CHAPTER 17

Databases Overview

What's a database? We can use pretty much anything as a database, as long as it allows us to store our data and retrieve it later. There are many different kinds of databases. Some allow us to store data and retrieve it years later; others are capable of preserving data only while there is an electricity supply. Some databases are designed for fast searches, others for fast insertions. Some databases are very easy to use, while some are very complicated (you may even have to learn a whole language to know how to operate them). There are also large price differences.

When we choose a database for our application, we first need to define the requirements in detail (this is known as a *specification*). If the application is for short-term use, we probably aren't going to use an expensive, advanced database. A quick-and-dirty hack may do. If, on the other hand, we design a system for long-term use, it makes sense to take the time to find the ideal database implementation.

Databases can be of two kinds: volatile and non-volatile. These two concepts pretty much relate to the two kinds of computer memory: RAM-style memory, which usually loses all its contents when the electricity supply is cut off; and magnetic (or optical) memory, such as hard disks and compact discs, which can retain the information even without power.

Volatile Databases

We use volatile databases all the time, even if we don't think about them as real databases. These databases are usually just part of the programs we run.

In-Memory Databases in a Single Process

If, for example, we want to store the number of Perl objects that exist in our program's data, we can use a variable as a volatile database:

```
package Book::ObjectCounter;
use strict;
```

```
my $object_count = 0;
sub new {
    my $class = shift;
    $object_count++;
    return bless {}, $class;
}
sub DESTROY {
    $object_count--;
}
```

In this example, `$object_count` serves as a database—it stores the number of currently available objects. When a new object is created this variable increments its value, and when an object gets destroyed the value is decremented.

Now imagine a server, such as mod_perl, where the process can run for months or even years without quitting. Doing this kind of accounting is perfectly suited for the purpose, for if the process quits, all objects are lost anyway, and we probably won't care how many of them were alive when the process terminated.

Here is another example:

```
$DNS_CACHE{$dns} ||= dns_resolve($dns);
print "Hostname $dns has $DNS_CACHE{$dns} IP\n";
```

This little code snippet takes the hostname stored in `$dns` and checks whether we have the corresponding IP address cached in `%DNS_CACHE`. If not, it resolves it and caches it for later reuse. At the end, it prints out both the hostname and the corresponding IP address.

`%DNS_CACHE` satisfies our definition of a database. It's a volatile database, since when the program quits the data disappears. When a mod_perl process quits, the cache is lost, but there is a good chance that we won't regret the loss, since we might want to cache only the latest IP addresses anyway. Now if we want to turn this cache into a non-volatile database, we just need to tie `%DNS_CACHE` to a DBM file, and we will have a permanent database. We will talk about Database Management (DBM) files in Chapter 19.

In Chapter 18, we will show how you can benefit from this kind of in-process database under mod_perl. We will also show how during a single request different handlers can share data and how data can persist across many requests.

In-Memory Databases Across Multiple Processes

Sharing results is more efficient than having each child potentially waste a lot of time generating redundant data. On the other hand, the information may not be important enough, or have sufficient long-term value, to merit being stored on disk. In this scenario, Inter-Process Communication (IPC) is a useful tool to have around.

This topic is non-specific to mod_perl and big enough to fill several books on its own. A non-exhaustive list of the modules to look at includes `IPC::SysV`, `IPC::Shareable`, `IPC::Semaphore`, `IPC::ShareLite`, `Apache::Session`, and `Cache::Cache`. And of course make sure to read the `perlipc` manpage. Also refer to the books listed in the reference section at the end of this chapter.

Non-Volatile Databases

Some information is so important that you cannot afford to lose it. Consider the name and password for authenticating users. If a person registers at a site that charges a subscription fee, it would be unfortunate if his subscription details were lost the next time the web server was restarted. In this case, the information must be stored in a non-volatile way, and that usually means on disk. Several options are available, ranging from flat files to DBM files to fully-fledged relational databases. Which one you choose will depend on a number of factors, including:

- The size of each record and the volume of the data to be stored
- The number of concurrent accesses (to the server or even to the same data)
- Data complexity (do all the records fit into one row, or are there relations between different kinds of record?)
- Budget (some database implementations are great but very expensive)
- Failover and backup strategies (how important it is to avoid downtime, how soon the data must be restored in the case of a system failure)

Flat-File Databases

If we have a small amount of data, sometimes the easiest technique is to just write this data in a text file. For example, if we have a few records with a fixed number of fields we can store them in a file, having one record per row and separating the fields with a delimiter. For example:

```
Eric|Cholet|cholet@logilune.com
Doug|MacEachern|dougm@pobox.com
Stas|Bekman|stas@stason.org
```

As long as we have just a few records, we can quickly insert, edit, and remove records by reading the flat-file database line by line and adjusting things as required. We can retrieve the fields easily by using the `split` function:

```
@fields = split /\|/, $record;
```

and we can put them back using `join`:

```
$record = join '|', @fields;
```

However, we must make sure that no field uses the field separator we have chosen (| in this case), and we must lock the file if it is to be used in a multiprocess environment where many processes may try to modify the same file simultaneously. This is the case whether we are using mod_perl or not.

If we are using some flavor of Unix, the */etc/passwd* file is a perfect example of a flat-file database, since it has a fixed number of fields and most systems have a relatively small number of users.* This is an example of such a file:

```
root:x:0:0:root:/root:/bin/tcsh
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
adm:x:3:4:adm:/var/adm:
lp:x:4:7:lp:/var/spool/lpd:
```

`:` is used to separate the various fields.

Working with flat-file databases is easy and straightforward in plain Perl. There are no special mod_perl tricks involved.

