foo" and a body of "bar":

```
SQL> select count(*) from bboard
where topic = 'photo.net'
and one_line = 'foo'
and dbms_lob.instr(message,'bar') > 0 ;

  COUNT(*)
----------
         0


Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE
```

```
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'BBOARD'
   3    2       INDEX (RANGE SCAN) OF 'BBOARD_BY_TOPIC' (NON-UNIQUE)


Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
      59967  consistent gets
      10299  physical reads
          0  redo size
        570  bytes sent via SQL*Net to client
        741  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

Note the "10,299 physical reads". Disk drives are very slow. You don't really want to be doing more than a handful of physical reads. Let's look at the heart of the query plan:

```
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'BBOARD'
   3    2       INDEX (RANGE SCAN) OF 'BBOARD_BY_TOPIC' (NON-UNIQUE)
```

Looks as though Oracle is hitting the `bboard_by_topic` index for the ROWIDs of "just the rows that have a topic of 'photo.net'". It is then using the ROWID, an internal Oracle pointer, to pull the actual rows from the BBOARD table. Presumably Oracle will then count up just those rows where the ONE_LINE and MESSAGE columns are appropriate. This might not actually be so bad in an installation where there were 500 different discussion groups. Hitting the index would eliminate 499/500 rows. But BBOARD_BY_TOPIC isn't a very selective index. Let's investigate the selectivity with the query `select topic, count(*) from bboard group by topic order by count(*) desc`:

topic	count(*)
photo.net	14159
Nature Photography	3289
Medium Format Digest	1639
Ask Philip	91
web/db	62

The `bboard` table only has about 19,000 rows and the photo.net topic has 14,000 of them, about 75%. So the index didn't do us much good. In fact, you'd have expected Oracle not to use the index. A full table scan is generally faster than an index scan if more than 20% of the rows need be examined. Why didn't Oracle do the full table scan? Because the table hadn't been "analyzed". There were no statistics for the cost-based optimizer so the older rule-based optimizer was employed. You have to periodically tell Oracle to build statistics on tables if you want the fancy cost-based optimizer:

```
SQL> analyze table bboard compute statistics;

Table analyzed.

SQL> select count(*) from bboard
where topic = 'photo.net'
```

```
and one_line = 'foo'
and dbms_lob.instr(message,'bar') > 0 ;

  COUNT(*)
----------
         0

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1808 Card=1 Bytes=828)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (FULL) OF 'BBOARD' (Cost=1808 Card=1 Bytes=828)

Statistics
----------------------------------------------------------
          0  recursive calls
          4  db block gets
      74280  consistent gets
      12266  physical reads
          0  redo size
        572  bytes sent via SQL*Net to client
        741  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

The final numbers don't look much better. But at least the cost-based optimizer has figured out that the topic index won't be worth much. Now we're just scanning the full `bboard` table. While transferring 20,000 rows from Illustra to Oracle during a photo.net upgrade, we'd not created any indices. This speeded up loading but then we were so happy to have the system running deadlock-free that we forgot to recreate an index that we'd been using on the Illustra system expressly for the purpose of making this query fast.

```
SQL> create index bboard_index_by_one_line on bboard ( one_line );

Index created.
```

Bboard postings are now indexed by subject line, which should be a very selective column because it is unlikely that many users would choose to give their question the same title. This particular query will be faster now but inserts and updates will be slower. Why? Every INSERT or UPDATE will have to update the `bboard` table blocks on the hard drive and also the `bboard_index_by_one_line` blocks, to make sure that the index always has up-to-date information on what is in the table. If we have multiple physical disk drives we can instruct Oracle to keep the index in a separate tablespace, which the database administrator has placed on a separate disk:

```
SQL> drop index bboard_index_by_one_line;

SQL> create index bboard_index_by_one_line
     on bboard ( one_line )
     tablespace philgidx;

Index created.
```

Now the index will be kept in a different tablespace (`philgidx`) from the main table. During inserts and updates, data will be written on two separate disk drives in parallel. Let's try the query again:

```
SQL> select count(*) from bboard
where topic = 'photo.net'
and one_line = 'foo'
and dbms_lob.instr(message,'bar') > 0 ;

  COUNT(*)
----------
         0

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=828)
   1    0   SORT (AGGREGATE)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'BBOARD' (Cost=2 Card=1 Bytes=828)
   3    2       INDEX (RANGE SCAN) OF 'BBOARD_INDEX_BY_ONE_LINE' (NON-UNIQUE)
(Cost=1 Card=1)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
          3  consistent gets
          3  physical reads
          0  redo size
        573  bytes sent via SQL*Net to client
        741  bytes received via SQL*Net from client
          4  SQL*Net roundtrips to/from client
          1  sorts (memory)
          0  sorts (disk)
          1  rows processed
```

We've brought physical reads down from 12266 to 3. Oracle is checking the index on `one_line` and then poking at the main table using the ROWIDs retrieved from the index. It might actually be better to build a concatenated index on two columns: the user ID of the person posting and the subject line, but at this point you might make the engineering decision that 3 physical reads is acceptable.

Tracing/Tuning Case 2: new questions

At the top of each forum page, e.g., http://photo.net/bboard/q-and-a.tcl?topic=photo.net, the ArsDigita Community System shows questions asked in the last few days (configurable, but the default is 7 days). After the forum filled up with 30,000 messages, this page was perceptibly slow.

```
SQL> select msg_id, one_line, sort_key, email, name
from bboard
where topic = 'photo.net'
and refers_to is null
and posting_time > (sysdate - 7)
order by sort_key desc;

...

61 rows selected.

Execution Plan
```

```
-------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=1828 Card=33 Bytes=27324)

   1    0   SORT (ORDER BY) (Cost=1828 Card=33 Bytes=27324)
   2    1     TABLE ACCESS (FULL) OF 'BBOARD' (Cost=1808 Card=33 Bytes=27324)


Statistics
-------------------------------------------------------
          0  recursive calls
          4  db block gets
      13188  consistent gets
      12071  physical reads
          0  redo size
       7369  bytes sent via SQL*Net to client
       1234  bytes received via SQL*Net from client
          8  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         61  rows processed
```

A full table scan and 12,071 physical reads just to get 61 rows! It was time to get medieval on this query. Since the query's WHERE clause contains topic, refers_to, and posting_time, the obvious thing to try is building a concatenated index on all three columns:

```
SQL> create index bboard_for_new_questions
     on bboard ( topic, refers_to, posting_time )
     tablespace philgidx;

Index created.

SQL> select msg_id, one_line, sort_key, email, name
from bboard
where topic = 'photo.net'
and refers_to is null
and posting_time > (sysdate - 7)
order by sort_key desc;

...

61 rows selected.

Execution Plan
-------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=23 Card=33 Bytes=27324)

   1    0   SORT (ORDER BY) (Cost=23 Card=33 Bytes=27324)
   2    1     TABLE ACCESS (BY INDEX ROWID) OF 'BBOARD' (Cost=3 Card=33
Bytes=27324)
   3    2       INDEX (RANGE SCAN) OF 'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE)
(Cost=2 Card=33)

Statistics
-------------------------------------------------------
          0  recursive calls
          0  db block gets
```

```
        66  consistent gets
        60  physical reads
         0  redo size
      7369  bytes sent via SQL*Net to client
      1234  bytes received via SQL*Net from client
         8  SQL*Net roundtrips to/from client
         2  sorts (memory)
         0  sorts (disk)
        61  rows processed
```

60 reads is better than 12,000. One bit of clean-up, though. There is no reason to have a BBOARD_BY_TOPIC index if we are going to keep this BBOARD_FOR_NEW_QUESTIONS index, whose first column is TOPIC. The query optimizer can use BBOARD_FOR_NEW_QUESTIONS even when the SQL only restricts based on the TOPIC column. The redundant index won't cause any services to fail, but it will slow down inserts.

```
SQL> drop index bboard_by_topic;

Index dropped.
```

We were so pleased with ourselves that we decided to drop an index on `bboard` by the `refers_to` column, reasoning that nobody ever queries `refers_to` without also querying on `topic`. Therefore they could just use the first two columns in the `bboard_for_new_questions` index. Here's a query looking for unanswered questions:

```
SQL> select msg_id, one_line, sort_key, email, name
from bboard bbd1
where topic = 'photo.net'
and 0 = (select count(*) from bboard bbd2 where bbd2.refers_to = bbd1.msg_id)
and refers_to is null
order by sort_key desc;

...

57 rows selected.

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=49 Card=33 Bytes=27324)

   1    0   SORT (ORDER BY) (Cost=49 Card=33 Bytes=27324)
   2    1     FILTER
   3    2       TABLE ACCESS (BY INDEX ROWID) OF 'BBOARD' (Cost=29 Card=33
Bytes=27324)
   4    3         INDEX (RANGE SCAN) OF 'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE)
(Cost=2 Card=33)
   5    2       INDEX (FULL SCAN) OF 'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE)
(Cost=26 Card=7 Bytes=56)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
     589843  consistent gets
     497938  physical reads
```

```
          0  redo size
       6923  bytes sent via SQL*Net to client
       1173  bytes received via SQL*Net from client
          7  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

Ouch! 497,938 physical reads. Let's try it with the index in place:

```
SQL> create index bboard_index_by_refers_to
     on bboard ( refers_to )
     tablespace philgidx;

Index created.

SQL> select msg_id, one_line, sort_key, email, name
from bboard bbd1
where topic = 'photo.net'
and 0 = (select count(*) from bboard bbd2 where bbd2.refers_to = bbd1.msg_id)
and refers_to is null
order by sort_key desc;

...

57 rows selected.

Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=49 Card=33 Bytes=27324)
   1    0   SORT (ORDER BY) (Cost=49 Card=33 Bytes=27324)
   2    1     FILTER
   3    2       TABLE ACCESS (BY INDEX ROWID) OF 'BBOARD' (Cost=29 Card=33
Bytes=27324)
   4    3         INDEX (RANGE SCAN) OF 'BBOARD_FOR_NEW_QUESTIONS' (NON-UNIQUE)
(Cost=2 Card=33)
   5    2       INDEX (RANGE SCAN) OF 'BBOARD_INDEX_BY_REFERS_TO' (NON-UNIQUE)
(Cost=1 Card=7 Bytes=56)

Statistics
----------------------------------------------------------
          0  recursive calls
          0  db block gets
       8752  consistent gets
       2233  physical reads
          0  redo size
       6926  bytes sent via SQL*Net to client
       1173  bytes received via SQL*Net from client
          7  SQL*Net roundtrips to/from client
          2  sorts (memory)
          0  sorts (disk)
         57  rows processed
```

This is still a fairly expensive query, but 200 times faster than before and it executes in a fraction of a second. That's probably fast enough considering that this is an infrequently requested page.

Tracing/Tuning Case 3: forcing Oracle to cache a full table scan

You may have a Web site that is basically giving users access to a huge table. For maximum flexibility, it might be the case that this table needs to be sequentially scanned for every query. In general, Oracle won't cache blocks retrieved during a full table scan. The Oracle tuning guide helpfully suggests that you include the following cache hints in your SQL:

```
select /*+ FULL (students) CACHE(students) */ count(*) from students;
```

You will find, however, that this doesn't work if your buffer cache (controlled by db_block_buffers; see above) isn't large enough to contain the table. Oracle is smart and ignores your hint. After you've reconfigured your Oracle installation to have a larger buffer cache, you'll probably find that Oracle is *still* ignoring your cache hint. That's because you also need to

```
analyze table students compute statistics;
```

and then Oracle will work as advertised in the tuning guide. It makes sense when you think about it because Oracle can't realistically start stuffing things into the cache unless it knows roughly how large the table is.

If it is still too slow

If your application is still too slow, you need to talk to the database administrator. If you *are* the database administrator as well as the programmer, you need to hire a database administrator ("dba").

A professional dba is great at finding queries that are pigs and building indices to make them faster. The dba might be able to suggest that you partion your tables so that infrequently used data are kept on a separate disk drive. The dba can make you extra tablespaces on separate physical disk drives. By moving partitions and indices to these separate disk drives, the dba can speed up your application by factors of 2 or 3.

A factor of 2 or 3? Sounds pretty good until you reflect on the fact that moving information from disk into RAM would speed things up by a factor of 100,000. This isn't really possible for database updates, which must be recorded in a durable medium (exception: fancy EMC disk arrays, which contain write caches and batteries to ensure durability of information in the write cache). However, it is relatively easy for queries. As a programmer, you can add indices and supply optimizer hints to increase the likelihood that your queries will be satisfied from Oracle's block cache. The dba can increase the amount of the server's RAM given over to Oracle. If that doesn't work, the dba can go out and order more RAM!

In 1999, Oracle running on a typical ArsDigita server gets 1 GB of RAM.

Reference

- Guy Harrison's *Oracle SQL High-Performance Tuning*
- Oracle8 Server Tuning



Data Warehousing

part of SQL for Web Nerds by Philip Greenspun

In the preceding chapters, you've been unwittingly immersed in the world of on-line transaction processing (OLTP). This world carries with it some assumptions:

1. Only store a piece of information once. If there are N copies of something in the database and you need to change it, you might forget to change it in all N places. Note that only storing information in one spot also enables updates to be fast.
2. It is okay if queries are complex because they are authored infrequently and by professional programmers.
3. Never sequentially scan large tables; reread the tuning chapter if Oracle takes more than one second to perform any operation.

These are wonderful rules to live by if one is booking orders, adding user comments to pages, recording a clickthrough, or seeing if someone is authorized to download a file.

You can probably continue to live by these rules if you want some answers from your data. Write down a list of questions that are important and build some report pages. You might need materialized views to make these reports fast and your queries might be complex, but you don't need to leave the OLTP world simply because business dictates that you answer a bunch of questions.

Why would anyone leave the OLTP world? Data warehousing is useful when you don't know what questions to ask.

What it means to facilitate exploration

Data exploration is only useful when non-techies are able to explore. That means people with very weak skills will be either authoring queries or specifying queries with menus. You can't ask a marketing executive to look at a 600-table data model and pick and choose the relevant columns. You can't ask a salesman to pull the answer to "is this a repeat customer or not?" out of a combination of the `customers` and `orders` tables.

If a data exploration environment is to be useful it must fulfill the following criteria:

- complex questions can be asked with a simple SQL query
- different questions imply very similar SQL query structure
- very different questions require very similar processing time to answer
- exploration can be done from any computer anywhere

The goal is that a business expert can sit down at a Web browser, use a sequence of forms to specify a query, and get a result back in an amount of time that seems reasonable.

It will be impossible to achieve this with our standard OLTP data models. Answering a particular question may require JOINing in four or five extra tables, which could result in a 10,000-fold increase in processing time. Even if a novice user could be guided to specifying a 7-way JOIN from among 600 tables, that person would have no way of understanding or predicting query processing time. Finally there is the question of whether you want novices querying your OLTP tables. If they are only typing SELECTs they might not be doing too much long-term harm but the short-term processing load might result in a system that feels crippled.

It is time to study *data warehousing*.

Classical Retail Data Warehousing

"Another segment of society that has constructed a language of its own is business. ... [The businessman] is speaking a language that is familiar to him and dear to him. Its portentous nouns and verbs invest ordinary events with high adventure; the executive walks among ink erasers caparisoned like a knight. This we should be tolerant of--every man of spirit wants to ride a white horse. ... A good many of the special words of business seem designed more to express the user's dreams than to express his precise meaning."
-- last chapter of *The Elements of Style*, Strunk and White

Let's imagine a conversation between the Chief Information Officer of WalMart and a sales guy from Sybase. We've picked these companies for concreteness but they stand for "big Management Information System (MIS) user" and "big relational database management system (RDBMS) vendor".

Walmart: "I want to keep track of sales in all of my stores simultaneously."

Sybase: "You need our wonderful RDBMS software. You can stuff data in as sales are rung up at cash registers and simultaneously query data out right here in your office. That's the beauty of concurrency control."

So Walmart buys a $1 million Sun E10000 multi-CPU server and a $500,000 Sybase license. They buy *Database Design for Smarties* and build themselves a normalized SQL data model:

