13

Perl and Databases

Building database applications is without doubt one of the most common uses of Perl. With its excellent support for interfacing with a very broad range of database formats, this support comes two guises:

- For simpler applications, we've got the DBM (DataBase Manager) modules. DBM is a generic term for a family of simple database formats, they come in several different flavors, and you'll find them in common use, particularly on UNIX platforms. Perl supports all DBM formats through a tied hash, so that the contents of the database look to us just like a normal hash variable.
- For more advanced database applications, Perl provides the DataBase Interface (DBI) module. DBI is a generic database interface for communicating with database servers that use Structured Query Language (SQL) to get at data. To use DBI, we also need to install a database driver (DBD) module for the database we want to connect to. These are available for all popular databases, including MySQL, mSQL, Oracle, Informix, and Sybase.

DBI provides abstraction – we can write our code without paying too much attention to the database that sits behind it all. Meanwhile, the DBD module provides us with all the database-specific parts. By using DBI, we can take the same approach to all our programming, no matter what underlying database we happen to be using.

> *ODBC and remote database access are also available as DBD modules, so any database that supports ODBC can also be accessed by Perl using DBI.*

In this chapter, we're going to look at how Perl works with both of these. Since DBM is far simpler and requires no database server to be installed and configured, we'll look at it first. We'll then move on to DBI, covering a little basic SQL as we go, and see how we can use it to implement more advanced database applications.

Perl and DBM

DBM databases have existed on UNIX systems for many years, and since Perl draws on a lot of UNIX history, it's supported DBM databases virtually from day one. Once upon a time, Perl actively supported `dbmopen` and `dbmclose` functions for the purpose of interfacing with DBM files, they still exist in the language, but really only for reasons of backward compatibility with older Perl scripts. In these enlightened times, we can use the more powerful (and infinitely more convenient) tied hash interface.

DBM databases can best be thought of as basic card-file databases – they support a fairly simple form of key-value association that translates easily into a Perl hash when tied. DBM does not support indexes, binary trees (with the exception of Berkeley DB), complex record structures, multiple tables, or transactions, for any of these we'll need to use a proper database server, most likely via a DBI interface.

Which DBM Implementation To Use

There are five main DBM implementations, each supported by a C library. Each uses its own file format for the databases it creates (although some libraries, notably GDBM, support the ability to access databases created with a different DBM implementation). Since it's entirely possible that your operating system has no support for any native DBM format (Windows, for example) Perl provides us with the SDBM format as a fall-back option. The five are:

- **gdbm** – the GNU DBM database. The fastest and most portable of the standard DBM implementations (only Berkeley DB is faster). As well as its own storage format, it can read and write NDBM databases. Supports limited file and record locking – handy for concurrent user access. Freely downloadable under the GNU Public License (from www.gnu.org and almost every FTP repository on the planet).
- **ndbm** – the "new" DBM implementation. The version of DBM most commonly found on current UNIX systems. Not as powerful or feature-rich as GDBM, but it's good enough for most purposes if GDBM isn't available.
- **odbm** – the "old" DBM implementation. Also known as just "DBM". This is the version of DBM that originally appeared on UNIX systems. It's largely been replaced by NDBM and should be avoided if possible.
- **sdbm** – comes as standard with Perl. Not as efficient as the other DBM formats (especially GDBM). It's not well-suited to large databases, but is guaranteed to work anywhere that Perl can be installed, so it's useful for ensuring cross-platform portability.
- **bsd-db** – the "Berkeley" DB format. Not strictly a DBM database at all, but it can be considered a close relative and is frequently found on BSD Unix systems. Like GDBM, DB supports file and record locking. More powerful than any of the DBM implementations. Supports a binary tree format as well as DBM's simple hash format – both can be used by the DBM-like interface provided by Perl. You can get it from http://www.sleepycat.com/.

Perl can only support a given DBM format if the supporting libraries are actually installed on the system. When Perl is built, it scans the system and builds Perl module wrappers for all DBM file formats for which it can find libraries. Therefore, to use `GDBM`, we must first install the `GDBM` package (from www.gnu.org and many mirrors) and then Perl itself.

We'll largely assume the use of SDBM throughout this chapter, but all the examples should also work with the other implementations above. Bear in mind that SDBM isn't ideal, so where you have an option, you should probably consider using GDBM. Although most of the following examples specify `use SDBM`, you can easily adapt them to use any other DBM format by substituting the relevant module name.

And we're on the subject… Say you're running a script that wants to use `GDBM`, and it fails because it can't find the Perl module for `GDBM` support. The chances are, it's not because Perl was installed incorrectly, but simply that you didn't have `GDBM` handy when Perl was installed. Surely this presents a problem if we're trying to write portable Perl scripts?

Well, not necessarily. There's one more module we should at least mention, called `AnyDBM_File`. It's not actually DBM implementation itself, but as we'll see later on in the chapter, we can use it to avoid having to explicitly specify any particular implementation in our program.

Accessing DBM Databases

While the various DBM libraries use different formats internally, the way we access each of them is identical. In the past, we would use the (now obsolete) `dbmopen` and `dbmclose` functions to create a connection between our Perl program and a DBM database. These days we use `tie` to bind our chosen DBM implementation to a hash variable – we can then manipulate the variable, and in doing so, directly modify the data stored in the underlying database file. As we'll see, handling DBM databases from Perl is actually *really easy*.

