;login:

THE MAGAZINE OF USENIX & SAGE
October 2002 volume 27 • number 5

inside:

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

practical perl

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long- time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

Lightweight Databases

Many programs need to store some kind of state information between sessions. What's the best way to maintain this data? It depends on the application, of course. This month, I investigate solutions that are easy to use and that fit somewhere in between text files and relational database servers.

Managing Persistent Data

A few months ago, I started writing a program to monitor the online catalogs of several technical publishers periodically. Every time my program would visit a Web site, it would look for recently added book titles. Some of these publishers maintain extensive online catalogs, and as a good Web citizen, I certainly don't want to overload their servers with requests for information my program has already seen. Furthermore, I wanted to highlight new titles to see what I might be interested in reading.

Obviously, my program would need to store some state on disk describing what links had been seen before. However, there are literally dozens of ways to accomplish this, and it's not entirely obvious which one is best.

The lazy programmer's solution would be to write out some flat text files on exit that would be read in the next time my program runs. In this case, flat files would be adequate, if the saved data were relatively simple, such as a list of links stored one per line. However, if I need to save more data (ISBN, title, author, etc.), then other issues arise. For example, I would need to synchronize the functions to read and write to my datafile, to make sure they use the exact same format for both input and output. This might become a little tricky if I need to upgrade my program to store even more information later on.

Another perfectly valid approach is to start out using a relational database engine, such as MySQL, PostgreSQL, or Oracle. This is generally a good choice, except in this situation, using a relational database server seemed like an overengineered solution. I didn't want to get bogged down with details of setting up databases, users, or passwords – I just wanted to write a simple Web crawler and save some data once my program finished. One issue I particularly wanted to avoid was having a program that magically breaks whenever a database server is moved, or when a database user or password changes.

In the end, I found that this simple little program fit into a sweet spot – somewhere between quick-and-dirty flat text files and full relational database servers. Many little programs I've written (most of them just quick hacks) fall into this category. One benefit of using Perl is that I'm not stuck using an inappropriate technology for my problem, whether that technology is overengineered or underengineered.

Of course, in Perl there is more than one way to solve this problem. In this article I want to examine two main types of solutions: the venerable DBM file, and lightweight relational databases.

Persistence Through DBM Files

The classic solution to this data storage problem is the venerable DBM file. DBM files come in many different forms, and all of them can be used to store simple key/value pairs. In this way, DBM files behave much like Perl hash variables, except that the keys and values can be saved and restored for later use.

Using a DBM file is quite simple and requires only a few lines of code to start:

```
#!/usr/bin/perl -w

use AnyDBM_File;
use Fcntl;

my %urls;

tie %urls, "AnyDBM_File", "url_data", O_RDWR | O_CREAT, 0640;

## ... %urls is transparently connected to the file "url_data.db" ...
```

First, we need to load a DBM library. In this case, I loaded the AnyDBM_File library through the use statement on line 3. I then create a new hash variable, %urls, on line 6, and connect that hash to the DBM file url_data.db with the tie statement on line 8. (The default DBM file implementation on my machine adds the .db suffix automatically.) From this point forward, any keys or values that are added or modified to the %urls hash in my program will also be stored on disk in the file url_data.db. The connection will be broken either when my program finishes or when I execute the statement untie %urls;.

All of the magic occurs in the tie statement. This tells Perl to associate the variable %urls with the package AnyDBM_File.

The other parameters are sent to AnyDBM_File to describe the file we wish to use. Here, the parameters are a portion of the filename we'll be using (url_data), the flags used to open the file (O_RDWR | O_CREAT, values that come from the Fcntl module), and the permissions mode (0640, or read/write for the owner, read only for the group).

For my program to monitor online publisher catalogs, I could add a new key to this hash for each URL I process, with the value being the day it was processed. By checking to see if an entry already exists for a particular URL, I can easily identify which URLs are new:

```
foreach (@links) {
      next if defined $urls{$_}; ## We've seen this URL before

      print "New URL: $_\n";
      $urls{$_} = localtime();
}
```

And that's it. The first time I run my little link checker, I'll see a whole slew of URLs fly by. Starting with the second time I run my program, I'll only see the URLs that have been added since the previous run.

Flavors of DBM Files

Using the AnyDBM_File module is guaranteed to work whenever a new DBM file is created, but it does have some problems. Depending on the configuration of your system, Perl will support some of the various implementations of DBM files, including NDBM, Berkeley DB, GDBM, and SDBM. By using AnyDBM, you tell Perl that you don't care which one to use, any one of them is fine. The main problem with AnyDBM is that it is not guaranteed to use the same implementation on two different machines, nor is it guaranteed to open any random DBM file you happen to have. It is quite possible that AnyDBM_File will load the NDBM_File module when you want to open a file created by DB_File or GDBM_File. This operation will fail because the naming conventions or the file formats differ.

Therefore, it is better to choose a specific type of DBM file module instead of the AnyDBM_File:

- NDBM_File uses the native NDBM library on your system, if one exists.
- DB_File uses the Berkeley DB 1.85 library, if present.
- GDBM_File uses the GNU GDBM library, if present.
- SDBM_File is Perl's own DBM library and is always available.

Each of these DBM libraries has its own advantages and disadvantages; see the documentation for AnyDBM_File for more information (man AnyDBM_File or perldoc AnyDBM_File). I use either DB_File or GDBM_File, because they tend to be available on most Perl installations. SDBM_File will always work, and it exists as a DBM implementation of last resort.