```
create table product_categories (
        product_category_id    integer primary key,
        product_category_name  varchar(100) not null
);

create table manufacturers (
        manufacturer_id        integer primary key,
        manufacturer_name      varchar(100) not null
);

create table products (
        product_id             integer primary key,
        product_name           varchar(100) not null,
        product_category_id    references product_categories,
        manufacturer_id        references manufacturers
);
```

```
create table cities (
        city_id                 integer primary key,
        city_name               varchar(100) not null,
        state                   varchar(100) not null,
        population              integer not null
);

create table stores (
        store_id                integer primary key,
        city_id                 references cities,
        store_location          varchar(200) not null,
        phone_number            varchar(20)
);

create table sales (
        product_id      not null references products,
        store_id        not null references stores,
        quantity_sold   integer not null,
        -- the Oracle "date" type is precise to the second
        -- unlike the ANSI date datatype
        date_time_of_sale       date not null
);

-- put some data in

insert into product_categories values (1, 'toothpaste');
insert into product_categories values (2, 'soda');

insert into manufacturers values (68, 'Colgate');
insert into manufacturers values (5, 'Coca Cola');

insert into products values (567, 'Colgate Gel Pump 6.4 oz.', 1, 68);
insert into products values (219, 'Diet Coke 12 oz. can', 2, 5);

insert into cities values (34, 'San Francisco', 'California', 700000);
insert into cities values (58, 'East Fishkill', 'New York', 30000);

insert into stores values (16, 34, '510 Main Street', '415-555-1212');
insert into stores values (17, 58, '13 Maple Avenue', '914-555-1212');

insert into sales values (567, 17, 1, to_date('1997-10-22 09:35:14', 'YYYY-MM-DD
HH24:MI:SS'));
insert into sales values (219, 16, 4, to_date('1997-10-22 09:35:14', 'YYYY-MM-DD
HH24:MI:SS'));
insert into sales values (219, 17, 1, to_date('1997-10-22 09:35:17', 'YYYY-MM-DD
HH24:MI:SS'));

-- keep track of which dates are holidays
-- the presence of a date (all dates will be truncated to midnight)
-- in this table indicates that it is a holiday
create table holiday_map (
holiday_date            date primary key
);
```

```
-- where the prices are kept
create table product_prices (
product_id      not null references products,
from_date       date not null,
price           number not null
);

insert into product_prices values (567,'1997-01-01',2.75);
insert into product_prices values (219,'1997-01-01',0.40);
```

What do we have now?

SALES table

product id	store id	quantity sold	date/time of sale
567	17	1	1997-10-22 09:35:14
219	16	4	1997-10-22 09:35:14
219	17	1	1997-10-22 09:35:17

...

PRODUCTS table

product id	product name	product category	manufacturer id
567	Colgate Gel Pump 6.4 oz.	1	68
219	Diet Coke 12 oz. can	2	5

...

PRODUCT_CATEGORIES table

product category id	product category name
1	toothpaste
2	soda

...

MANUFACTURERS table

manufacturer id	manufacturer name
68	Colgate
5	Coca Cola

...

STORES table

store id	city id	store location	phone number
16	34	510 Main Street	415-555-1212
17	58	13 Maple Avenue	914-555-1212

...

CITIES table

city id	city name	state	population
34	San Francisco	California	700,000
58	East Fishkill	New York	30,000

...

After a few months of stuffing data into these tables, a WalMart executive, call her Jennifer Amolucre asks "I noticed that there was a Colgate promotion recently, directed at people who live in small towns. How much Colgate toothpaste did we sell in those towns yesterday? And how much on the same day a month ago?"

At this point, reflect that because the data model is normalized, this information can't be obtained from scanning one table. A normalized data model is one in which all the information in a row depends only on the primary key. For example, the city population is not contained in the `stores` table. That information is stored once per city in the `cities` table and only `city_id` is kept in the `stores` table. This ensures efficiency for transaction processing. If Walmart has to update a city's population, only one record on disk need be touched. As computers get faster, what is more interesting is the consistency of this approach. With the city population kept only in one place, there is no risk that updates will be applied to some records and not to others. If there are multiple stores in the same city, the population will be pulled out of the same slot for all the stores all the time.

Ms. Amolucre's query will look something like this...

```
select sum(sales.quantity_sold)
from sales, products, product_categories, manufacturers, stores, cities
where manufacturer_name = 'Colgate'
and product_category_name = 'toothpaste'
and cities.population < 40000
and trunc(sales.date_time_of_sale) = trunc(sysdate-1)  -- restrict to yesterday
and sales.product_id = products.product_id
and sales.store_id = stores.store_id
and products.product_category_id = product_categories.product_category_id
and products.manufacturer_id = manufacturers.manufacturer_id
and stores.city_id = cities.city_id;
```

This query would be tough for a novice to read and, being a 6-way JOIN of some fairly large tables, might take quite a while to execute. Moreover, these tables are being updated as Ms. Amolucre's query is executed.

Soon after the establishment of Jennifer Amolucre's quest for marketing information, store employees notice that there are times during the day when it is impossible to ring up customers. Any attempt to update the database results in the computer freezing up for 20 minutes. Eventually the database administrators realize that the system collapses every time Ms. Amolucre's toothpaste query gets run. They complain to Sybase tech support.

Walmart: "We type in the toothpaste query and our system wedges."
Sybase: "Of course it does! You built an on-line transaction processing (OLTP) system. You can't feed it a decision support system (DSS) query and expect things to work!"
Walmart: "But I thought the whole point of SQL and your RDBMS was that users could query and insert simultaneously."
Sybase: "Uh, not exactly. If you're reading from the database, nobody can write to the database. If you're writing to the database, nobody can read from the database. So if you've got a query that takes 20 minutes to run and don't specify special locking instructions, nobody can update those tables for 20 minutes."
Walmart: "That sounds like a bug."
Sybase: "Actually it is a feature. We call it *pessimistic locking*."
Walmart: "Can you fix your system so that it doesn't lock up?"
Sybase: "No. But we made this great loader tool so that you can copy everything from your OLTP system into a separate DSS system at 100 GB/hour."

Since you are reading this book, you are probably using Oracle, which is one of the few database management systems that achieves consistency among concurrent users via versioning rather than locking (the other notable example is the free open-source PostgreSQL RDBMS). However, even if you are using Oracle, where readers never wait for writers and writers never wait for readers, you still might not want the transaction processing operation to slow down in the event of a marketing person entering an expensive query.

Basically what IT vendors want Walmart to do is set up another RDBMS installation on a separate computer. Walmart needs to buy another $1 million of computer hardware. They need to buy another RDBMS license. They also need to hire programmers to make sure that the OLTP data is copied out nightly and stuffed into the DSS system--*data extraction*. Walmart is now building the *data warehouse*.

Insight 1

A data warehouse is a separate RDBMS installation that contains copies of data from on-line systems. A physically separate data warehouse is not absolutely necessary if you have a lot of extra computing horsepower. With a DBMS that uses optimistic locking you might even be able to get away with keeping only one copy of your data.

As long as we're copying...

As long as you're copying data from the OLTP system into the DSS system ("data warehouse"), you might as well think about organizing and indexing it for faster retrieval. Extra indices on production tables are bad because they slow down inserts and updates. Every time you add or modify a row to a table, the RDBMS has to update the indices to keep them consistent. But in a data warehouse, the data are static. You build indices once and they take up space and sometimes make queries faster and that's it.

If you know that Jennifer Amolucre is going to do the toothpaste query every day, you can denormalize the data model for her. If you add a `town_population` column to the `stores` table and copy in data from the `cities` table, for example, you sacrifice some cleanliness of data model but now Ms. Amolucre's query only requires a 5-way JOIN. If you add `manufacturer` and `product_category` columns to the `sales` table, you don't need to JOIN in the `products` table.

Where does denormalization end?

Once you give up the notion that the data model in the data warehouse need bear some resemblance to the data model in the OLTP system, you begin to think about reorganizing the data model further. Remember that we're trying to make sure that new questions can be asked by people with limited SQL experience, i.e., many different questions can be answered with morphologically similar SQL. Ideally the task of constructing SQL queries can be simplified enough to be doable from a menu system. Also, we are trying to delivery predictable response time. A minor change in a question should not result in a thousand-fold increase in system response time.

The irreducible problem with the OLTP data model is that it is tough for novices to construct queries. Given that computer systems are not infinitely fast, a practical problem is inevitably that the response times of a query into the OLTP tables will vary in a way that is unpredictable to the novice.

Suppose, for example, that Bill Novice wants to look at sales on holidays versus non-holidays with the OLTP model. Bill will need to go look at the data model, which on a production system will contain hundreds of tables, to find out if any of them contain information on whether or not a date is a holiday. Then he will need to use it in a query, something that isn't obvious given the peculiar nature of the Oracle `date` data type:

```
select sum(sales.quantity_sold)
from sales, holiday_map
where trunc(sales.date_time_of_sale) = trunc(holiday_map.holiday_date)
```

That one was pretty simple because JOINing to the `holiday_map` table knocks out sales on days that aren't holidays. To compare to sales on non-holidays, he will need to come up with a different query strategy, one that knocks out sales on days that *are* holidays. Here is one way:

```
select sum(sales.quantity_sold)
from sales
where trunc(sales.date_time_of_sale)
not in
(select holiday_date from holiday_map)
```

Note that the morphology (structure) of this query is completely different from the one asking for sales on holidays.

Suppose now that Bill is interested in unit sales just at those stores where the unit sales tended to be high overall. First Bill has to experiment to find a way to ask the database for the big-selling stores. Probably this will involve grouping the `sales` table by the `store_id` column:

```
select store_id
from sales
group by store_id
having sum(quantity_sold) > 1000
```

Now we know how to find stores that have sold more than 1000 units total, so we can add this as a subquery:

```
select sum(quantity_sold)
from sales
where store_id in
(select store_id
 from sales
 group by store_id
 having sum(quantity_sold) > 1000)
```

Morphologically this doesn't look very different from the preceding non-holiday query. Bill has had to figure out how to use the GROUP BY and HAVING constructs but otherwise it is a single table query with a subquery. Think about the time to execute, however. The `sales` table may contain millions of rows. The `holiday_map` table probably only contains 50 or 100 rows, depending on how long the OLTP system has been in place. The most obvious way to execute these subqueries will be to perform the subquery for each row examined by the main query. In the case of the "big stores" query, the subquery requires scanning and sorting the entire `sales` table. So the time to execute this query might be 10,000 times longer than the time to execute the "non-holiday sales" query. Should Bill Novice expect this behavior? Should he have to think about it? Should the OLTP system grind to a halt because he didn't think about it hard enough?

Virtually all the organizations that start by trying to increase similarity and predictability among decision support queries end up with a *dimensional data warehouse*. This necessitates a new data model that shares little with the OLTP data model.

Dimensional Data Modeling: First Steps

Dimensional data modeling starts with a *fact table*. This is where we record what happened, e.g., someone bought a Diet Coke in East Fishkill. What you want in the fact table are facts about the sale, ideally ones that are numeric, continuously valued, and additive. The last two properties are important because typical fact tables grow to a billion rows or more. People will be much happier looking at sums or averages than detail. An important decision to make is the granularity of the fact table. If Walmart doesn't care about whether or not a Diet Coke was sold at 10:31 AM or 10:33 AM, recording each sale individually in the fact table is too granular. CPU time, disk bandwidth, and disk space will be needlessly consumed. Let's aggregate all the sales of any particular product in one store on a per-day basis. So we will only have one row in the fact table recording that 200 cans of Diet Coke were sold in East Fishkill on November 30, even if those 200 cans were sold at 113 different times to 113 different customers.