Opening a DBM Database

As we mentioned above, DBM databases are accessed by using `tie` to associate them with a regular hash variable. Once tied, all accesses to the hash are invisibly translated into database reads, and all modifications, additions, or deletions are invisibly translated into database writes. This **tied hash** lets us maintain the database invisibly, just using ordinary Perl statements to manipulate the hash.

The `tie` statement for DBM files takes five parameters:

- the hash variable to be tied
- the DBM module providing the actual database
- the name of the database to tie to
- the file-access options
- the file-access mode

For now, let's assume we already have a DBM database, `demo.dbm` – you can get this sample file as part of the book's code download (available from www.wrox.com). Here's how we'd open it up for read-write access:

```
#!/usr/bin/perl
#opendbm.plx
use warnings;
use strict;
use POSIX;
use SDBM_File;           # or GDBM_File / NDBM_File / AnyDBM_File...
```

```
my %dbm;
my $db_file="demo.dbm";

tie %dbm, 'SDBM_File', $db_file, O_RDWR, 0;
```

Most of this is self-explanatory, with the exception of the last two arguments to `tie`:

- `O_RDWR` is a symbol imported from the `POSIX` module, which defines common labels for system values. In this case, we have specified the **open read-write** flag, telling perl that we want to open the file for both reading and writing.
- '0' specifies the **file permissions** we're using to open the database with. For now, this default value is fine. When we start to create databases, things become more interesting, as we'll see later.

Checking the State of a DBM Database

Just like any other system call, `tie` returns a true value if successful, so we should really say:

```
tie %dbm, 'SDBM_File', $db_file, O_RDWR, 0 or die "Error opening $db_file: $!\n";
```

Alternatively, we can check that the tie was successful with `tied`. If the hash is tied, the database was opened successfully. If not, it's because the `tie` failed and will have returned an error:

```
unless (tied %dbm) {
print "Database is not open - cannot continue!\n");
return;
} else {
# do stuff
}
```

It's also possible for `tie` to return a fatal error if we feed it parameters it doesn't like. We can trap such errors by placing an `eval` around the `tie` statement. `eval { BLOCK }` effectively says "try this out, but it may go wrong, so don't die if it does", any calls to `die` that originate from within the block won't kill the program. Instead, they'll be intercepted and the relevant error message placed in `$@`, from where we can access them as normal to provide an error message. All in all, it's a good way to cover yourself if you're undertaking a risky operation. However, it's also inherently unpredictable, and therefore worth taking extra special care with if you do use it:

```
eval {
tie %dbm, 'SDBM_File', $db_file, O_RDWR, 0;
};

if ($@) {
   print "Error tieing to $db_file: $@\n";
} elsif (!tied(%dbm)) {
print "Error opening $db_file: $!\n";
}
```

Creating DBM Databases

If a requested database doesn't exist, then the above example will return a file not found error. We can tell perl to create the database (if it doesn't already exist) by adding the `O_CREAT` (create) flag, which we can combine with `O_RDWR` using a bitwise `or`:

```
tie %dbm, 'SDBM_File', $db_file, O_CREAT|O_RDWR, 0644;
```

Because we're potentially creating the file, we specify a file mode in octal; `0644` specifies read and write access for us (`6`), but read-only access for other groups and users (`4`). Obviously, this only has any real meaning if the underlying operating system understands the concept of users and file permissions, but it's worth specifying for portability reasons. For more details on file modes, see Chapter 6, and `perldoc -f sysopen`.

Finally, here's how we could open a DBM database for read-only access. We could use this in a CGI script that's meant to read (but not modify) a database, thus making it more secure:

```
tie %dbm, 'SDBM_File', $db_file, O_RDONLY, 0;
```

Emptying the Contents of a DBM Database

Because the DBM database is represented as a tied hash, we can empty the entire database using a single `undef` on the hash itself:

```
undef %dbm;
```

This wipes out every key in the hash and, along with it, every entry in the underlying DBM. It's a good demonstration of just how important it is to take care with DBM files – one false move and you've wiped out all your data. (You do make backups though, yes? Good. I thought so.)

Closing a DBM Database

When we've finished with a database, it's good practice to disconnect from it – break the link between the hash and the file on disk. Just as file handles are automatically closed when a script ends, tied variables are automatically untied. However, it's bad programming practice to rely on this, since we never know how our script might be modified in the future.

It's simple enough to untie a DBM database – just use the `untie` operator:

```
untie %dbm;
```

Note that, as with any tied variable, `untie` will produce warnings if we untie the DBM hash when there are references to it still in existence. See the `perltie` documentation page for more details.

Adding and Modifying DBM Entries

Once a DBM database is tied to our hash variable, we can add and modify data in it by simply accessing the hash variable. To create a new entry in an open database that's tied to `$dbm`, we simply add a new key-value pair to the hash:

```
$dbm{'newkey'}="New Value";
```

The value must be a scalar. We cannot supply a reference to a hash or list and expect the database to store it. Although the database *will* store the reference, it will store it as a string (in the same way that `print` translates a reference if we try to print it). This string can't be converted back into a reference, and the data that it points to is not stored in the DBM database.