Filesystem Databases

Many people don't realize that in some cases, the filesystem can serve perfectly well as a database. In fact, you are probably using this kind of database every day on your PC—for example, if you store your MP3 files categorized by genres, artists, and albums. If we run:

```
panic% cd /data/mp3
panic% find .
```

We can see all the MP3 files that we have under */data/mp3*:

```
./Rock/Bjork/MTV Unplugged/01 - Human Behaviour.mp3
./Rock/Bjork/MTV Unplugged/02 - One Day.mp3
./Rock/Bjork/MTV Unplugged/03 - Come To Me.mp3
...
./Rock/Bjork/Europa/01 - Prologue.mp3
./Rock/Bjork/Europa/02 - Hunter.mp3
...
./Rock/Nirvana/MTV Unplugged/01 - About A Girl.mp3
./Rock/Nirvana/MTV Unplugged/02 - Come As You Are.mp3
...
./Jazz/Herbie Hancock/Head Hunters/01 - Chameleon.mp3
./Jazz/Herbie Hancock/Head Hunters/02 - Watermelon Man.mp3
```

Now if we want to query what artists we have in the Rock genre, we just need to list the files in the *Rock/* directory. Once we find out that Bjork is one of the artists in the Rock category, we can do another enquiry to find out what Bjork albums we have bought by listing the files under the *Rock/Bjork/* directory. Now if we want to see the

* Disregard the fact that the actual password is stored in */etc/shadow* on modern systems.

actual MP3 files from a particular album (e.g., *MTV Unplugged*), we list the files under that directory.

What if we want to find all the albums that have *MTV* in their names? We can use `ls` to give us all the albums and MP3 files:

```
panic% ls -l ./*/*/*MTV*
```

Of course, filesystem manipulation can be done from your Perl program.

Let's look at another example. If you run a site about rock groups, you might want to store images relating to different groups. Using the filesystem as a database is a perfect match. Chances are these images will be served to users via `<img>` tags, so it makes perfect sense to use the real path (`DocumentRoot` considerations aside) to the image. For example:

```
<img src="/images/rock/ACDC/cover-front.gif" alt="ACDC" ...>
<img src="/images/rock/ACDC/cover-back.gif"  alt="ACDC" ...>
```

In this example we treat *ACDC* as a record and *cover-front.gif* and *cover-back.gif* as fields. This database implementation, just like the flat-file database, has no special benefits under mod_perl, so we aren't going to expand on the idea, but it's worth keeping in mind.

DBM Databases

DBM databases are very similar to flat-file databases, but if all you need is to store the key/value pairs, they will do it much faster. Their use is much simpler, too. Perl uses `tie()` to interact with DBM databases, and you work with these files as with normal hash data structures. When you want to store a value, you just assign it to a hash tied to the DBM database, and to retrieve some data you just read from the hash.

A good example is session tracking: any user can connect to any of several mod_perl processes, and each process needs to be able to retrieve the session ID from any other process. With DBM this task is trivial. Every time a lookup is needed, tie the DBM file, get the shared lock, and look up the *session_id* there. Then retrieve the data and untie the database. Each time you want to update the session data, you tie the database, acquire an exclusive lock, update the data, and untie the database. It's probably not the fastest approach, and probably not the best one if you need to store more than a single scalar for each record, but it works quite well.

In Chapter 20 we give some important background information about DBM files and provide a few examples of how you can benefit from usingDBM files under mod_perl.

Too Many Files

There is one thing to beware of: in some operating systems, when too many files (or directories) are stored in a single directory, access can be sluggish. It depends on the filesystem you are using. If you have a few files, simple linear access will be good enough. Many filesystems employ hashing algorithms to store the i-nodes (files or directories) of a directory. You should check your filesystem documentation to see how it will behave under load.

If you find that you have lots of files to store and the filesystem implementation won't work too well for you, you can implement your own scheme by spreading the files into an extra layer or two of subdirectories. For example, if your filenames are numbers, you can use something like the following function:

```
my $dir = join "/", (split '', sprintf "%02d", $id)[0..1], $id;
```

So if you want to create a directory *12345*, it will be converted into *1/2/12345*. The directory *12* could become *0/0/12*, and *124* could become *0/1/124*. If your files have a reasonable distribution, which is often true with numerical data, you might end up with two-llevel hashing. So if you have 10,000 directories to create, each end-level directory will have at most about 100 subdirectories, which is probably good enough for a fast lookup. If you are going to have many more files you may need to think about adding more levels.

Also remember that the more levels you add, the more overhead you are adding, since the OS has to search through all the intermediate directories that you have added. Only do that if you really need to. If you aren't sure, and you start with a small number of directories, abstract the resolution of the directories so that in the future you can switch to a hashed implementation or add more levels to the existing one.

Relational Databases

Of course, the most advanced solution is a relational database. But even though it provides the best solution in many cases, it's not always the one you should pick. You don't need a sledgehammer to crack a nut, right?

Relational databases come in different implementations. Some are very expensive and provide many tools and extra features that aren't available with the cheaper and free implementations. What's important to keep in mind is that it's not necessarily the most expensive one that is the best choice in a given situation. Just as you need to choose the right database structure, you need to choose the right relational database. For example, ask yourself whether you need speed, or support for transactions, or both.

It makes sense to try to write your code in such a way that if later in the course of development you discover that your choice of relational database wasn't the best, it will be easy to switch to a different one.

mod_perl greatly helps work with relational databases, mainly because it allows persistent database connections. We'll talk extensively about relational databases and mod_perl in Chapter 20.

References

- Chapters 2 and 3 of the book *Programming the Perl DBI*, by Alligator Descartes and Tim Bunce (O'Reilly), provide a good overview of relational and nonrelational databases
- Chapter 10 of the book *Advanced Perl Programming*, by Sriram Srinivasan (O'Reilly), talks about persistence