```
create table sales_fact (
        sales_date      date not null,
        product_id      integer,
        store_id        integer,
```

```
        unit_sales      integer,
        dollar_sales    number
);
```

So far so good, we can pull together this table with a query JOINing the `sales`, `products`, and `product_prices` (to fill the `dollar_sales` column) tables. This JOIN will group by `product_id`, `store_id`, and the truncated `date_time_of_sale`. Constructing this query will require a professional programmer but keep in mind that this work only need be done once. The marketing experts who will be using the data warehouse will be querying from the `sales_fact` table.

In building just this one table, we've already made life easier for marketing. Suppose they want total dollar sales by product. In the OLTP data model this would have required tangling with the `product_prices` table and its different prices for the same product on different days. With the sales fact table, the query is simple:

```
select product_id, sum(dollar_sales)
from sales_fact
group by product_id
```

We have a *fact table*. In a dimensional data warehouse there will always be just one of these. All of the other tables will define the *dimensions*. Each dimension contains extra information about the facts, usually in a human-readable text string that can go directly into a report. For example, let us define the time dimension:

```
create table time_dimension (
        time_key                integer primary key,
        -- just to make it a little easier to work with; this is
        -- midnight (TRUNC) of the date in question
        oracle_date             date not null,
        day_of_week             varchar(9) not null, -- 'Monday', 'Tuesday'...
        day_number_in_month     integer not null, -- 1 to 31
        day_number_overall      integer not null, -- days from the epoch (first day
is 1)
        week_number_in_year     integer not null, -- 1 to 52
        week_number_overall     integer not null, -- weeks start on Sunday
        month                   integer not null, -- 1 to 12
        month_number_overall    integer not null,
        quarter                 integer not null, -- 1 to 4
        fiscal_period           varchar(10),
        holiday_flag            char(1) default 'f' check (holiday_flag in ('t',
'f')),
        weekday_flag            char(1) default 'f' check (weekday_flag in ('t',
'f')),
        season                  varchar(50),
        event                   varchar(50)
);
```

Why is it useful to define a time dimension? If we keep the date of the sales fact as an Oracle date column, it is still just about as painless as ever to ask for holiday versus non-holiday sales. We need to know about the existence of the `holiday_map` table and how to use it. Suppose we redefine the fact table as follows:

```
create table sales_fact (
        time_key        integer not null references time_dimension,
        product_id      integer,
        store_id        integer,
```

```
        unit_sales      integer,
        dollar_sales    number
);
```

Instead of storing an Oracle date in the fact table, we're keeping an integer key pointing to an entry in the time dimension. The time dimension stores, for each day, the following information:

- whether or not the day was a holiday
- into which fiscal period this day fell
- whether or not the day was part of the "Christmas season" or not

If we want a report of sales by season, the query is straightforward:

```
select td.season, sum(f.dollar_sales)
from sales_fact f, time_dimension td
where f.time_key = td.time_key
group by td.season
```

If we want to get a report of sales by fiscal quarter or sales by day of week, the SQL is structurally identical to the above. If we want to get a report of sales by manufacturer, however, we realize that we need another dimension: *product*. Instead of storing the `product_id` that references the OLTP `products` table, much better to use a synthetic product key that references a product dimension where data from the OLTP `products`, `product_categories`, and `manufacturers` tables are aggregated.

Since we are Walmart, a multi-store chain, we will want a *stores* dimension. This table will aggregate information from the `stores` and `cities` tables in the OLTP system. Here is how we would define the stores dimension in an Oracle table:

```
create table stores_dimension (
        stores_key              integer primary key,
        name                    varchar(100),
        city                    varchar(100),
        county                  varchar(100),
        state                   varchar(100),
        zip_code                varchar(100),
        date_opened             date,
        date_remodeled          date,
        -- 'small', 'medium', 'large', or 'super'
        store_size              varchar(100),
        ...
);
```

This new dimension gives us the opportunity to compare sales for large versus small stores, for new and old ones, and for stores in different regions. We can aggregate sales by geographical region, starting at the state level and drilling down to county, city, or ZIP code. Here is how we'd query for sales by city:

```
select sd.city, sum(f.dollar_sales)
from sales_fact f, stores_dimension sd
where f.stores_key = sd.stores_key
group by sd.city
```

Dimensions can be combined. To report sales by city on a quarter-by-quarter basis, we would use the following query:

```
select sd.city, td.fiscal_period, sum(f.dollar_sales)
from sales_fact f, stores_dimension sd, time_dimension td
where f.stores_key = sd.stores_key
and f.time_key = td.time_key
group by sd.stores_key, td.fiscal_period
```
(extra SQL compared to previous query shown in bold).

The final dimension in a generic Walmart-style data warehouse is *promotion*. The marketing folks will want to know how much a price reduction boosted sales, how much of that boost was permanent, and to what extent the promoted product cannibalized sales from other products sold at the same store. Columns in the promotion dimension table would include a promotion type (coupon or sale price), full information on advertising (type of ad, name of publication, type of publication), full information on in-store display, the cost of the promotion, etc.

At this point it is worth stepping back from the details to notice that the data warehouse contains less information than the OLTP system but it can be more useful in practice because queries are easier to construct and faster to execute. Most of the art of designing a good data warehouse is in defining the dimensions. Which aspects of the day-to-day business may be condensed and treated in blocks? Which aspects of the business are interesting?

Real World Example: A Data Warehouse for Levis Strauss

In 1998, ArsDigita Corporation built a Web service as a front end to an experimental custom clothing factory operated by Levi Strauss. Users would visit our site to choose a style of khaki pants, enter their waist, inseam, height, weight, and shoe size, and finally check out with their credit card. Our server would attempt to authorize a charge on the credit card through CyberCash. The factory IT system would poll our server's Oracle database periodically so that it could start cutting pants within 10 minutes of a successfully authorized order.

The whole purpose of the factory and Web service was to test and analyze consumer reaction to this method of buying clothing. Therefore, a data warehouse was built into the project almost from the start.

We did not buy any additional hardware or software to support the data warehouse. The public Web site was supported by a mid-range Hewlett-Packard Unix server that had ample leftover capacity to run the data warehouse. We created a new "dw" Oracle user, GRANTed SELECT on the OLTP tables to the "dw" user, and wrote procedures to copy all the data from the OLTP system into a star schema of tables owned by the "dw" user. For queries, we added an IP address to the machine and ran a Web server program bound to that second IP address.

Here is how we explained our engineering decisions to our customer (Levi Strauss):

```
We employ a standard star join schema for the following reasons:

* Many relational database management systems, including Oracle 8.1,
are heavily optimized to execute queries against these schemata.

* This kind of schema has been proven to scale to the world's
largest data warehouses.

* If we hired a data warehousing nerd off the street, he or she
```

would have no trouble understanding our schema.

In a star join schema, there is one fact table ("we sold a pair of khakis at 1:23 pm to Joe Smith") that references a bunch of dimension tables. As a general rule, if we're going to narrow our interest based on a column, it should be in the dimension table. I.e., if we're only looking at sales of grey dressy fabric khakis, we should expect to accomplish that with WHERE clauses on columns of a product dimension table. By contrast, if we're going to be aggregating information with a SUM or AVG command, these data should be stored in the columns of the fact table. For example, the dollar amount of the sale should be stored within the fact table. Since we have so few prices (essentially only one), you might think that this should go in a dimension. However, by keeping it in the fact table we're more consistent with traditional data warehouses.

After some discussions with Levi's executives, we designed in the following dimension tables:

- **time**
  for queries comparing sales by season, quarter, or holiday
- **product**
  for queries comparing sales by color or style
- **ship to**
  for queries comparing sales by region or state
- **promotion**
  for queries aimed at determining the relationship between discounts and sales
- **consumer**
  for queries comparing sales by first-time and repeat buyers
- **user experience**
  for queries looking at returned versus exchanged versus accepted items (most useful when combined with other dimensions, e.g., was a particular color more likely to lead to an exchange request)

These dimensions allow us to answer questions such as

- In what regions of the country are pleated pants most popular? (fact table joined with the product and ship-to dimensions)
- What percentage of pants were bought with coupons and how has that varied from quarter to quarter? (fact table joined with the promotion and time dimensions)
- How many pants were sold on holidays versus non-holidays? (fact table joined with the time dimension)

The Dimension Tables

The `time_dimension` table is identical to the example given above.

```
create table time_dimension (
        time_key                integer primary key,
        -- just to make it a little easier to work with; this is
        -- midnight (TRUNC) of the date in question
        oracle_date             date not null,
```

```
        day_of_week             varchar(9) not null, -- 'Monday', 'Tuesday'...
        day_number_in_month     integer not null, -- 1 to 31
        day_number_overall      integer not null, -- days from the epoch (first day
is 1)
        week_number_in_year     integer not null, -- 1 to 52
        week_number_overall     integer not null, -- weeks start on Sunday
        month                   integer not null, -- 1 to 12
        month_number_overall    integer not null,
        quarter                 integer not null, -- 1 to 4
        fiscal_period           varchar(10),
        holiday_flag            char(1) default 'f' check (holiday_flag in ('t',
'f')),
        weekday_flag            char(1) default 'f' check (weekday_flag in ('t',
'f')),
        season                  varchar(50),
        event                   varchar(50)
);
```

We populated the `time_dimension` table with a single INSERT statement. The core work is done by Oracle date formatting functions. A helper table, `integers`, is used to supply a series of numbers to add to a starting date (we picked July 1, 1998, a few days before our first real order).

```
-- Uses the integers table to drive the insertion, which just contains
-- a set of integers, from 0 to n.
-- The 'epoch' is hardcoded here as July 1, 1998.

-- d below is the Oracle date of the day we're inserting.
insert into time_dimension
(time_key, oracle_date, day_of_week, day_number_in_month,
 day_number_overall, week_number_in_year, week_number_overall,
 month, month_number_overall, quarter, weekday_flag)
select n, d, rtrim(to_char(d, 'Day')), to_char(d, 'DD'), n + 1,
       to_char(d, 'WW'),
       trunc((n + 3) / 7), -- July 1, 1998 was a Wednesday, so +3 to get the week
numbers to line up with the week
       to_char(d, 'MM'), trunc(months_between(d, '1998-07-01') + 1),
       to_char(d, 'Q'), decode(to_char(d, 'D'), '1', 'f', '7', 'f', 't')
from (select n, to_date('1998-07-01', 'YYYY-MM-DD') + n as d
      from integers);
```

Remember the Oracle date minutia that you learned in the chapter on dates. If you add a number to an Oracle date, you get another Oracle date. So adding 3 to "1998-07-01" will yield "1998-07-04".

There are several fields left to be populated that we cannot derive using Oracle date functions: season, fiscal period, holiday flag, season, event. Fiscal period depended on Levi's choice of fiscal year. The `event` column was set aside for arbitrary blocks of time that were particularly interesting to the Levi's marketing team, e.g., a sale period. In practice, it was not used.

To update the `holiday_flag` field, we used two helper tables, one for "fixed" holidays (those which occur on the same day each year), and one for "floating" holidays (those which move around).

```
create table fixed_holidays (
        month                   integer not null check (month >= 1 and month <= 12),
        day                     integer not null check (day >= 1 and day <= 31),
        name                    varchar(100) not null,
```

```
        primary key (month, day)
);

-- Specifies holidays that fall on the Nth DAY_OF_WEEK in MONTH.
-- Negative means count backwards from the end.
create table floating_holidays (
        month                   integer not null check (month >= 1 and month <= 12),
        day_of_week             varchar(9) not null,
        nth                     integer not null,
        name                    varchar(100) not null,
        primary key (month, day_of_week, nth)
);
```

Some example holidays:

```
insert into fixed_holidays (name, month, day)
   values ('New Year''s Day', 1, 1);
insert into fixed_holidays (name, month, day)
   values ('Christmas', 12, 25);
insert into fixed_holidays (name, month, day)
   values ('Veteran''s Day', 11, 11);
insert into fixed_holidays (name, month, day)
   values ('Independence Day', 7, 4);

insert into floating_holidays (month, day_of_week, nth, name)
   values (1, 'Monday', 3, 'Martin Luther King Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (10, 'Monday', 2, 'Columbus Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (11, 'Thursday', 4, 'Thanksgiving');
insert into floating_holidays (month, day_of_week, nth, name)
   values (2, 'Monday', 3, 'President''s Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (9, 'Monday', 1, 'Labor Day');
insert into floating_holidays (month, day_of_week, nth, name)
   values (5, 'Monday', -1, 'Memorial Day');
```

An extremely clever person who'd recently read *SQL for Smarties* would probably be able to come up with an SQL statement to update the `holiday_flag` in the `time_dimension` rows. However, there is no need to work your brain that hard. Recall that Oracle includes two procedural languages, Java and PL/SQL. You can implement the following pseudocode in the procedural language of your choice:

```
foreach row in "select name, month, day from fixed_holidays"
    update time_dimension
      set holiday_flag = 't'
      where month = row.month and day_number_in_month = row.day;
end foreach

foreach row in "select month, day_of_week, nth, name from floating_holidays"
    if row.nth > 0 then
        # If nth is positive, put together a date range constraint
        # to pick out the right week.
        ending_day_of_month := row.nth * 7
        starting_day_of_month := ending_day_of_month - 6

        update time_dimension
          set holiday_flag = 't'
```

```
          where month = row.month
            and day_of_week = row.day_of_week
            and starting_day_of_month <= day_number_in_month
            and day_number_in_month <= ending_day_of_month;
    else
        # If it is negative, get all the available dates
        # and get the nth one from the end.
        i := 0;
        foreach row2 in "select day_number_in_month from time_dimension
                         where month = row.month
                           and day_of_week = row.day_of_week
                         order by day_number_in_month desc"
            i := i - 1;
            if i = row.nth then
                update time_dimension
                  set holiday_flag = 't'
                  where month = row.month
                    and day_number_in_month = row2.day_number_in_month
                break;
            end if
        end foreach
    end if
end foreach
```

The product dimension

The product dimension contains one row for each unique combination of color, style, cuffs, pleats, etc.

```
create table product_dimension (
        product_key     integer primary key,
        -- right now this will always be "ikhakis"
        product_type    varchar(20) not null,
        -- could be "men", "women", "kids", "unisex adults"
        expected_consumers      varchar(20),
        color           varchar(20),
        -- "dressy" or "casual"
        fabric          varchar(20),
        -- "cuffed" or "hemmed" for pants
        -- null for stuff where it doesn't matter
        cuff_state      varchar(20),
        -- "pleated" or "plain front" for pants
        pleat_state     varchar(20)
);
```

To populate this dimension, we created a one-column table for each field in the dimension table and use a multi-table join without a WHERE clause. This generates the cartesian product of all the possible values for each field:

```
create table t1 (expected_consumers varchar(20));
create table t2 (color varchar(20));
create table t3 (fabric varchar(20));
create table t4 (cuff_state varchar(20));
create table t5 (pleat_state varchar(20));

insert into t1 values ('men');
insert into t1 values ('women');
insert into t1 values ('kids');
```

```
insert into t1 values ('unisex');
insert into t1 values ('adults');
[etc.]

insert into product_dimension
(product_key, product_type, expected_consumers,
color, fabric, cuff_state, pleat_state)
select
  product_key_sequence.nextval,
  'ikhakis',
  t1.expected_consumers,
  t2.color,
  t3.fabric,
  t4.cuff_state,
  t5.pleat_state
from t1,t2,t3,t4,t5;
```

Notice that an Oracle sequence, `product_key_sequence`, is used to generate unique integer keys for each row as it is inserted into the dimension.

The promotion dimension

The art of building the promotion dimension is dividing the world of coupons into a broad categories, e.g., "between 10 and 20 dollars". This categorization depended on the learning that the marketing executives did not care about the difference between a $3.50 and a $3.75 coupon.