Reading DBM Entries

Similarly, we read data from a DBM database by accessing the tied hash variable in the normal ways. So to read a particular key value we might put:

```
my $value=$dbm{'keyname'};
```

To check if a given key exists in the database:

```
if (exists $dbm{'keyname'}) {...}
```

To get a list of all keys in the database:

```
my @keys=keys %dbm;
```

To dump a sorted table of all the keys and values in the database:

```
foreach (sort keys(%dbm)) {
   print "$_ => $dbm{$_}\n";
}
```

As the above examples show, we can treat our database almost exactly as if it was an ordinary hash variable – that's the beauty of tie.

Deleting from a DBM Database

If we want to remove the key and its associated data entirely, we can use Perl's delete function, just as with an ordinary hash:

```
delete $dbm{'key'};
```

Normally, `delete` just removes a key-value pair from a hash. Remember though, if the hash is tied to a DBM database, then the database record will be removed as well.

Try It Out – A Simple DBM Database

Let's have a quick look at how we can bring together what we've seen so far. The following program is a simple DBM database manipulator, which we can use to store on disk whatever information we like, in the form of key-value pairs:

```
#!/usr/bin/perl
#simpledb.plx
use warnings;
use strict;
use POSIX;
use SDBM_File;      # or GDBM_File / NDBM_File / AnyDBM_File...
```

```
my %dbm;
my $db_file = "simpledb.dbm";

tie %dbm, 'SDBM_File', $db_file, O_CREAT|O_RDWR, 0644;

if (tied %dbm) {
   print "File $db_file now open.\n";
} else {
   die "Sorry - unable to open $db_file\n";
}

$_ = "";         # make sure that $_ is defined

until (/^q/i) {

print "What would you like to do? ('o' for options): ";
chomp($_ = <STDIN>);

if ($_ eq "o") { dboptions() }
elsif ($_ eq "r") { readdb() }
elsif ($_ eq "l") { listdb() }
elsif ($_ eq "w") { writedb() }
elsif ($_ eq "d") { deletedb() }
elsif ($_ eq "x") { cleardb() }
else { print "Sorry, not a recognized option.\n"; }
}

untie %dbm;

#*** Option Subs ***#

sub dboptions {
   print<<EOF;
      Options available:
      o - view options
      r - read entry
      l - list all entries
      w - write entry
      d - delete entry
      x - delete all entries
EOF
}

sub readdb {
   my $keyname = getkey();
   if (exists $dbm{"$keyname"}) {
      print "Element '$keyname' has value $dbm{$keyname}";
   } else {
      print "Sorry, this element does not exist.\n"
   }
}

sub listdb {
   foreach (sort keys(%dbm)) {
      print "$_ => $dbm{$_}\n";
   }
}
```

```
sub writedb {
   my $keyname = getkey();
   my $keyval = getval();

   if (exists $dbm{$keyname}) {
      print "Sorry, this element already exists.\n"
   } else {
      $dbm{$keyname}=$keyval;
   }
}

sub deletedb {
   my $keyname = getkey();
   if (exists $dbm{$keyname}) {
      print "This will delete the entry $keyname.\n";
      delete $dbm{$keyname} if besure();
   }
}

sub cleardb {
   print "This will delete the entire contents of the current database.\n";
   undef %dbm if besure();
}

#*** Input Subs ***#

sub getkey {
   print "Enter key name of element: ";
   chomp($_ = <STDIN>);
   $_;
}

sub getval {
   print "Enter value of element: ";
   chomp($_ = <STDIN>);
   $_;
}

sub besure {
   print "Are you sure you want to do this?";
   $_ = <STDIN>;
   /^y/i;
}
```

How It Works

Once we've done our usual preliminaries, specifying `use POSIX` and `use SDBM_File`, we declare our hash and specify the filename to use:

```
my %dbm;
my $db_file = "simpledb.dbm";
```

Next, we use these values to `tie` together the hash and the file (creating the file if necessary), confirming success if it works, and telling the program to `die` otherwise:

```
tie %dbm, 'SDBM_File', $db_file, O_CREAT|O_RDWR, 0644;

if (tied %dbm) {
   print "File $db_file now open.\n";
} else {
   die "Sorry - unable to open $db_file\n";
}
```

Now, we set up an `until` loop. This prompts the user for a standard input and, for specific responses, calls appropriate subroutines. The loop continues until `$_` can be matched to the regular expression `/^q/i` – in other words, the user enters **q** or **Quit** (or, for that matter, **qwertyuiop**):

```
until (/^q/i) {

print "What would you like to do? ('o' for options): ";
chomp($_ = <STDIN>);

if ($_ eq "o") { dboptions() }
elsif ($_ eq "r") { readdb() }
elsif ($_ eq "l") { listdb() }
elsif ($_ eq "w") { writedb() }
elsif ($_ eq "d") { deletedb() }
elsif ($_ eq "x") { cleardb() }
else { print "Sorry, not a recognized option.\n"; }
}
```

and once we're done with the `until` loop, we're done with the database – so we `untie` from the hash:

```
untie %dbm;
```