Remember that DBM files store simple key/value pairs. If you are programming multi-level data structures, such as a hash of hashes or a hash of lists, then regular DBM files will not store all of your data properly. For these kinds of data structures, look into Joshua Chamas' "multi-level DBM" module, MLDBM, available on the Comprehensive Perl Archive Network (CPAN).

Limits to DBM Files

Perl's support for DBM files makes it easy to add persistent data structures to a program with just a few lines of code. The main disadvantage is that you need to manage all of the data yourself, using Perl hashes. This may be a useful technique in the small, but tends not to scale very well as requirements grow.

Suppose I wanted to create a report program to count URLs, grouped by the day they were first encountered. Using DBM files, that code might look something like this program:

```
#!/usr/bin/perl -w

use strict;

use DB_File;
use Fcntl;

## Load in the cache of URL => date values
my %url_dates;
tie %url_dates, "DB_File", "url_dates", O_RDWR | O_CREAT, 0640;

## Count books (hash entries), grouped by the day they were found
my %count_by_day;
foreach (values %url_dates) {
      ## Strip out the time component of the date
      ## "Sun Aug 11 13:18:59 2002"  -> "Sun Aug 11 2002"
      my $date = $_;
      $date =~ s/\d{2}:\d{2}:\d{2} //;

      $count_by_day{$date}++;
}

## Print out the results (unsorted)
my ($date, $count);
while (($date, $count) = each %count_by_day) {
      print "$date:$count\n";
}
```

This small program re-uses the existing DBM file created by my Web-crawling program that finds new links. Note that this "little" program is 28 lines long (with whitespace and comments). More interesting reports, like one that counts books that contain the word "Perl" in the title, would require more data and might actually be significantly more involved. Now, imagine that two, three, or more of these reports become useful. All of a sudden, the quick-and-dirty solution is starting to run out of steam, since each new report might require a few dozen lines of new code.

It's clear that DBM files, while useful in some circumstances, aren't always the best or the simplest solution available.

Lightweight Relational Databases

As the requirements for my quick little book-catalog program slowly grow, it's clear that a SQL database is the most appropriate solution, especially if I intend to perform multiple queries on this data. Remember that the problems I intentionally want to avoid are some of the administrative details of setting up databases and passwords with a database engine like MySQL or PostgreSQL. That is, I want my program to "just work," and not be impacted if I happen to move my MySQL server to another computer, convert to PostgreSQL, or change a username or password. Additionally, I want my program to "just work" if I move it to another computer, without requiring that a particular database engine be installed to run this little hack.

Again, we're using Perl, so there's more than one way to do it.

Two ready-to-use modules are available on CPAN that meet my requirements. The first is Jeff Zucker's DBD::CSV module, and the second is Matt Sergeant's DBD::SQLite module. Both of these are database drivers that work with Perl's DBI module, Perl's generic interface to many different database engines. DBD::CSV simulates a relational database by using text files with comma-separated values for each table in the database. DBD::SQLite contains a full-fledged relational database engine written in C that's embedded in the database driver module itself. Neither of these modules require setup, configuration, or a server process to manage the database. They just work.

If you're already familiar with using DBI to connect to MySQL or other relational databases, there is nothing new to learn here. Furthermore, should you need to upgrade from a CSV or a SQLite database, all you need to do is change the DBI connection string, and possibly some of your SQL statements – the rest of your Perl programs remain unchanged.

I used to recommend and use DBD::CSV when I wanted to create a lightweight relational database. Once Matt released his DBD::SQLite module, I started using that instead, since it contains a more robust database engine. This is mostly due to the hard work of Richard Hipp, who created SQLite as a full-featured, embeddable relational database, complete with indexes, transactions, and multiuser access.

Creating a Perl program that uses SQLite is straightforward, assuming you're already familiar with DBI and SQL (another issue entirely):

```
#!/usr/bin/perl -w

use strict;
use DBI;

my $dbname = "url_dates.db";
my $dbh = DBI->connect("dbi:SQLite:dbname=$dbname");

## ... use this SQLite database just like any other DBI database ...
```

One interesting feature of SQLite is that its columns are generally typeless. The column types that are declared in a CREATE TABLE statement are ignored (with the exception of integer primary keys), so there is no need to worry about losing data when storing a 30-character string in a column declared to be of type CHAR(25), or getting an error when storing a string value in an INTEGER column.

Using a relational database makes reporting much easier. For example, a program to count all URLs in the database, grouped by date, would be much simpler than the DBM version seen above:

```
#!/usr/bin/perl -w

use strict;
use DBI;

my $dbh = DBI->connect("dbi:SQLite:dbname=url_dates.db");

my $stmt = $dbh->prepare("SELECT day, COUNT(day)
               FROM urls GROUP BY day");

$stmt->execute();
while (my @row = $stmt->fetchrow_array()) {
      print join(": ", @row), "\n";
}
```

This program is half the size of my previous report program, and all of the logic for this report is contained in the SQL statement on line 4. The while loop at the bottom is reasonably generic and can be abstracted out into a separate sub. It would also be relatively easy to add another SQL query to count the number of books that contain "Perl" in the title – something that would have required more than one extra line of code in the DBM version of the program.

Conclusion

Maintaining persistent data is a common task in Perl programs, and there are easily dozens of ways to do it. For the truly simple tasks, Perl makes simple DBM files available easily and transparently. For more complicated tasks, the easiest solution tends to involve using the DBI, along with a suitable database engine, whether that's something big and powerful, or something small and easy to set up.