```
create table promotion_dimension (
        promotion_key           integer primary key,
        -- can be "coupon" or "no coupon"
        coupon_state            varchar(20),
        -- a text string such as "under $10"
        coupon_range            varchar(20)
);
```

The separate `coupon_state` and `coupon_range` columns allow for reporting of sales figures broken down into fullprice/discounted or into a bunch of rows, one for each range of coupon size.

The consumer dimension

We did not have access to a lot of demographic data about our customers. We did not have a lot of history since this was a new service. Consequently, our consumer dimension is extremely simple. It is used to record whether or not a sale in the fact table was to a new or a repeat customer.

```
create table consumer_dimension (
        consumer_key            integer primary key,
        -- 'new customer' or 'repeat customer'
        repeat_class            varchar(20)
);
```

The user experience dimension

If we are interested in building a report of the average amount of time spent contemplating a purchase versus whether the purchase was ultimately kept, the `user_experience_dimension` table will help.

```
create table user_experience_dimension (
        user_experience_key     integer primary key,
        -- 'shipped on time', 'shipped late'
        on_time_status          varchar(20),
```

```
        -- 'kept', 'returned for exchange', 'returned for refund'
        returned_status         varchar(30)
);
```

The ship-to dimension

Classically one of the most powerful dimensions in a data warehouse, our `ship_to_dimension` table allows us to group sales by region or state.

```
create table ship_to_dimension (
        ship_to_key     integer primary key,
        -- e.g., Northeast
        ship_to_region  varchar(30) not null,
        ship_to_state   char(2) not null
);

create table state_regions (
        state           char(2) not null primary key,
        region          varchar(50) not null
);

-- to populate:
insert into ship_to_dimension
(ship_to_key, ship_to_region, ship_to_state)
select ship_to_key_sequence.nextval, region, state
from state_regions;
```

Notice that we've thrown out an awful lot of detail here. Had this been a full-scale product for Levi Strauss, they would probably have wanted at least extra columns for county, city, and zip code. These columns would allow a regional sales manager to look at sales within a state.

(In a data warehouse for a manufacturing wholesaler, the ship-to dimension would contain columns for the customer's company name, the division of the customer's company that received the items, the sales district of the salesperson who sold the order, etc.)

The Fact Table

The granularity of our fact table is one order. This is finer-grained than the canonical Walmart-style data warehouse as presented above, where a fact is the quantity of a particular SKU sold in one store on one day (i.e., all orders in one day for the same item are aggregated). We decided that we could afford this because the conventional wisdom in the data warehousing business in 1998 was that up to billion-row fact tables were manageable. Our retail price was $40 and it was tough to foresee a time when the factory could make more than 1,000 pants per day. So it did not seem extravagant to budget one row per order.

Given the experimental nature of this project we did not delude ourselves into thinking that we would get it right the first time. Since we were recording one row per order we were able to cheat by including pointers from the data warehouse back into the OLTP database: `order_id` and `consumer_id`. We never had to use these but it was nice to know that if we couldn't get a needed answer for the marketing executives the price would have been some custom SQL coding rather than rebuilding the entire data warehouse.

```
create table sales_fact (
        -- keys over to the OLTP production database
```

```
        order_id                integer primary key,
        consumer_id             integer not null,
        time_key                not null references time_dimension,
        product_key             not null references product_dimension,
        promotion_key           not null references promotion_dimension,
        consumer_key            not null references consumer_dimension,
        user_experience_key     not null references user_experience_dimension,
        ship_to_key             not null references ship_to_dimension,
        -- time stuff
        minutes_login_to_order          number,
        days_first_invite_to_order      number,
        days_order_to_shipment          number,
        -- this will be NULL normally (unless order was returned)
        days_shipment_to_intent         number,
        pants_id                integer,
        price_charged           number,
        tax_charged             number,
        shipping_charged        number
);
```

After defining the fact table, we populated it with a single insert statement:

```
-- find_product, find_promotion, find_consumer, and find_user_experience
-- are PL/SQL procedures that return the appropriate key from the dimension
-- tables for a given set of parameters

insert into sales_fact
 select o.order_id, o.consumer_id, td.time_key,
        find_product(o.color, o.casual_p, o.cuff_p, o.pleat_p),
        find_promotion(o.coupon_id),
        find_consumer(o.pants_id),
        find_user_experience(o.order_state, o.confirmed_date, o.shipped_date),
        std.ship_to_key,
        minutes_login_to_order(o.order_id, usom.user_session_id),
        decode(sign(o.confirmed_date - gt.issue_date), -1, null,
round(o.confirmed_date - gt.issue_date, 6)),
        round(o.shipped_date - o.confirmed_date, 6),
        round(o.intent_date - o.shipped_date, 6),
        o.pants_id, o.price_charged, o.tax_charged, o.shipping_charged
 from khaki.reportable_orders o, ship_to_dimension std,
      khaki.user_session_order_map usom, time_dimension td,
      khaki.addresses a, khaki.golden_tickets gt
 where o.shipping = a.address_id
        and std.ship_to_state = a.usps_abbrev
        and o.order_id = usom.order_id(+)
        and trunc(o.confirmed_date) = td.oracle_date
        and o.consumer_id = gt.consumer_id;
```

As noted in the comment at top, most of the work here is done by PL/SQL procedures such as `find_product` that dig up the right row in a dimension table for this particular order.

The preceding insert will load an empty data warehouse from the on-line transaction processing system's tables. Keeping the data warehouse up to date with what is happening in OLTP land requires a similar INSERT with an extra restriction WHERE clause limiting orders to only those order ID is larger than the maximum of the order IDs currently in the warehouse. This is a safe transaction to

execute as many times per day as necessary--even two simultaneous INSERTs would not corrupt the data warehouse with duplicate rows because of the primary key constraint on `order_id`. A daily update is traditional in the data warehousing world so we scheduled one every 24 hours using the Oracle `dbms_job` package (http://www.oradoc.com/ora816/server.816/a76956/jobq.htm#750).

Sample Queries

We have (1) defined a star schema, (2) populated the dimension tables, (3) loaded the fact table, and (4) arranged for periodic updating of the fact table. Now we can proceed to the interesting part of our data warehouse: getting information back out.

Using only the `sales_fact` table, we can ask for

- the total number of orders, total revenue to date, tax paid, shipping costs to date, the average price paid for each item sold, and the average number of days to ship:

```
select count(*) as n_orders,
       round(sum(price_charged)) as total_revenue,
       round(sum(tax_charged)) as total_tax,
       round(sum(shipping_charged)) as total_shipping,
       round(avg(price_charged),2) as avg_price,
       round(avg(days_order_to_shipment),2) as avg_days_to_ship
from sales_fact;
```

- the average number of minutes from login to order (we exclude user sessions longer than 30 minutes to avoid skewing the results from people who interrupted their shopping session to go out to lunch or sleep for a few hours):

```
select round(avg(minutes_login_to_order), 2)
from sales_fact
where minutes_login_to_order < 30
```

- the average number of days from first being invited to the site by email to the first order (excluding periods longer than 2 weeks to remove outliers):

```
select round(avg(days_first_invite_to_order), 2)
from sales_fact
where days_first_invite_to_order < 14
```

Joining against the `ship_to_dimension` table lets us ask how many pants were shipped to each region of the United States:

```
select ship_to_region, count(*) as n_pants
from sales_fact f, ship_to_dimension s
where f.ship_to_key = s.ship_to_key
group by ship_to_region
order by n_pants desc
```

Region	Pants Sold
New England Region	612
NY and NJ Region	321
Mid Atlantic Region	318

Western Region	288
Southeast Region	282
Southern Region	193
Great Lakes Region	177
Northwestern Region	159
Central Region	134
North Central Region	121

Note: these data are based on a random subset of orders from the Levi's site and we have also made manual changes to the report values. The numbers are here to give you an idea of what these queries do, not to provide insight into the Levi's custom clothing business.

Joining against the `time_dimension`, we can ask how many pants were sold for each day of the week:

```
select day_of_week, count(*) as n_pants
from sales_fact f, time_dimension t
where f.time_key = t.time_key
group by day_of_week
order by n_pants desc
```

Day of Week	**Pants Sold**
Thursday	3428
Wednesday	2823
Tuesday	2780
Monday	2571
Friday	2499
Saturday	1165
Sunday	814

We were able to make pants with either a "dressy" or "casual" fabric. Joining against the `product_dimension` table can tell us how popular each option was as a function of color:

```
select color, count(*) as n_pants, sum(decode(fabric,'dressy',1,0)) as n_dressy
from sales_fact f, product_dimension p
where f.product_key = p.product_key
group by color
order by n_pants desc
```

Color	**Pants Sold**	**% Dressy**
dark tan	486	100
light tan	305	49
dark grey	243	100

black	225	97
navy blue	218	61
medium tan	209	0
olive green	179	63

Note: 100% and 0% indicate that those colors were available only in one fabric.

Here is a good case of how the data warehouse may lead to a practical result. If these were the real numbers from the Levi's warehouse, what would pop out at the manufacturing guys is that 97% of the black pants sold were in one fabric style. It might not make sense to keep an inventory of casual black fabric if there is so little consumer demand for it.

Query Generation: The Commercial Closed-Source Route

The promise of a data warehouse is not fulfilled if all users must learn SQL syntax and how to run SQL*PLUS. From being exposed to 10 years of advertising for query tools, we decided that the state of forms-based query tools must be truly advanced. We thus suggested to Levi Strauss that they use Seagate Crystal Reports and Crystal Info to analyze their data. These packaged tools, however, ended up not fitting very well with what Levi's wanted to accomplish. First, constructing queries was not semantically simpler than coding SQL. The Crystal Reports consultant that we brought in said that most of his clients ended up having a programmer set up the report queries and the business people would simply run the report every day against new data. If professional programmers had to construct queries, it seemed just as easy just to write more admin pages using our standard Web development tools, which required about 15 minutes per page. Second, it was impossible to ensure availability of data warehouse queries to authorized users anywhere on the Internet. Finally there were security and social issues associated with allowing a SQL*Net connection from a Windows machine running Crystal Reports out through the Levi's firewall to our Oracle data warehouse on the Web.

Not knowing if any other commercial product would work better and not wanting to disappoint our customer, we extended the ArsDigita Community System with a data warehouse query module that runs as a Web-only tool. This is a free open-source system and comes with the standard ACS package that you can download from http://www.arsdigita.com/download/.

Query Generation: The Open-Source ACS Route

The "dw" module in the ArsDigita Community System is designed with the following goals:

1. naive users can build simple queries by themselves
2. professional programmers can step in to help out the naive users
3. a user with no skill can re-execute a saved query

We keep one row per query in the `queries` table:

```
create table queries (
        query_id        integer primary key,
        query_name      varchar(100) not null,
        query_owner     not null references users,
        definition_time date not null,
        -- if this is non-null, we just forget about all the query_columns
```

```
        -- stuff; the user has hand-edited the SQL
        query_sql       varchar(4000)
);
```

Unless the `query_sql` column is populated with a hand-edited query, the query will be built up by looking at several rows in the `query_columns` table:

```
-- this specifies the columns we we will be using in a query and
-- what to do with each one, e.g., "select_and_group_by" or
-- "select_and_aggregate"

-- "restrict_by" is tricky; value1 contains the restriction value, e.g., '40'
-- or 'MA' and value2 contains the SQL comparion operator, e.g., "=" or ">"

create table query_columns (
        query_id        not null references queries,
        column_name     varchar(30),
        pretty_name     varchar(50),
        what_to_do      varchar(30),
        -- meaning depends on value of what_to_do
        value1          varchar(4000),
        value2          varchar(4000)
);

create index query_columns_idx on query_columns(query_id);
```

The `query_columns` definition appears strange at first. It specifies the name of a column but not a table. This module is predicated on the simplifying assumption that we have one enormous view, `ad_hoc_query_view`, that contains all the dimension tables' columns alongside the fact table's columns.

Here is how we create the view for the Levi's data warehouse:

```
create or replace view ad_hoc_query_view
as
select minutes_login_to_order, days_first_invite_to_order,
       days_order_to_shipment, days_shipment_to_intent, pants_id,
       price_charged, tax_charged, shipping_charged,
       oracle_date, day_of_week,
       day_number_in_month, week_number_in_year, week_number_overall,
       month, month_number_overall, quarter, fiscal_period,
       holiday_flag, weekday_flag, season, color, fabric, cuff_state,
       pleat_state, coupon_state, coupon_range, repeat_class,
       on_time_status, returned_status, ship_to_region, ship_to_state
from sales_fact f, time_dimension t, product_dimension p,
     promotion_dimension pr, consumer_dimension c,
     user_experience_dimension u, ship_to_dimension s
where f.time_key = t.time_key
and f.product_key = p.product_key
and f.promotion_key = pr.promotion_key
and f.consumer_key = c.consumer_key
and f.user_experience_key = u.user_experience_key
and f.ship_to_key = s.ship_to_key;
```

At first glance, this looks like a passport to sluggish Oracle performance. We'll be doing a seven-way JOIN for every data warehouse query, regardless of whether we need information from some of the dimension tables or not.

We can test this assumption as follows:

```
-- tell SQL*Plus to turn on query tracing
set autotrace on

-- let's look at how many pants of each color
-- were sold in each region

SELECT ship_to_region, color, count(pants_id)
FROM ad_hoc_query_view
GROUP BY ship_to_region, color;
```

Oracle will return the query results first...

ship_to_region	color	count(pants_id)
Central Region	black	46
Central Region	dark grey	23
Central Region	dark tan	39
..		
Western Region	medium tan	223
Western Region	navy blue	245
Western Region	olive green	212

... and then explain how those results were obtained:

```
Execution Plan
----------------------------------------------------------
   0      SELECT STATEMENT Optimizer=CHOOSE (Cost=181 Card=15 Bytes=2430)
   1    0   SORT (GROUP BY) (Cost=181 Card=15 Bytes=2430)
   2    1     NESTED LOOPS (Cost=12 Card=2894 Bytes=468828)
   3    2       HASH JOIN (Cost=12 Card=885 Bytes=131865)
   4    3         TABLE ACCESS (FULL) OF 'PRODUCT_DIMENSION' (Cost=1 Card=336
Bytes=8400)
   5    3         HASH JOIN (Cost=6 Card=885 Bytes=109740)
   6    5           TABLE ACCESS (FULL) OF 'SHIP_TO_DIMENSION' (Cost=1 Card=55
Bytes=1485)
   7    5           NESTED LOOPS (Cost=3 Card=885 Bytes=85845)
   8    7             NESTED LOOPS (Cost=3 Card=1079 Bytes=90636)
   9    8               NESTED LOOPS (Cost=3 Card=1316 Bytes=93436)
  10    9                 TABLE ACCESS (FULL) OF 'SALES_FACT' (Cost=3 Card=1605
Bytes=93090)
  11    9                 INDEX (UNIQUE SCAN) OF 'SYS_C0016416' (UNIQUE)
  12    8               INDEX (UNIQUE SCAN) OF 'SYS_C0016394' (UNIQUE)
  13    7             INDEX (UNIQUE SCAN) OF 'SYS_C0016450' (UNIQUE)
  14    2       INDEX (UNIQUE SCAN) OF 'SYS_C0016447' (UNIQUE)
```

As you can see from the table names in bold face, Oracle was smart enough to examine only tables relevant to our query: `product_dimension`, because we asked about color; `ship_to_dimension`, because we asked about region; `sales_fact`, because we asked for a count of pants sold. Bottom line: Oracle did a 3-way JOIN instead of the 7-way JOIN specified by the view.

To generate a SQL query into `ad_hoc_query_view` from the information stored in `query_columns` is most easily done with a function in a procedural language such as Java, PL/SQL, Perl, or Tcl (here is pseudocode):