Now we move on to the subroutines. The first six of these correspond to our six options above. The first displays a list of those options, using a here-document:

```
sub dboptions {
   print<<EOF;
      Options available:
      o - view options
      r - read entry
      l - list all entries
      w - write entry
      d - delete entry
      x - delete all entries
EOF
}
```

The second lets the user specify the name of a hash key and displays the corresponding value. That is, unless the key doesn't exist, in which case we offer an explanation:

```
sub readdb {
   my $keyname = getkey();
   if (exists $dbm{"$keyname"}) {
      print "Element '$keyname' has value $dbm{$keyname}";
   } else {
      print "Sorry, this element does not exist.\n"
   }
}
```

Next, a variation on the above. This simply lists all the key-value pairs in the database:

```
sub listdb {
   foreach (sort keys(%dbm)) {
      print "$_ => $dbm{$_}\n";
   }
}
```

The fourth subroutine lets the user specify both a key and a value, and as long as the key hasn't already been used, it uses this pair to define a new entry in the database:

```
sub writedb {
   my $keyname = getkey();
   my $keyval = getval();

   if (exists $dbm{$keyname}) {
      print "Sorry, this element already exists.\n"
   } else {
      $dbm{$keyname}=$keyval;
   }
}
```

Next, the user can specify a key, and (following a warning) the corresponding entry in the database is deleted:

```
sub deletedb {
   my $keyname = getkey();
   if (exists $dbm{$keyname}) {
      print "This will delete the entry $keyname.\n";
      delete $dbm{$keyname} if besure();
   }
}
```

Finally, the `cleardb` subroutine lets the user wipe the whole database clean:

```
sub cleardb {
   print "This will delete the entire contents of the current database.\n";
   undef %dbm if besure();
}
```

In several of the subroutines above, we had cause to perform certain checks several times over. Rather than spelling them out for each subroutine, we put them into subroutines of their own, and these are what we now come to.

The first two of these are essentially the same – both prompt the user for an input, which is chomped and then returned to the calling code:

```
sub getkey {
   print "Enter key name of element: ";
   chomp($_ = <STDIN>);
   $_;
}
```

Only the text of the prompt differs between the two: one requesting a key, the other a value:

```
sub getval {
   print "Enter value of element: ";
   chomp($_ = <STDIN>);
   $_;
}
```

The very last subroutine lets us add warnings to potentially dangerous operations – once again, this will prompt for a user input, but then return `TRUE` if (and only if) that input matches `/^y/i`, that is **y**, **Yes** (or even **yeah!**):

```
sub besure {
   print "Are you sure you want to do this?";
   $_ = <STDIN>;
   /^y/i;
}
```

As we saw above, this can be added to operations *very* simply, by saying:

```
<do_something_risky> if besure();
```

with the result that nothing happens *unless* the user specifically responds 'y'.

Writing Portable DBM Programs with the AnyDBM Module

Sometimes we won't care which DBM format we use, just so long as it works. This is particularly true if we want to write a portable script that will work on any system, regardless of which DBM implementations it supports. If we want our program to run on someone else's computer, there's no way we can tell in advance what DBM library they have.

Fortunately, there's a way around this problem. The `AnyDBM` module is a convenient wrapper around all the DBM modules, which can be substituted wherever we'd normally use a specific DBM module. It searches the system for different DBM implementations and uses the first one it finds. By using this, we can avoid having to choose a DBM format and leave it to the script. Here is an example of how we can use `AnyDBM` to `tie` to an arbitrary DBM database format:

```
#!/usr/bin/perl
#anydbm.plx
use strict;
use warnings;
use AnyDBM_File;
use POSIX;

my %dbm;
my $db_file="anydbmdemo.dbm";

tie (%dbm, 'AnyDBM_File', $db_file, O_CREAT|O_RDWR, 0644);
```

```
unless (tied %dbm) {
    print "Error opening $db_file $!\n";
} else {
    $dbm{'Created'}=localtime;
    foreach (sort keys %dbm) {
        print "$_ => $dbm{$_}\n";
    }
untie %dbm;
}
```

AnyDBM searches for DBM database implementations in a predefined order, defined by the contents of its @ISA array. As we saw in the previous chapter, this is a special array, used to define what parents a child object inherits its methods from. The search will therefore look for modules in the order specified by the elements in that array – this is the default order:

- NDBM_File
- DB_File
- GDBM_File
- SDBM_File
- ODBM_File

AnyDBM will therefore create an NDBM database in preference to any other kind; failing that, a Berkeley (BSD) DB database; then a GDBM database; an SDBM database; and finally, an ODBM database. Since SDBM is guaranteed to exist, ODBM will typically never be reached.

By predefining AnyDBM's @ISA array we can change the order in which it searches the various DBM modules. If we want to tell AnyDBM that we prefer GDBM (which we probably do), with NBDM second and SDBM third, but that we do not want to use ODBM or BSD DB, even if they are installed, we'd write:

```
BEGIN {
    @AnyDBM_File::ISA = qw(GDBM_File NDBM_File SDBM_File);
}
use AnyDBM_File;
```

Note that this works because AnyDBM specifically checks to see if its @ISA array has already been defined before setting it up with the default order. This won't necessarily be the case for other Perl modules.