```
proc generate_sql_for_query(a_query_id)
    select_list_items list;
    group_by_items list;
    order_clauses list;

    foreach row in "select column_name, pretty_name
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'select_and_group_by'"]
        if row.pretty_name is null then
            append_to_list(group_by_items, row.column_name)
        else
            append_to_list(group_by_items, row.column_name || ' as "' ||
row.pretty_name || '"'
        end if
    end foreach

    foreach row in "select column_name, pretty_name, value1
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'select_and_aggregate'"
         if row.pretty_name is null then
            append_to_list(select_list_items, row.value1 || row.column_name)
         else
            append_to_list(select_list_items, row.value1 || row.column_name || ' as
"' || row.pretty_name || '"'
         end if
    end foreach

    foreach row in "select column_name, value1, value2
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'restrict_by'"
        append_to_list(where_clauses, row.column_name || ' ' || row.value2 || ' '
|| row.value1)
    end foreach

    foreach row in "select column_name
                    from query_columns
                    where query_id = a_query_id
                      and what_to_do = 'order_by'"]
        append_to_list(order_clauses, row.column_name)
    end foreach

    sql := "SELECT " || join(select_list_items, ', ') ||
           " FROM ad_hoc_query_view"

    if list_length(where_clauses) > 0 then
        append(sql, ' WHERE ' || join(where_clauses, ' AND '))
```

```
    end if

    if list_length(group_by_items) > 0 then
        append(sql, ' GROUP BY ' || join(group_by_items, ', '))
    end if

    if list_length(order_clauses) > 0 then
        append(sql, ' ORDER BY ' || join(order_clauses, ', '))
    end if

    return sql
end proc
```

How well does this work in practice? Suppose that we were going to run regional advertisements. Should the models be pictured where pleated or plain front pants? We need to look at recent sales by region. With the ACS query tool, a user can use HTML forms to specify the following:

- pants_id : select and aggregate using count
- ship_to_region : select and group by
- pleat_state : select and group by

The preceding pseudocode turns that into

```
SELECT ship_to_region, pleat_state, count(pants_id)
FROM ad_hoc_query_view
GROUP BY ship_to_region, pleat_state
```

which is going to report sales going back to the dawn of time. If we weren't clever enough to anticipate the need for time windowing in our forms-based interface, the "hand edit the SQL" option will save us. A professional programmer can be grabbed for a few minutes to add

```
SELECT ship_to_region, pleat_state, count(pants_id)
FROM ad_hoc_query_view
WHERE oracle_date > sysdate - 45
GROUP BY ship_to_region, pleat_state
```

Now we're limiting results to the last 45 days:

ship_to_region	pleat_state	count(pants_id)
Central Region	plain front	8
Central Region	pleated	26
Great Lakes Region	plain front	14
Great Lakes Region	pleated	63
Mid Atlantic Region	plain front	56
Mid Atlantic Region	pleated	162
NY and NJ Region	plain front	62
NY and NJ Region	pleated	159
New England Region	plain front	173
New England Region	pleated	339
North Central Region	plain front	7

North Central Region	pleated	14
Northwestern Region	plain front	20
Northwestern Region	pleated	39
Southeast Region	plain front	51
Southeast Region	pleated	131
Southern Region	plain front	13
Southern Region	pleated	80
Western Region	plain front	68
Western Region	pleated	120

If we strain our eyes and brains a bit, we can see that plain front pants are very unpopular in the Great Lakes and South but more popular in New England and the West. It would be nicer to see percentages within region, but standard SQL does not make it possible to combine results to values in surrounding rows. We will need to refer to the "SQL for Analysis" chapter in the Oracle data warehousing documents to read up on extensions to SQL that makes this possible:

```
SELECT
  ship_to_region,
  pleat_state,
  count(pants_id),
  ratio_to_report(count(pants_id))
       over (partition by ship_to_region) as percent_in_region
FROM ad_hoc_query_view
WHERE oracle_date > sysdate - 45
GROUP BY ship_to_region, pleat_state
```

We're asked Oracle to window the results ("partition by ship_to_region") and compare the number of pants in each row to the sum across all the rows within a regional group. Here's the result:

ship_to_region	pleat_state	count(pants_id)	percent_in_region
...			
Great Lakes Region	plain front	14	.181818182
Great Lakes Region	pleated	63	.818181818
...			
New England Region	plain front	173	.337890625
New England Region	pleated	339	.662109375
...			

This isn't quite what we want. The "percents" are fractions of 1 and reported with far too much precision. We tried inserting the Oracle built-in `round` function in various places of this SQL statement but all we got for our troubles was "ERROR at line 5: ORA-30484: missing window specification for this function". We had to add an extra layer of SELECT, a view-on-the-fly, to get the report that we wanted:

```
select ship_to_region, pleat_state, n_pants, round(percent_in_region*100)
from
(SELECT
```

```
   ship_to_region,
   pleat_state,
   count(pants_id) as n_pants,
   ratio_to_report(count(pants_id))
        over (partition by ship_to_region) as percent_in_region
 FROM ad_hoc_query_view
 WHERE oracle_date > sysdate - 45
 GROUP BY ship_to_region, pleat_state)
```

returns

ship_to_region	pleat_state	count(pants_id)	percent_in_region
...			
Great Lakes Region	plain front	14	18
Great Lakes Region	pleated	63	82
...			
New England Region	plain front	173	34
New England Region	pleated	339	66
...			

What if you're in charge of the project?

If you are in charge of a data warehousing project, you need to assemble the necessary tools. Do not be daunted by this prospect. The entire Levi Strauss system described above was implemented in three days by two programmers.

The first tool that you need is intelligence and thought. If you pick the right dimensions and put the required data into them, your data warehouse will be useful. If you don't get your dimensions right, you won't even be able to ask the interesting questions. If you're not smart or thoughtful, probably the best thing to do is find a boutique consulting firm with expertise in building data warehouses for your industry. Get them to lay out the initial star schema. They won't get it right but it should be close enough to live with for a few months. If you can't find an expert, *The Data Warehouse Toolkit* (Ralph Kimball 1996) contains example schemata for 10 different kinds of businesses.

You will need some place to store your data and query parts back out. Since you are using SQL your only choice is a relational database management system. There are specialty vendors that have historically made RDBMSes with enhanced features for data warehousing, such as the ability to compute a value based on information from the current row compared to information from a previously output row of the report. This gets away from the strict unordered set-theoretic way of looking at the world that E.F. Codd sketched in 1970 but has proven to be useful. Starting with version 8.1.6, Oracle has added most of the useful third-party features into their standard product. Thus all but the very smallest and very largest modern data warehouses tend to be built using Oracle (see the "SQL for Analysis" chapter in the *Oracle8i Data Warehousing Guide* volume of the Oracle documentation).

Oracle contains two features that may enable you to construct and use your data warehouse without investing in separate hardware. First is the optimistic locking system that Oracle has employed since the late 1980s. If someone is doing a complex query it will not affect transactions that need to update

the same tables. Essentially each query runs in its own snapshot of the database as it existed when the query was started. The second Oracle feature is *materialized views* or *summaries*. It is possible to instruct the database to keep a summary of sales by quarter, for example. If someone asks for a query involving quarterly sales, the small summary table will be consulted instead of the comprehensive sales table. This could be 100 to 1000 times faster.

One typical goal of a data warehousing project is to provide a unified view of a company's disparate information systems. The only way to do this is to extract data from all of these information systems and clean up those data for consistency and accuracy. This is purportedly a challenging task when RDBMSes from different vendors are involved, though it might not seem so on the surface. After all, every RDBMS comes with a C library. You could write a C program to perform queries on the Brand X database and do inserts on the Brand Y database. Perl and Tcl have convenient facilities for transforming text strings and there are db connectivity interfaces from these scripting languages to DBMS C libraries. So you could write a Perl script. Most databases within a firm are accessible via the Web, at least within a company's internal network. Oracle includes a Java virtual machine and Java libraries to fetch Web pages and parse XML. So you could write a Java or PL/SQL program running inside your data warehouse Oracle installation to grab the foreign information and bring it back (see the chapter on foreign and legacy data).

If you don't like to program or have a particularly knotty connectivity problem involving an old mainframe, various companies make software that can help. For high-end mainframe stuff, Oracle Corporation itself offers some useful layered products. For low-end "more-convenient-than-Perl" stuff, Data Junction (www.datajunction.com) is useful.

Given an already-built data warehouse, there are a variety of useful query tools. The theory is that if you've organized your data model well enough, a non-technical user will be able to navigate around via a graphic user interface or a Web browser. The best known query tool is Crystal Reports (www.seagatesoftware.com), which we tried to use in the Levi Strauss example. See http://www.arsdigita.com/doc/dw for details on the free open-source ArsDigita Community System data warehouse query module.

Is there a bottom line to all of this? If you can think sufficiently clearly about your organization and its business to construct the correct dimensions and program SQL reasonably well, you will be successful with the raw RDBMS alone. Extra software tools can potentially make the project a bit less painful or a bit shorter but they won't be of critical importance.

More Information

The construction of data warehouses is a guild-like activity. Most of the expert knowledge is contained within firms that specialize not in data warehousing but in data warehousing for a particular kind of company. For example, there are firms that do nothing but build data warehouses for supermarkets. There are firms that do nothing but build data warehouses for department stores. Part of what keeps this a tight guild is the poor quality of textbooks and journal articles on the subject. Most of the books on data warehousing are written by and for people who do not know SQL. The books focus on (1) stuff that you can buy from a vendor, (2) stuff that you can do from a graphical user interface after the data

warehouse is complete, and (3) how to navigate around a large organization to get all the other suits to agree to give you their data, their money, and a luxurious schedule.

The only worthwhile introductory book that we've found on data warehousing in general is Ralph Kimball's *The Data Warehouse Toolkit*. Kimball is also the author of an inspiring book on clickstream data warehousing: *The Data Webhouse Toolkit*. The latter book is good if you are interested in applying classical dimensional data warehousing techniques to user activity analysis.

It isn't exactly a book and it isn't great for beginners but the *Oracle8i Data Warehousing Guide* volume of the official Oracle server documentation is extremely useful.

Data on consumer purchasing behavior are available from A.C. Nielsen (www.acnielsen.com), Information Resources Incorporated (IRI; www.infores.com), and a bunch of other companies listed in http://dir.yahoo.com/Business_and_Economy/Business_to_Business/Marketing_and_Advertising/Market_Research/.

Reference

- *Oracle8i Data Warehousing Guide*, particularly the the "SQL for Analysis" chapter
- ROLLUP examples from the Oracle Application Developer's Guide: http://www.oradoc.com/keyword/rollup

Next: Foreign and Legacy Data

---

philg@mit.edu

Reader's Comments

I really like that Walmart/Sybase example, because Walmart is actually running the largest commercial data warehouse in the world including 2 years of detail data with tens of billions of detail rows. Of course, it's not using an OLTP system like Sybase/Oracle, it's a decision support database, Teradata.

-- Dieter Noeth, May 14, 2003

Foreign and Legacy Data

part of SQL for Web Nerds (this chapter written by Michael Booth and Philip Greenspun)

Most of the world's useful data are either residing on a server beyond your control or inside a database management system other than Oracle. Either way, we refer to these data as *foreign* from the perspective of one's local

Oracle database. Your objective is always the same: **Treat foreign data as though they were residing in local SQL tables.**

The benefits of physically or virtually dragging foreign data back to your Oracle cave are the following:

- New developers don't have to think about where data are coming from.
- Developers can work with foreign data using the same query language that they use every day: SQL.
- Developers can work with the same programming tools and systems that they've been using daily. They might use COBOL, they might use C, they might use Java, they might use Common Lisp, they might use Perl or Tcl, but you can be sure that they've learned how to send an SQL query to Oracle from these tools and therefore that they will be able to use the foreign data.

A good conceptual way to look at what we're trying to accomplish is the construction of an SQL view of the foreign data. A standard SQL view may hide a 5-way join from the novice programmer. A *foreign table* hides the fact that data are coming from a foreign source.

We will refer to the system that makes foreign data available locally as an *aggregation architecture*. In designing an individual aggregation architecture you will need to address the following issues:

1. Coherency: Is it acceptable for your local version of the data to be out of sync with the foreign data? If so, to what extent?
2. Updatability: Is your local view updatable? I.e., can a programmer perform a transaction on the foreign database management system by updating the local view?
3. Social: To what extent is the foreign data provider willing to let you into his or her database?

Degenerate Aggregation

If the foreign data are stored in an Oracle database and the people who run that database are cooperative, you can practice a degenerate form of aggregation: Oracle to Oracle communication.

The most degenerate form of degenerate aggregation is when the foreign data are in the same Oracle installation but owned by a different user. To make it concrete, let's assume that you work for Eli Lilly in the data warehousing department, which is part of marketing. Your Oracle user name is "marketing". The foreign data in which you're interested are in the `prozac_orders` table, owned by the "sales" user. In this case your aggregation architecture is the following:

1. connect to Oracle as the foreign data owner (sales) and GRANT SELECT ON PROZAC_ORDERS TO MARKETING
2. connect to Oracle as the local user (marketing) and type SELECT * FROM SALES.PROZAC_ORDERS

Done.

Slightly Less Degenerate Aggregation

Imagine now that Prozac orders are processed on a dedicated on-line transaction processing (OLTP) database installation and your data warehouse (DW) is running on a physically separate computer. Both

the OLTP and DW systems are running the Oracle database server and they are connected via a network. You need to take the following steps:

1. set up SQL*Net (Net8) on the OLTP system
2. set up the DW Oracle server to be a client to the OLTP server. If you are not using the Oracle naming services, you must edit $ORACLE_HOME/network/admin/tnsnames.ora to reference the OLTP system.
3. create a "marketing" user on the OLTP system
4. on the OLTP system, log in as "sales" and GRANT SELECT ON PROZAC_ORDERS TO MARKETING
5. on the DW system, create a database link to the OLTP system named "OLTP": CREATE DATABASE LINK OLTP CONNECT TO MARKETING IDENTIFIED BY *password for marketing* USING 'OLTP';
6. on the DW system, log in as "marketing" and SELECT * FROM SALES.PROZAC_ORDERS@OLTP;

In both of these degenerate cases, there were no coherency issues. The foreign data were queried in real-time from their canonical database. This was made possible because of social agreement. The owners of the foreign data were willing to grant you unlimited access. Similarly, social issues decided the issue of updatability. With a GRANT only on SELECT, the foreign table would not be updatable.

Non-Oracle-Oracle Aggregation

What if foreign data aren't stored in an Oracle database or they are but you can't convince the owners to give you access? You will need some sort of computer program that knows how to fetch some or all of the foreign data and stuff it into an Oracle table locally. Let's start with a concrete example.

Suppose you work at Silicon Valley Blue Cross. The dimensional data warehouse has revealed a strong correlation between stock market troubles and clinical depression. People working for newly public companies with volatile stocks tend to get into a funk when their paper wealth proves illusory. The suits at Blue Cross think that they can save money and doctors' visits by automatically issuing prescriptions for Prozac whenever an insured's employer's stock drops more than 10% in a day. To find candidates for happy pills, the following query should suffice:

```
select patients.first_names, patients.last_name, stock_quotes.percent_change
from patients, employers, stock_quotes
where patients.employer_id = employers.employer_id
and employers.ticker_symbol = stock_quotes.ticker_symbol
and stock_quotes.percent_change < -0.10
order by stock_quotes.percent_change
```

The `stock_quotes` table is the foreign table here. Blue Cross does not operate a stock exchange. Therefore the authoritative price change data must necessarily be pulled from an external source. Imagine that the external source is http://quote.yahoo.com/. The mature engineering perspective on a Web site such as quote.yahoo.com is that it is an object whose methods are its URLs and the arguments to those methods are the form variables. To get a quotation for the software company Ariba (ticker ARBA), for example, we need to visit http://quote.yahoo.com/q?s=arba in a Web browser. This is invoking the method "q" with an argument of "arba" for the form variable "s". The results come back in

a human-readable HTML page with a lot of presentation markup and English text. It would be more convenient if Yahoo gave us results in a machine-readable form, e.g., a comma-separated list of values or an XML document. However, the HTML page may still be used as long as its structure does not vary from quote to quote and the percent change number can be pulled out with a regular expression or other computer program.

What are the issues in designing an aggregation architecture for this problem? First is coherency. It would be nice to have up-to-the-minute stock quotes but, on the other hand, it seems kind of silly to repeatedly query quote.yahoo.com for the same symbol. In fact, after 4:30 PM eastern time when the US stock market closes, there really isn't any reason to ask for a new quote on a symbol until 9:30 AM the next day. Given some reasonable assumptions about caching, once the `stock_quotes` table has been used a few times, queries will be able to execute much much faster since quote data will be pulled from a local cache rather than fetched over the Internet.

We don't have to think very hard about updatability. Blue Cross does not run a stock exchange therefore Blue Cross cannot update a stock's price. Our local view will not be updatable.

The social issue seems straightforward at first. Yahoo is making quotes available to any client on the public Internet. It looks at first glance as though our computer program can only request one quote at a time. However, if we fetch http://quote.yahoo.com/q?s=arba+**ibm**, we can get two quotes at the same time. It might even be possible to grab all of our insureds' employers' stock prices in one big page. A potential fly in the ointment is Yahoo's terms of service at http://docs.yahoo.com/info/terms/ where they stipulate

```
10. NO RESALE OF SERVICE

You agree not to reproduce, duplicate, copy, sell, resell or exploit for
any commercial purposes, any portion of the Service, use of the Service,
or access to the Service.
```

Where and when to run our programs

We need to write a computer program (the "fetcher") that can fetch the HTML page from Yahoo, pull out the price change figures, and stuff them into the `stock_quotes` table. We also need a more general computer program (the "checker") that can look at the foreign data required, see how old the cached data in `stock_quotes` are, and run the fetcher program if necessary.

There are three Turing-complete computer languages built into Oracle: C, Java, PL/SQL. "Turing-complete" means that any program that can be written for any computer can be written to run inside Oracle. Since you eventually want the foreign data to be combined with data inside Oracle, it makes sense to run all of your aggregation code inside the database. Oracle includes built-in functions to facilitate the retrieval of Web pages (see http://oradoc.photo.net/ora816/server.816/a76936/utl_http.htm#998100).

In an ideal world you could define a database trigger that would fire every time a query was about to SELECT from the `stock_quotes` table. This trigger would somehow figure out which rows of the foreign table were going to be required. It would run the checker program to make sure that none of the cached data were too old, and the checker in turn might run the fetcher.

Why won't this work? As of Oracle version 8.1.6, it is impossible to define a trigger on SELECT. Even if you could, there is no advertised way for the triggered program to explore the SQL query that is being executed or to ask the SQL optimizer which rows will be required.

The PostgreSQL RDBMS has a "rule system" (see http://www.postgresql.org/docs/programmer/x968.htm) which can intercept and transform a SELECT. It takes the output of the SQL parser, applies one or more transformation rules, and produces a new set of queries to be executed. For example, a rule may specify that any SELECT which targets the table "foo" should be turned into a SELECT from the table "bar" instead; this is how Postgres implements views. As it stands, the only transformation that can be applied to a SELECT is to replace it with a single, alternative SELECT - but PostgreSQL is open source software which anyone is free to enhance.

The long term fix is to wait for the RDBMS vendors to augment their products. Help is on the way. The good news is that a portion of the ANSI/ISO SQL-99 standard mandates that RDBMS vendors, including Oracle, provide support for wrapping external data sources. The bad news is that the SQL-99 standard is being released in chunks, the wrapper extension won't be published until 2001, and it may be several years before commercial RDBMSes implement the new standard.

The short term fix is to run a procedure right before we send the query to Oracle:

```
call checker.stock_quotes( 0.5 )

select patients.first_names, patients.last_name, stock_quotes.percent_change ...
```

Our checker is an Oracle stored procedure named `checker.stock_quotes`. It checks every ticker symbol in `stock_quotes` and calls the fetcher if the quote is older than the specified interval, measured in days. If we want to add a new `ticker_symbol` to the table, we call a different version of `checker.stock_quotes`:

```
call checker.stock_quotes( 0.5, 'IBM' )
```

If there is no entry for IBM which is less than half a day old, the checker will ask the fetcher to get a stock quote for IBM.

An Aggregation Example

Blue Cross will dispense a lot of Prozac before Oracle implements the SQL-99 wrapper extension. So let's build a `stock_quotes` foreign table which uses Java stored procedures to do the checking and fetching. We'll begin with a data model:

```
create table stock_quotes (
        ticker_symbol          varchar(20) primary key,
        last_trade             number,
         -- the time when the last trade occurred (reported by Yahoo)
        last_trade_time        date,
        percent_change         number,
         -- the time when we pulled this data from Yahoo
        last_modified          date not null
);
```

This is a stripped-down version, where we only store the most recent price quote for each ticker symbol. In a real application we would certainly want to maintain an archive of old quotes, perhaps by

using triggers to populate an audit table whenever `stock_quotes` is updated. Even if your external source provides its own historical records, fetching them is bound to be slower, less reliable, and more complicated than pulling data from your own audit tables.

We'll create a single source code file, `StockUpdater.java`. Oracle 8.1.6 includes a Java compiler as well as a virtual machine, so when this file is ready we can load it into Oracle and compile it with a single command:

```
bash-2.03$ loadjava -user username/password -resolve -force StockUpdater.java
    ORA-29535: source requires recompilation
    StockUpdater:171: Class Perl5Util not found.
    StockUpdater:171: Class Perl5Util not found.
    StockUpdater:218: Class PatternMatcherInput not found.
    StockUpdater:218: Class PatternMatcherInput not found.
     Info: 4 errors
loadjava: 6 errors
bash-2.03$
```

Oops. The `-resolve` option tells Oracle's `loadjava` utility to compile and link the class right away, but `StockUpdater` depends on classes that haven't yet been loaded into Oracle. Most Java virtual machines are designed to automatically locate and load classes at runtime by searching through the filesystem, but the Oracle JVM requires every class to be loaded into the database in advance.

We need to obtain the `Perl5Util` and `PatternMatcherInput` classes. These are part of the Oro library, an open-source regular expression library that's available from http://jakarta.apache.org/oro/index.html. When we download and untar the distribution, we'll find a JAR file that contains the classes we need. We'll load the entire JAR file into Oracle and then try to load `StockUpdater` again.

```
bash-2.03$ loadjava -user username/password -resolve jakarta-oro-2.0.jar
bash-2.03$ loadjava -user username/password -resolve -force StockUpdater.java
bash-2.03$
```

These commands take a while to execute. When they're done, we can check the results by running this SQL query:

```
SELECT RPAD(object_name,31) ||
       RPAD(object_type,14) ||
       RPAD(status,8)
       "Java User Objects"
  FROM user_objects
 WHERE object_type LIKE 'JAVA %';
```

Here's a small portion of the output from this query:

```
Java User Objects
-------------------------------------------------
StockUpdater                   JAVA CLASS    VALID
StockUpdater                   JAVA SOURCE   VALID
org/apache/oro/text/awk/OrNode JAVA CLASS     VALID
org/apache/oro/text/regex/Util JAVA CLASS     VALID
org/apache/oro/util/Cache      JAVA CLASS     VALID
org/apache/oro/util/CacheFIFO  JAVA CLASS     VALID
...
```

Our source code is marked `VALID`, and there's an associated class which is also `VALID`. There are a bunch of `VALID` regexp classes. All is well.

If we wanted, we could have compiled the `StockUpdater` class using a free-standing Java compiler and then loaded the resulting class files into Oracle. We aren't required to use the built-in Oracle compiler.

The `-force` option forces `loadjava` to overwrite any existing class with the same name, so if we change our class we don't necessarily have to drop the old version before loading the new one. If we do want to drop one of Oracle's stored Java classes, we can use the `dropjava` utility.

Calling our Stored Procedures

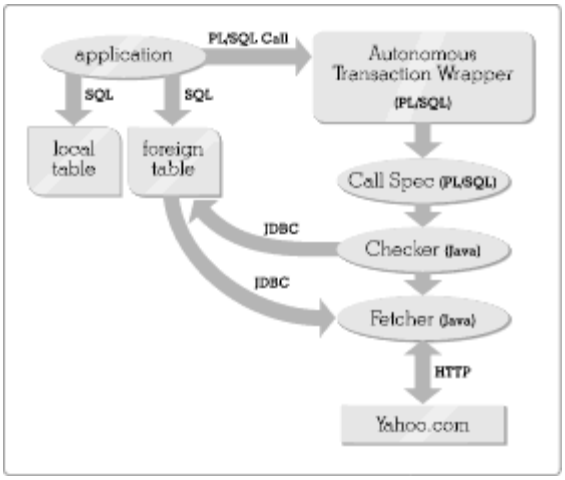


Figure 15-1: The aggregation architecture. The client application obtains data by querying Oracle tables using SQL. To keep the foreign tables up to date, the application calls the Checker and the Fetcher, which are Java stored procedures running inside Oracle. The Checker is called via two layers of PL/SQL: one layer is a call spec which translates a PL/SQL call to a Java call, and the other is a wrapper procedure which provides an autonomous transaction for the aggregation code to run in.
In order to call Java stored procedures from SQL, we need to define *call specs*, which are PL/SQL front ends to static Java methods. Here's an example of a call spec:

```
PROCEDURE stock_quotes_spec ( interval IN number )
    AS LANGUAGE JAVA
    NAME 'StockUpdater.checkAll( double )';
```

This code says: "when the programmer calls this PL/SQL procedure, call the `checkAll` method of the Java class `StockUpdater`." The `checkAll` method must be `static`: Oracle doesn't automatically construct a new `StockUpdater` object.

We don't allow developers to use the call spec directly. Instead we make them call a separate PL/SQL procedure which initiates an *autonomous transaction*. We need to do this because an application might call the checker in the middle of a big transaction. The checker uses the fetcher to add fresh data to the `stock_quotes` table. Now the question arises: when do we commit the changes to the `stock_quotes` table? There are three options:

1. Have the fetcher issue a COMMIT. This will commit the changes to `stock_quotes`. It will also commit any changes that were made before the checker was called. This is a very bad idea.

2. Have the fetcher update `stock_quotes` without issuing a COMMIT. This is also a bad idea: if the calling routine decides to abort the transaction, the new stock quote data will be lost.

3. Run the checker and the fetcher in an independent transaction of their own. The fetcher can commit or roll back changes without affecting the main transaction. Oracle provides the `AUTONOMOUS_TRANSACTION` pragma for this purpose, but the pragma doesn't work in a call spec - it's only available for regular PL/SQL procedures. So we need a separate layer of glue code just to initiate the autonomous transaction.

Here's the SQL which defines all of our PL/SQL procedures:

```
CREATE OR REPLACE PACKAGE checker AS
    PROCEDURE stock_quotes( interval IN number );
    PROCEDURE stock_quotes( interval IN number, ticker_symbol IN varchar );
END checker;
/
show errors


CREATE OR REPLACE PACKAGE BODY checker AS

    -- Autonomous transaction wrappers
    PROCEDURE stock_quotes ( interval IN number )
    IS
        PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
        stock_quotes_spec( interval );
    END;

    PROCEDURE stock_quotes ( interval IN number, ticker_symbol IN varchar )
    IS
        PRAGMA AUTONOMOUS_TRANSACTION;
    BEGIN
        stock_quotes_spec( interval, ticker_symbol );
    END;


    -- Call specs
    PROCEDURE stock_quotes_spec ( interval IN number )
        AS LANGUAGE JAVA
        NAME 'StockUpdater.checkAll( double )';

    PROCEDURE stock_quotes_spec ( interval IN number, ticker_symbol IN varchar )
        AS LANGUAGE JAVA
        NAME 'StockUpdater.checkOne( double, java.lang.String )';