Copying from One DBM Format to Another

Because DBM databases are represented through tie as hashes, converting one database format to another is almost disturbingly easy. Say we wanted to convert an NDBM database to the newer GDBM format. Here's how we do it:

```
#!/usr/bin/perl
#copydbm.plx
use warnings;
use strict;
use POSIX;
use NDBM_File;
use GDBM_File;

my (%ndbm_db,%gdbm_db);
my $ndbm_file='/tmp/my_old_ndbm_database';
my $gdbm_file='/tmp/my_new_gdbm_database';

tie %ndbm_db, 'NDBM_File',$ndbm_file, O_RDONLY, 0;
tie %gdbm_db, 'GDBM_File',$gdbm_file, O_CREAT|O_WRONLY, 0644;

%gdbm_db=%ndbm_db;

untie %ndbm_db;
untie %gdbm_db;
```

As the above example shows, the hard part of the conversion is handled for us in a simple hash copy.

Complex Data Storage

Now, as we've seen, DBM databases get on just fine with scalar variables, but it seems that's about *all* they can handle. So what if we want to store complex data like lists and hashes? The rough-and-ready answer is we need to convert them into a form that is, scalar string values that DBM *can* store. If we're mainly storing strings of varying sizes, the easiest option is join them with a separator that's guaranteed never to occur in the data. For example, to store a list we might use:

```
$dbm{'key'}=join ("_XYZ_",@list);
```

We can subsequently retrieve the packed values with the `split` function:

```
my @list=split "_XYZ_",$dbm{'key'};
```

However, it turns out we don't actually have to labor over interminable `join`s and `split`s, because (surprise, surprise!) we can use one of Perl's **serializing** modules. These do exactly the same job, but rather more efficiently and flexibly. The three main choices are `Data::Dumper`, `Storable`, and `FreezeThaw` (all of which are available from your nearest CPAN mirror).

Of the three, `Storable` is the most flexible, and `FreezeThaw` the most lightweight. `Data::Dumper` is the oldest, but also the least efficient. Here's an example using `Storable`'s `freeze` and `thaw` to store hashes in a DBM file:

```
#!/usr/bin/perl
#hashdbm.plx
use warnings;
use strict;
use POSIX;
```

```
use SDBM_File;
use Storable;

my %dbm;
my $db_file="demo.dbm";

tie %dbm, 'SDBM_File', $db_file, O_CREAT|O_RDWR, 0644;

# store a hash in DBM (note that we must supply a reference):
$dbm{'key'}=Storable::freeze({Name=>"John", Value=>"Smith", Age=>"42"});

# retrieve a hash from DBM (as a reference or as a hash):
my $href=Storable::thaw($dbm{'key'});
my %hash=%{ Storable::thaw($dbm{'key'}) };
```

Multi-Level DBM (MLDBM)

We know that DBM databases only store scalar values – they won't store lists or hashes unless we take steps to convert them into strings, a process known as serializing. Fortunately, we don't have to do the work of serializing ourselves, since there are several Perl modules that will do it for us – we just saw how the `Storable` module can do this.

However, even *this* is more work than we need to do. There's a module available on CPAN called `MLDBM`, which bundles a DBM module together with a serializing module transparently. This allows us to create complex data structures in a DBM file without having to worry about how they're stored. With `MLDBM` we can store hashes of hashes, lists of lists, hashes of list, and even hashes of lists of hashes. *Any* type of data structure that can be created in Perl can be stored in an `MLDBM` database.

Opening an `MLDBM` database is similar to opening a regular DBM database:

```
#!/usr/bin/perl
#mldbm1.plx
use warnings;
use strict;
use MLDBM;
use POSIX; #for O_CREAT and O_RDWR symbols
use strict;

my %mldbm;
my $mldb_file="mlanydbmdemo.dbm";

tie %mldbm, 'MLDBM', $mldb_file, O_CREAT|O_RDWR, 0644;
```

This creates an `SDBM` database to store the actual data, and uses the `Data::Dumper` module to do the serializing. Neither of these choices is a particularly good one: `SDBM` is not great for anything but small databases, and `Data::Dumper` serializes data as actual Perl code – great if we want to `eval` it, but not very efficient in terms of storage.

`MLDBM` is agnostic about which actual DBM package and serializer we use, just so long as the functions it requires are supported. Here's an example of using `MLDBM` to manage a `GDBM` database with data serialized with the `Storable` module – a much more efficient solution:

```
#!/usr/bin/perl
#mldbm2.plx
use warnings;
use strict;
use GDBM_File;
use Storable;
use MLDBM qw(GDBM_File Storable);
use POSIX;                    #for O_CREAT and O_RDWR symbols
use strict;

my %mldbm;
my $mldb_file="mlanydbmdemo.dbm";

tie %mldbm, 'MLDBM', $mldb_file, O_CREAT|O_RDWR, 0644;
```