END checker;
/
show errors
```

We've placed the routines in a *package* called `checker`. Packages allow us to group procedures and datatypes together. We're using one here because packaged procedure definitions can be *overloaded*.

The `checker.stock_quotes` procedure can be called with either one or two arguments and a different version will be run in each case. The `stock_quotes_spec` procedure also comes in two versions.

Writing a Checker in Java

We're ready to start looking at the `StockUpdater.java` file itself. It begins in typical Java fashion:

```
// Standard Java2 classes, already included in Oracle
import java.sql.*;
import java.util.*;
import java.io.*;
import java.net.*;

// Regular expression classes
import org.apache.oro.text.perl.*;
import org.apache.oro.text.regex.*;


public class StockUpdater {
```

Then we have the two checker routine, starting with the one that updates the entire table:

```
    public static void checkAll( double interval )
    throws SQLException {

        // Query the database for the ticker symbols that haven't
        // been updated recently
        String sql = new String( "SELECT ticker_symbol " +
                                 "FROM stock_quotes " +
                                "WHERE (sysdate - last_modified) > " +
                                   String.valueOf( interval ) );

        // Build a Java List of the ticker symbols

        // Use JDBC to execute the given SQL query.
        Connection conn = getConnection();
        Statement stmt = conn.createStatement();
        stmt.execute( sql );
        ResultSet res = stmt.getResultSet();

        // Go through each row of the result set and accumulate a list
        List tickerList = new ArrayList();
        while ( res.next() ) {
            String symbol = res.getString( "ticker_symbol" );
            if ( symbol != null ) {
                tickerList.add( symbol );
            }
        }
        stmt.close();

        System.out.println( "Found a list of " + tickerList.size() + " symbols.");

        // Pass the List of symbols on to the fetcher
        fetchList( tickerList );
    }
```

This routine uses JDBC to access the `stock_quotes` table. JDBC calls throw exceptions of type `SQLException` which we don't bother to catch; instead, we propagate them back to the calling programmer to indicate that something went wrong. We also print debugging information to standard output. When running this class outside Oracle, the debug messages will appear on the screen. Inside Oracle, we can view them by issuing some SQL*Plus commands in advance:

```
SET SERVEROUTPUT ON
CALL dbms_java.set_output(5000);
```

Standard output will be echoed to the screen, 5000 characters at a time.

The second checker operates on one ticker symbol at a time, and is used to add a new ticker symbol to the table:

```
public static void checkOne( double interval, String tickerSymbol )
throws SQLException {

    // Set up a list in case we need it
    List tickerList = new ArrayList();
    tickerList.add( tickerSymbol );

    // Query the database to see if there's recent data for this tickerSymbol
    String sql = new String( "SELECT " +
                             "  ticker_symbol, " +
                             "  (sysdate - last_modified) as staleness " +
                            "FROM stock_quotes " +
                            "WHERE ticker_symbol = '" + tickerSymbol + "'");
    Connection conn = getConnection();
    Statement stmt = conn.createStatement();
    stmt.execute( sql );
    ResultSet res = stmt.getResultSet();

    if ( res.next() ) {
        // A row came back, so the ticker is in the DB
        // Is the data recent?
        if ( res.getDouble("staleness") > interval ) {
            // Fetch fresh data
            fetchList( tickerList );
        }

    } else {
        // The stock isn't in the database yet
        // Insert a blank entry
        stmt.executeUpdate( "INSERT INTO stock_quotes " +
                            "(ticker_symbol, last_modified) VALUES " +
                            "('" + tickerSymbol + "', sysdate)" );
        conn.commit();

        // Now refresh the blank entry to turn it into a real entry
        fetchList( tickerList );
    }
    stmt.close();
}
```

Writing a Fetcher in Java

The fetcher is implemented as a long Java method called `fetchList`. It begins by retrieving a Web page from Yahoo. For speed and simplicity, we extract all of the stock quotes on a single page.

```
    /** Accepts a list of stock tickers and retrieves stock quotes from Yahoo
Finance
        at http://quote.yahoo.com/
    */
    private static void fetchList( List tickerList )
    throws SQLException {

        // We need to pass Yahoo a string containing ticker symbols separated by
"+"
        String tickerListStr = joinList( tickerList, "+" );

        if ( tickerListStr.length() == 0 ) {
            // We don't bother to fetch a page if there are no ticker symbols
            System.out.println("Fetcher: no ticker symbols were supplied");
            return;
        }

        try {
            // Go get the Web page
            String url = "http://quote.yahoo.com/q?s=" + tickerListStr;
            String yahooPage = getPage( url );
```

The fetcher uses a helper routine called `getPage` to retrieve Yahoo's HTML, which we stuff into the `yahooPage` variable. Now we can use Perl 5 regular expressions to extract the values we need. We create a new `Perl5Util` object and use the `split()` and `match()` methods to extract the section of the page where the data is:

```
            // ... continuing the definition of fetchList ...

            // Get a regular expression matcher
            Perl5Util regexp = new Perl5Util();

            // Break the page into sections using </table> tags as boundaries
            Vector allSections = regexp.split( "/<\\/table>/", yahooPage );

            // Pick out the section which contains the word "Symbol"
            String dataSection = "";
            boolean foundSymbolP = false;
            Iterator iter = allSections.iterator();
            while ( iter.hasNext() ) {
                dataSection = (String) iter.next();
                if ( regexp.match( "/<th.*?>Symbol<\\/th>/", dataSection )) {
                    foundSymbolP = true;
                    break;
                }
            }

            // If we didn't find the section we wanted, throw an error
            if ( ! foundSymbolP ) {
                throw new SQLException( "Couldn't find the word 'Symbol' in " + url
);
```

```
            }
```

We need to pick out today's date from the page. This is the date when the page was retrieved, which we'll call the "fetch date". Each stock quote also has an individual timestamp, which we'll call the "quote date". We use a little class of our own (`OracleDate`) to represent dates, and a helper routine (`matchFetchDate`) to do the regexp matching.

```
            OracleDate fetchDate = matchFetchDate( dataSection );
            if ( fetchDate == null ) {
                throw new SQLException("Couldn't find the date in " + url);
            }
            System.out.println("The date appears to be: '" + fetchDate.getDate() +
"'");
```

If we can't match the fetch date, we throw an exception to tell the client programmer that the fetcher didn't work. Perhaps the network is down, or Yahoo's server is broken, or Yahoo's graphic designers decided to redesign the page layout.

We're ready to extract the stock quotes themselves. They're in an HTML table, with one row for each quote. We set up a single JDBC statement which will be executed over and over, using placeholders to represent the data:

```
            String update_sql = "UPDATE stock_quotes SET " +
                "last_trade = ?, " +
                "last_trade_time = to_date(?, ?), " +
                "percent_change = ?, " +
                "last_modified = sysdate " +
                "WHERE ticker_symbol = ? ";
            Connection conn = getConnection();
            PreparedStatement stmt = conn.prepareStatement( update_sql );
```

Now we pick apart the HTML table one row at a time, using a huge regexp that represents an entire table row. By using a `PatternMatcherInput` object, we can make `regexp.match()` traverse the `dataSection` string and return one match after another until it runs out of matches. For each stock quote we find, we clean up the data and perform a database INSERT.

```
            // Use a special object to make the regexp search run repeatedly
            PatternMatcherInput matchInput = new PatternMatcherInput( dataSection
);

            // Search for one table row after another
            while ( regexp.match( "/<tr.*?>.*?" +
                                  "<td nowrap.*?>(.*?)<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)<\\/td>.*?" +
                                  "<td nowrap.*?>(.*?)<\\/td>.*?" +
                                  "<\\/tr>/s" , matchInput )) {
                // Save the regexp groups into variables
                String tickerSymbol = regexp.group(1);
                String timeStr = regexp.group(2);
                String lastTrade = regexp.group(3);
                String percentChange = regexp.group(5);

                // Filter the HTML from the ticker symbol
                tickerSymbol = regexp.substitute("s/<.*?>//g", tickerSymbol);
                stmt.setString( 5, tickerSymbol );
```

```
                // Parse the time stamp
                OracleDate quoteDate = matchQuoteDate( timeStr, fetchDate );
                if ( quoteDate == null ) {
                    throw new SQLException("Bad date format");
                }
                stmt.setString( 2, quoteDate.getDate() );
                stmt.setString( 3, quoteDate.getDateFormat() );

                // Parse the lastTrade value, which may be a fraction
                stmt.setFloat( 1, parseFraction( lastTrade ));

                // Filter HTML out of percentChange, and remove the % sign
                percentChange = regexp.substitute( "s/<.*?>//g", percentChange);
                percentChange = regexp.substitute( "s/%//g", percentChange);
                stmt.setFloat( 4, Float.parseFloat( percentChange ));

                // Do the database update
                stmt.execute();
            }

            stmt.close();
            // Commit the changes to the database
            conn.commit();

        } catch ( Exception e ) {
            throw new SQLException( e.toString() );
        }
    } // End of the fetchList method
```

Helper Routines for the Fetcher

The fetcher loads HTML pages using the `getPage` method, which uses the `URL` class from the Java standard library. For a simple HTTP GET, this routine is all we need.

```
    /** Fetch the text of a Web page using HTTP GET
     */
    private static String getPage( String urlString )
    throws MalformedURLException, IOException {
        URL url = new URL( urlString );
        BufferedReader pageReader =
            new BufferedReader( new InputStreamReader( url.openStream() ) );
        String oneLine;
        String page = new String();
        while ( (oneLine = pageReader.readLine()) != null ) {
            page += oneLine + "\n";
        }
        return page;
    }
```

Dates, along with their Oracle format strings, are stored inside `OracleDate` objects. `OracleDate` is an "inner class", defined inside the StockUpdater class. Because it is a private class, it can't be seen or used outside of StockUpdater. Later, if we think `OracleDate` will be useful for other programmers, we can turn it into a public class by moving the definition to a file of its own.

```
    /** A class which represents Oracle timestamps. */
    private static class OracleDate {
        /** A string representation of the date */
        private String date;
        /** The date format, in Oracle's notation */
        private String dateFormat;

        /** Methods for accessing the date and the format */
        String getDate() { return date; }
        String getDateFormat() { return dateFormat; }
        void setDate( String newDate ) { date = newDate; }
        void setDateFormat( String newFormat ) { dateFormat = newFormat; }

        /** A constructor that builds a new OracleDate */
        OracleDate( String newDate, String newFormat ) {
            setDate( newDate );
            setDateFormat( newFormat );
        }
    }
```

To extract the dates from the Web page, we have a couple of routines called `matchFetchDate` and `matchQuoteDate`:

```
    /** Search through text from a Yahoo quote page to find a date stamp */
    private static OracleDate matchFetchDate( String text ) {
        Perl5Util regexp = new Perl5Util();

        if ( regexp.match("/<p>\\s*(\\S+\\s+\\S+\\s+\\d+\\s+\\d\\d\\d\\d)\\s+[0-
9:]+[aApP][mM][^<]*<table>/", text) ) {
            return new OracleDate( regexp.group(1), "Day, Month DD YYYY" );

        } else {
            return null;
        }
    }


    /** Search through the time column from a single Yahoo stock quote
        and set the time accordingly.  */
    private static OracleDate matchQuoteDate( String timeText, OracleDate fetchDate
) {
        Perl5Util regexp = new Perl5Util();

        if ( regexp.match("/\\d?\\d:\\d\\d[aApP][mM]/", timeText) ) {
            // When the time column of the stock quote doesn't include the day,
            // the day is pulled from the given fetchDate.
            String date = fetchDate.getDate() + " " + timeText;
            String format = fetchDate.getDateFormat() + " HH:MIam";
            return new OracleDate( date, format );

        } else if ( regexp.match("/[A-Za-z]+ +\\d\\d?/", timeText )) {
            // After midnight but before the market opens, Yahoo reports the date
            // rather than the time.
            return new OracleDate( timeText, "Mon DD" );

        } else {
```

```
            return null;
        }
    }
```

The stock prices coming back from Yahoo often contain fractions, which have special HTML markup. The `parseFraction` method pulls the HTML apart and returns the stock price as a Java `float`:

```
    /** Convert some HTML from the Yahoo quotes page to a float, handling
        fractions if necessary */
    private static float parseFraction( String s ) {
        Perl5Util regexp = new Perl5Util();
        if ( regexp.match( "/^\\D+(\\d+)\\s*<sup>(\\d*)</sup>/<sub>(\\d*)</sub>/",
s)) {
            // There's a fraction
            float whole_num = Float.parseFloat( regexp.group(1) );
            float numerator = Float.parseFloat( regexp.group(2) );
            float denominator = Float.parseFloat( regexp.group(3) );
            return whole_num + numerator / denominator;

        } else {
            // There is no fraction
            // strip the HTML and go
            return Float.parseFloat( regexp.substitute( "s/<.*?>//g", s) );
        }
    }
```

Odds and Ends

All of our methods obtain their JDBC connections by calling `getConnection()`. By routing all database connection requests through this method, our class will be able to run either inside or outside the database - `getConnection` checks its environment and sets up the connection accordingly. Loading Java into Oracle is a tedious process, so it's nice to be able to debug your code from an external JVM.

```
    public static Connection getConnection()
    throws SQLException {

        Connection conn;

        // In a real program all of these constants should
        // be pulled from a properties file:
        String driverClass = "oracle.jdbc.driver.OracleDriver";
        String connectString = "jdbc:oracle:oci8:@ora8i_ipc";
        String databaseUser = "username";
        String databasePassword = "password";

        try {
            // Figure out what environment we're running in
            if ( System.getProperty("oracle.jserver.version") == null ) {
                // We're not running inside Oracle
                DriverManager.registerDriver( (java.sql.Driver)
Class.forName(driverClass).newInstance() );
                conn = DriverManager.getConnection( connectString, databaseUser,
databasePassword );

            } else {
                // We're running inside Oracle
```

```
            conn = DriverManager.getConnection( "jdbc:default:connection:" );
        }

        // The Oracle JVM automatically has autocommit=false,
        // and we want to be consistent with this if we're in an external JVM
        conn.setAutoCommit( false );

        return conn;

    } catch ( Exception e ) {
        throw new SQLException( e.toString() );
    }
}
```

To call `StockUpdater` from the command line, we also need to provide a `main` method:

```
/** This method allows us to call the class from the command line
 */
public static void main(String[] args)
throws SQLException {
    if ( args.length == 1 ) {
        checkAll( Double.parseDouble( args[0] ));
    } else if ( args.length == 2 ) {
        checkOne( Double.parseDouble(args[0]), args[1] );
    } else {
        System.out.println("Usage: java StockUpdater update_interval
[stock_ticker]");
    }
}
```

Finally, the fetcher needs a utility which can join strings together. It's similar to the "join" command in Perl or Tcl.