We can use `MLDBM` with AnyDBM, too, removing the need to choose the underlying database. Because we've decided to have a preference for `GDBM`, we'll also alter AnyDBM's search order:

```
#!/usr/bin/perl
#mldbm3.plx
use warnings;
use strict;
BEGIN {
    @AnyDBM_File::ISA = qw(GDBM_File DB_File NDBM_File SDBM_File);
}

use AnyDBM_File;
use Storable;
use MLDBM qw(AnyDBM_File Storable);
use POSIX;                    #for O_CREAT and O_RDWR symbols
use strict;

my %mldbm;
my $mldb_file="mlanydbmdemo.dbm";

tie (%mldbm, 'MLDBM', $mldb_file, O_CREAT|O_RDWR, 0644);

unless (tied %mldbm) {
    print "Error opening $mldb_file: $!\n";
} else {
    if (exists $mldbm{'Created'}) {
        $mldbm{'Created'}=localtime;
    } else {
        $mldbm{'Updated'}=localtime;
    }
    foreach (sort keys %mldbm) {
        print "$_ => $mldbm{$_}\n";
    }
    untie %mldbm;
}
```

Once a DBM database has been opened or created via `MLDBM`, we can modify its contents as before, but we're no longer limited to storing scalar values.

To finish off our discussion of DBM databases, we'll take a look at program that creates an `MLDBM` database and writes various kinds of values into it. All the assignments below are valid, but note the comments:

```
#!/usr/bin/perl
#mldbm4.plx
use MLDBM qw(SDBM_File Storable);
use POSIX;
use warnings;
use strict;

my %mldbm;
my $mldb_file="mldbmdemo.dbm";

tie (%mldbm, 'MLDBM', $mldb_file, O_CREAT|O_RDWR, 0644);

unless (tied %mldbm) {
    print "Error opening $mldb_file: $!\n";
} else {
    # wipe out the old contents, if any
    undef %mldbm;

    $mldbm{'Created'}=localtime;

    # assign a list anonymously, directly and as a copy
    $mldbm{'AnonymousList'}=[1,2,3,4,"Five",6,7.8];
    my @list=(9,"Ten",11,12.13,14);
    $mldbm{'OriginalList'}=\@list;
    $mldbm{'CopyOfList'}=[ @list ];
    $mldbm{'NumberOfListElems'}=@list;
    $list[0]="Nine"; #does NOT modify 'OriginalList'

    # assign a hash anonymously, directly and as a copy
    $mldbm{'AnonymousHash'}={One=>'1',Two=>'2',Three=>'3'};
    my %hash=(Four=>'4',Five=>'5',Six=>'6');
    $mldbm{'OriginalHash'}=\%hash;
    $mldbm{'CopyOfHash'}={ %hash };
    $mldbm{'NumberOfHashKeys'}=keys %hash;
    $hash{Four}="IV"; #does NOT modify 'OriginalHash'

    # assign a random key and value
    $mldbm{rand()}=rand;

    # a more complex assignment
    $mldbm{'HashOfMixedValues'}={
        List1=>[1,2,3],
        List2=>[4,5,6],
        String=>"A String",
        Hash1=>{
                A=>"a",
                B=>"b",
                Hash2=>{
                    C=>"c",
                },
        },
        Number=>14.767,
        List3=>[7,8,9],
    };
```

```
    # now dump out the contents again
    foreach (sort keys %mldbm) {
        print "$_ => $mldbm{$_}\n";
        if (my $ref=ref $mldbm{$_}) {
            if ($ref eq 'HASH') {
                foreach my $key (sort keys %{ $mldbm{$_} }) {
                    print "\t$key => $mldbm{$_}{$key}\n";
                }
            } else {
                print "\t",(join ",",@{ $mldbm{$_} }),"\n";
            }
        }
    }

    untie %mldbm;
}
```

There are three main points to note about this example:

- We can assign an existing hash or list either:
  - with a backslash reference to the original,
  - or with a reference constructor (using curly or square brackets).

  In both cases, `MLDBM` makes a **copy** of the variable. If we try using a backslash reference to point to the original variable, and then change it, the change isn't reflected in the database.
- Similarly, if we try taking a reference to anything in a `MLDBM` database and use it later, it won't work. The reference isn't tied, so it won't be handled by the `MLDBM` module. We can *only* access values through the **top** of the data structure, where the tie is.
- Finally, just as with a normal list, if we don't supply a reference for a value, then we get the number of elements instead – probably not what we intended.

Beyond Flat Files and DBM

There's real power to be had when we're dealing with huge quantities of data in all shapes and sizes. It's enough to take your breath away. The trouble is that so far, we've only really looked at kindergarten-level data stores – while working with DBM is great for speedy solutions, a real-world application of any great size needs to work with a good, powerful, reliable database server.

Flat files are very simple to work with: They're in an easy-to-read format, even in a simple text editor, and (as long as they're small enough) you can pass on your files around on a floppy disk, should you need to use your data on another machine. Unfortunately, it's what they are *not* that's the problem. Our `.dbm` files are essentially text files, so:

- Text files aren't scalable. When you search them, each key-value pair in the file is searched sequentially. Consequently, the bigger the file, the more time it's going to take to find what you're looking for.
- Cross-referencing data between files is tricky and gets more and more perplexing the greater the number of tables you add into the mix.
- It's unwise to give multiple users simultaneous access – if you let them work on the same data at the same time, your files could end up containing inconsistencies.

There's no easy solution to these problems, at least no set-up that will make everything as easy as working with flat files. However, we do at least have the technology to address these problems and make for workable solutions – we can store our information in **relational databases**.

Introducing Relational Databases

The relational database model was first devised in 1970 and has since developed to a point where practically every major database server – SQL Server, Oracle, Sybase, DB2, Informix, uses it to store data. In this model, items of data are stored in **tables**, which group together **records** containing the same type of information. So, for example, there might be a record for each patient in a doctor's waiting rooms. The database would hold details such as name, address, previous ailments, date of previous visit, and prescriptions stored in separate **fields**.

You'd want to hold the same sort of details for every patient, but each set would be specific to a certain one. Each record would therefore require a unique identifier, known as a **key**, corresponding to one particular patient. This key could then be used to cross-reference information stored in other tables in the same database.

Nowadays, most database vendors following this model also use a similarly generic way to **query** the database so as to retrieve information. This method takes the form of a language for creating queries to the database – asking questions if you like. This is called **Structured Query Language**, or by its more familiar name **SQL** (pronounced 'sequel'). We'll come back to this in a bit, but now, suffice it to say that the way in which we write SQL queries remains the same, no matter what database you're querying.

RDBMS (or Relational DataBase Management Servers, to use their full title) work in quite a different fashion from flat files. Perhaps most notable is the actual lack of a file corresponding to your data. Instead, the database server (from now on, just called the 'server') holds all the info within itself, and as a rule, viewing the data externally isn't possible, save through querying the server first.

You may also find it strange to realize that in order to query a database, the server needn't be located on your machine, although in the wonderful world of the Internet, that might not be such a shock. Indeed in general, the larger the database and the more data it contains, the more likely it is that the server will be accessed remotely.

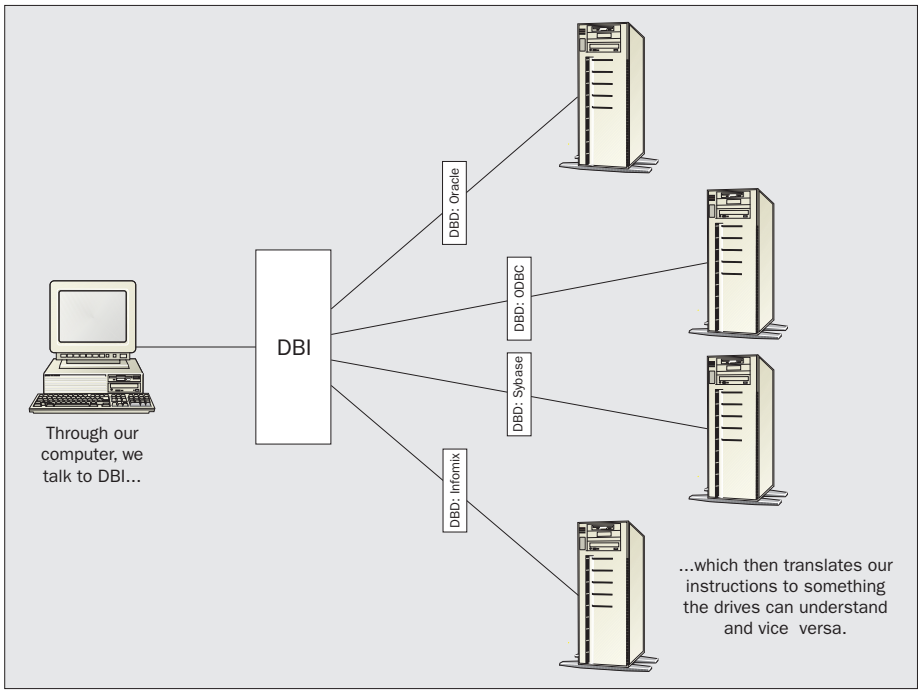
Introducing DBI

So, relational databases are pretty great – they do things that flat files and DBM can't, we can ask them questions using the same SQL queries no matter which actual database we're using. So what's the catch? Well, it's a matter of delivering that query to the database and getting back the data we want.

Technically speaking, it's because each server has a different API (Application Programming Interface) and therefore a different set of commands for doing the same things. Somewhat less technically, it's because behind the scenes, each database uses a different language to talks about the same things.

When we looked at DBM files, there were five different modules for the five types of file we could use. In the same way, there's a **database driver** or **DBD** module for each type of database server. So if you want to work with a database on the Sybase platform, you'd install the `DBD::Sybase` module and use it to query your databases in Sybase language only. You can see how this can quickly become rather a pain if you're working with more than one brand of database and want to port your code between them.

Enter DBI and your solution. **DBI** (the standard **DataBase Interface**) is a database-independent interface that sits on top of all these database drivers and acts as a translator for you. All we need do is tell DBI which driver we're using, and it will translate your instructions into those that the driver can understand:



The real beauty of DBI is that, if for some reason you come to the conclusion that, say, MySQL isn't offering you the facilities of one of the more high-end databases (like DB2 or Sybase), all you have to do is transfer your data to the new DB and redirect your code to look at it – you won't have to rewrite your code at all.

So What Do We Need?

Before going any further, we should find out what we already have installed. We've already established a rough shopping list. We'll need a database, a driver for that database and DBI too – let's start off with DBI.