```
/** Builds a single string by taking a list of strings
    and sticking them together with the given separator.
    If any of the elements of the list is not a String,
    an empty string in inserted in place of that element.
*/
public static String joinList( List stringList, String separator ) {

    StringBuffer joinedStr = new StringBuffer();

    Iterator iter = stringList.iterator();
    boolean firstItemP = true;
    while ( iter.hasNext() ) {
        if ( firstItemP ) {
            firstItemP = false;
        } else {
            joinedStr.append( separator );
        }

        Object s = iter.next();
        if ( s != null && s instanceof String ) {
            joinedStr.append( (String) s );
        }
    }
    return joinedStr.toString();
}
```

```
} // End of the StockUpdater class
```

A Foreign Table In Action

Now that we've implemented all of this, we can take a look at our foreign table in SQL*Plus:

```
SQL> select ticker_symbol, last_trade,
     (sysdate - last_modified)*24 as hours_old
     from stock_quotes;

TICKER_SYMBOL       LAST_TRADE  HOURS_OLD
------------------- ---------- ----------
AAPL                   22.9375 3.62694444
IBM                    112.438 3.62694444
MSFT                     55.25 3.62694444
```

This data is over three hours old. Let's request data from within the last hour:

```
SQL> call checker.stock_quotes( 1/24 );

Call completed.

SQL> select ticker_symbol, last_trade,
     (sysdate - last_modified)*24 as hours_old
     from stock_quotes;

TICKER_SYMBOL       LAST_TRADE  HOURS_OLD
------------------- ---------- ----------
AAPL                    23.625 .016666667
IBM                    114.375 .016666667
MSFT                   55.4375 .016666667
```

That's better. But I'm curious about the Intel Corporation.

```
SQL> call checker.stock_quotes( 1/24, 'INTC' );

Call completed.

SQL> select ticker_symbol, last_trade,
     (sysdate - last_modified)*24 as hours_old
     from stock_quotes;

TICKER_SYMBOL       LAST_TRADE  HOURS_OLD
------------------- ---------- ----------
AAPL                    23.625 .156666667
IBM                    114.375 .156666667
MSFT                   55.4375 .156666667
INTC                        42 .002777778
```

Reference

- The pioneering work in this area was done in the mid-1990s by the Garlic project at IBM Almaden Research Center. This is the same laboratory that developed System R, the first relational database management system, which eventually became IBM's DB2 product. To see how IBM's ideas unfolded in the area of wrapping legacy databases and foreign Web sites, visit http://www.almaden.ibm.com/cs/garlic/.

- Java Stored Procedures Developer's Guide at http://www.oradoc.com/keyword/java_stored_procedures.
- PL/SQL User's Guide and Reference at http://www.oradoc.com/keyword/plsql.
- Sun's Java2 Documentation at http://java.sun.com/j2se/1.3/docs/index.html.

Afterword

part of SQL for Web Nerds by Philip Greenspun

Congratulations: You've learned SQL. You can tap into the power of the relational database management system for concurrency control and transaction management. This power is useful for almost any activity in which computers are employed.

If you aren't already saddled with a job, I encourage you to think about building Web services for collaboration, which I think are the most valuable things that we nerds can give to society. To that end, I wrote *Philip and Alex's Guide to Web Publishing*, available at http://photo.net/wtr/thebook/.

Appendix A: Setting up your own RDBMS

by Philip Greenspun, part of SQL for Web Nerds

This book was written for students at MIT who have access to our Web/db development systems. The second category of reader who could use this book painlessly is the corporate slave who works at Bloatco, Inc. where they have a professionally-maintained Oracle server. If you are unfortunate enough to fall outside of those categories, you might need to install and maintain your own RDBMS. This appendix is intended to help you choose and run an RDBMS that will be good for learning SQL and that will let you grow into running production Web sites.

Choosing an RDBMS Vendor (Quick and Dirty)

The quick and dirty way to choose a database management system is to start from a list of products that seem viable in the long term. Basically you can choose from among three:

- Microsoft SQL Server (popularity maintained by Microsoft's overall market power)
- Oracle (you won't get fired)
- PostgreSQL (free open-source)

If want to ignore the RDBMS and concentrate your energy on attacking higher-level application challenges, Oracle is the best choice. It is the most powerful and feature-rich RDBMS. You can run the same software on a $500 PC or in a $5 million multiply redundant server configuration.

PostgreSQL is an interesting alternative. It is free and open-source. Like Oracle, it has optimistic locking (writers need not wait for readers; readers need not wait for writers). PostgreSQL can be easier to install and maintain than Oracle. PostgreSQL was built from the ground up as an object-relational database and offers some important features that Oracle still lacks. The support situation for PostgreSQL continues to improve, with the latest important event being the formation of GreatBridge (www.greatbridge.com) in May 2000 funded with $25 million of venture capital. Bottom line: if you pick PostgreSQL and your site grows large and important, you'll have a tough time answering the business folks' concerns, e.g., "How exactly can we make sure that this site won't go down if the computer running the DBMS fails?"

Microsoft SQL Server is an uninteresting alternative. Microsoft started with the source code from Sybase and has gradually improved the product. The system tends to be unsuitable for Web use because of its traditional pessimistic locking architecture. If you hire a new programmer and he or she executes a slow-to-return query, users won't be able to update information, place orders, or make comments until the query completes. In theory the management of these locks can be manually adjusted but in practice Web programmers never have the time, ability, or inclination to manage locks properly. SQL Server is generally far behind Oracle in terms of features, e.g., the ability to run Java inside the database, SQL extensions that are convenient for data warehousing, or layered products that help organizations with extreme performance or reliability demands. All of this said, SQL Server probably won't disappear because Microsoft has so much power in a large portion of the server world. So if you're part of an organization that is 100% Microsoft and people are already skilled at maintaining SQL Server, it is a reasonable technical decision to continue to use it.

Choosing an RDBMS Vendor (From First Principles)

Here are the factors that we think 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. ease of running standard programming languages internally
6. support

Cost/Complexity to Administer

Sloppy RDBMS administration is one of the most common causes of downtime at sophisticated sites. If you don't have an experienced staff of database administrators to devote to your site, you should consider either outsourcing database administration or running a simple RDBMS such as PostgreSQL.

Lock Management System

Relational database management systems exist to support concurrent users. If you didn't have 100 people simultaneously updating information, you'd probably be better off with a Perl script 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 is simple to implement, works great in the research lab, and can be proven correct mathematically. The only problem with it? It doesn't work. Sometimes it doesn't work because of a bug. A particular RDBMS's implementation of this scheme get confused and stuck when there are a bunch of users. More often it doesn't work because pessimistic locking *is* a bug. A programmer writes an hour-long back-end query and forgets that by doing so he or she will cause every updating page on the Web site to wait for the full hour.

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. Oracle and PostgreSQL offer it.

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 8.1 with ConText for "rows relating to 'running' or its synonyms", the SQL syntax will not be useful for asking the same question of Microsoft SQL Server 7.0 with its corresponding full-text indexing option.

My best experiences have been with the Illustra/PLS combination. I fed it 500 short classified ads for photography equipment then asked "What word is most related to *Nikon*". The answer according to Illustra/PLS: *Nikkor* (Nikon's brand name for lenses).

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 be sure how long a user's classified ad or bulletin board posting is going to be. Modern database vendors typically provide a character large object (CLOB) data type. A CLOB theoretically allows you to store arbitrarily large data. However, in practice there are so many restrictions on a CLOB column that it isn't very useful. For example, with Oracle 8i you can't use a CLOB in a SQL WHERE clause and thus the preceding "LIKE '%dogs%'" would fail. You can't build a standard index on a LOB column. You may also have a hard time getting strings into or out of a LOB. The Oracle SQL parser only accepts string literals up to 4,000 characters in length. After that, you'll have to use special C API calls. LOBs will give your Oracle database

administrator fits: they break the semantics of EXPORT and IMPORT. At least as of Oracle 8.1.6, if you export a database containing LOBs you won't be able to import it to another Oracle installation unless that installation happens to have a tablespace with the same name as the one where the LOBs were stored in the exported installation.

PostgreSQL has a "text" data type that theoretically has no limit. However, an entire PostgreSQL row must be no longer than 8,000 characters. So in practice PostgreSQL is less powerful than Oracle in this respect.

*** research Microsoft SQL Server but last I checked it was 255 characters! *****

Caveat emptor.

Ease of Running Standard Programming Languages Internally

Within Oracle it is fairly easy to run Java and the quasi-standard PL/SQL. Within PostgreSQL it is fairly easy to run Perl, Tcl, and a sort-of-PL/SQL-like PL/pgSQL. Within Microsoft SQL Server ***** (research this).

Support

In theory you won't be calling for support very often but you want to make sure that when you do it is to an organization that takes RDBMS reliability and uptime very seriously.

Paying an RDBMS Vendor

"PostgreSQL is available without cost," is the opening to Chapter 1 of the PostgreSQL documentation. Microsoft has the second easiest-to-figure-out pricing: visit http://www.microsoft.com/sql/ and click on "pricing and licensing". The price in 1998 was $4400 for software that could be used on a 4-CPU machine sitting behind a Web site. As of September 2000 they were charging either $20,000 or $80,000 for a 4-CPU Web server, depending on whether you wanted "enterprise" or "standard" edition.

Despite its industrial heritage, Oracle can be much cheaper than Microsoft. Microsoft charges $500 for a crippled developer edition of SQL Server; Oracle lets developers download the real thing for free from technet.oracle.com. Microsoft wants $20,000 per CPU; Oracle negotiates the best deal that they can get but lately has been selling startups a "garage" license for $10,000 for two years.

Performance

Be assured that any RDBMS product will be plenty slow. We once 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 from a custom C application. It took about 20 minutes (60 rows/second). By using SQL*Loader we probably could have approached 1000 rows/second but that still would have been 70 times slower than the Perl script. Providing application programmers with the ACID guarantees is always going to be slow.

There are several ways to achieve high performance. If most of your activity is queries, you could start by buying a huge multi-processor computer with enough RAM to hold your entire database at once. Unfortunately, if you are paying by the CPU, your RDBMS vendor will probably give your bank account a reaming that it will not soon forget. And if you are processing a lot of INSERTs and UPDATEs, all those CPUs bristling with RAM won't help you. The 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. It would be best to avoid ever going to disk as we did in the case of SELECTs by buying up enough RAM to hold the entire data set. 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.

Oracle manages to process more transactions per second than a disk's writes/second capacity. What the DBMS does is batch up transactions that come in at roughly the same time from different users. It writes enough to disk to make them all durable and then returns to those users all at once.

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 *Oracle8i DBA Handbook* (Loney and Theriault; 1999) 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 the *Oracle8 DBA Handbook*'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 simple 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.

Don't forget to back up

Be afraid. Be very afraid. Standard Unix or Windows NT 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 Unix 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 introduction of frames and failure to include WIDTH and HEIGHT tags on IMGs. 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, Postgres, or 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 NT 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 is the shell command that will export a snapshot of an Oracle database into a dump file:

```
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. If you read *Oracle Performance Tuning* (Gurry and Corrigan 1996; O'Reilly), you'll find some dark warnings that you *must* export periodically in order to flush out cases where Oracle has corrupted its internal data structures. Another good reason to export is that periodically dropping all of your tables and importing them is a great way to defragment data. At ArsDigita we export every customer's Oracle database every night, except the handful of customers with terabytes of data.

What if your database is too large to be exported to a disk and can't be taken offline? Here's a technique practiced by a lot of experienced IT groups:

- 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. Some organizations have three mirrors. They can pull pull off one set of physical disks and back them up without running the risk that a drive failure during the backup window will take the database management system offline.

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.

If all of this sounds rather tedious just to ensure that your data are still around tomorrow, you might be cheered by the knowledge that Oracle dbas are always in high demand and start at $60,000 to $80,000 a year. When the Web bubble bursts and your friends who are "HTML programmers" are singing in the subway, you'll be kicking back at some huge financial services firm.

We'll close by quoting Perrin Harkins. A participant in the Web/db question-and-answer forum (http://www.arsdigita.com/bboard/q-and-a.tcl?topic=web/db) asked whether caching database 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)."

Reference

- Oracle8i Server Backup and Recovery Guide, part of the Oracle doc set
- Oracle Server Administrator's Guide, part of the Oracle doc set
- *Oracle 8i Backup and Recovery* (Velpuri 2000; Oracle Press)

Appendix B: Getting Real Work Done with Oracle

by Philip Greenspun, part of SQL for Web Nerds

Consider John Q. Nerd with his bona fide SQL expertise and MIT PhD. How much useful work can John get out of Oracle? None. John Q. Nerd only knows how to drive Oracle from SQL*Plus. This appendix covers the little details such as getting data into or out of Oracle. It is organized as an FAQ.

How do I get data into Oracle?

- Answer 1: Start up SQL*Plus. Type really fast.
- Answer 2: If you need to load 1000+ rows per second, a fairly common requirement in the data warehousing world, read the Oracle8 Server Utilities, SQL*Loader section.
- Answer 3: If SQL*Loader makes you feel like ripping out what's left of your hair, go with Perl DBD/DBI (a module available from http://www.cpan.org; explanation available in *Advanced Perl Programming* (Srinivasan 1997; O'Reilly).
- Answer 4: If your data are imprisoned in another RDBMS, consider using a fancy GUI tool such as Data Junction (http://www.datajunction.com). These are PC-based applications that open simultaneous connections to your Oracle database and some other RDBMS. They can generally show you what's available in the other database and drag whatever you think necessary into Oracle.
- Answer 5: If your data are in another Oracle database, read up in the Oracle Server Utilities manual about `exp` and `imp` (http://www.oradoc.com/keyword/export_import).
- Answer 6: If your data are coming in from an email message, configure your mailer to fork a Perl DBD/DBI script that will then parse out the content from the headers and other such crud, open up Oracle, and then send an insert statement. Due to the fork and the opening of Oracle, this is 1/100th the efficiency of a Web script running in a threaded Web server with database connection pooling. However, you probably won't be getting 40 email messages per second so it doesn't really matter. The free open-source ArsDigita Community System includes a Perl script that you could use as a starting point. See http://www.arsdigita.com/doc/email-handler for details.

How do I get data out of Oracle?

- If you want to publish data on the Web, look at the approaches articulated in *Philip and Alex's Guide to Web Publishing*, Chapter 13 (http://www.arsdigita.com/books/panda/databases-interfacing).

- If you are sending data to another Oracle installation, use `exp`, documented in the Oracle8 Server Utilities manual, `exp` and `imp` section.