If you're coming to this subject for the first time, the chances are you've not got DBI installed yet, but you can do a simple check by typing `perldoc DBI` at the command prompt. If it has been installed, you'll see:

```
>perldoc DBI
NAME
      DBI - Database independent interface for Perl

SYNOPSIS
...
```

and so on. On the other hand, if it hasn't been installed yet, you'll get

```
>perldoc DBI
No documentation found for "DBI".
>
```

If you get this, you'll need to do the obvious and get it up and running. Here's how.

Installing DBI

DBI is a module just like any other, and can be installed in the same ways we saw in Chapter 10.

Installing with PPM

Once again, you probably have it easiest if you've installed ActiveState Perl and now have PPM at your disposal. If you're a PPM user, installing DBI is a matter of activating PPM on the command prompt and issuing the command:

```
>install DBI
```

The rest of the installation is automatic.

Installing from the Source

The latest version of the DBI module source code is always available at http://www.symbolstone.org/technology/perl/DBI/. At time of writing, this was at version 1.13. Download the zipped source code and decompress it with Winzip (on Windows) or the command:

```
>gzip -dc DBI-1.13.tar.gz | tar -xvf
```

We now need to issue the following four commands to compile the source and install our module:

```
>perl makefile.pl
>make
>make test
>make install
```

Installing from CPAN

The last option here is to use the CPAN exporter module and install DBI directly from CPAN. From the command prompt then, there are two simple steps:

```
>perl -MCPAN -e shell
cpan> install DBI
```

and don't forget to **quit** CPAN when it's done.

Try It Out - Quizzing the Drivers

Now that we're all on a level playing field with DBI installed, let's have a look and see what we get in the base installation that we can use. The following program will do just that for us.

```
#!/usr/bin/perl
#available.plx
use warnings;
use strict;
use DBI;
```

```
print "Available DBI Drivers and Data Sources:\n\n";
my @drivers=DBI->available_drivers('quiet');
my @sources;

foreach my $driver (@drivers) {
    print "$driver\n";
    @sources=eval { DBI->data_sources($driver) };
    if ($@) {
        print "\tError: ",substr($@,0,60),"\n";
    } elsif (@sources) {
        foreach (@sources) {
            print "\t$_\n";
        }
    } else {
        print "\tNo known data sources\n";
    }
}
```

With any luck, you'll see the following after a new installation of DBI:

```
>perl available.plx
Available DBI Drivers and Data Sources:

ADO
      No known data sources
ExampleP
      dbi:ExampleP:dir=.
Proxy
      Error: install_driver(Proxy) failed: Can't locate RPC/PlClient.pm ...
>
```

We can see then that DBI comes ready with three supplied drivers – `ADO`, `Proxy`, and `ExampleP`. We'll return in just a short while to see what ADO and Proxy are exactly, when we look at all the possible DBDs you can download. However, it's worth noting now that `ExampleP` is an example DBD 'stub' for developers of DBI drivers to work from and provides no useful functionality: that's why it won't be mentioned later.

How It Works

After the usual headers, the first thing we do is to import the methods that DBI has to offer and then print out a header for our results:

```
use DBI;

print "Available DBI Drivers and Data Sources:\n\n";
```

Now we get to grips with our first DBI method: `available_drivers()` simply searches through the directories listed in your `@INC` array, and if it finds any `DBD::*` modules, stores them away in `@drivers`:

```
my @drivers=DBI->available_drivers('quiet');
my @sources;
```

Now we simply loop through the drivers we've found and see which databases we can talk to with them:

```
foreach my $driver (@drivers) {
    print "$driver\n";
```

Another DBI method, `data_sources()` returns a list of data stores we can talk to through the driver. Note that while it should work fine by itself, we've wrapped our call in an `eval` clause in case a DBD fails to load. If you remember, `eval` runs the code under its auspices, but ignores any warnings or calls to `die` from within, which might occur here if the driver fails to install:

```
    @sources=eval { DBI->data_sources($driver) };
```

If an error does occur within `eval()`, it will get stored in `$@`, so we'll print that first. If there isn't one, we either print the data stores we've found that correspond to the driver, or a nice message saying we couldn't find any:

```
    if ($@) {
        print "\tError: ",substr($@,0,60),"\n";
    } elsif (@sources) {
        foreach (@sources) {
            print "\t$_\n";
        }
    } else {
        print "\tNo known data sources\n";
    }
}
```

What's Available

So DBI installs two drivers for future use plus an example one – there are plenty more out there though, enough to let us work with pretty much any database we choose. Most DBD modules are simply drivers for specific third-party database servers (such as Oracle, MySQL, or Informix), but some are in fact interfaces to other database connectivity protocols (such as `DBD::ADO` and `DBD::ODBC`), allowing Perl to communicate with servers that support these protocols. Programmers wanting to access Microsoft SQL servers therefore have both ADO and ODBC (as well as the `DBD::Sybase` module) as options.

> *A few DBD modules do* not *require a database server. Notable amongst these is the `DBD::CSV` driver, which makes use of several other Perl modules to provide a SQL interface to database files in comma-separated values (CSV) format. It's a very convenient way to implement a SQL database with no additional software – it's also a good way to build prototype database applications before migrating them to a real database server. DBI allows us to write generic code without worrying about the underlying server, so migrating from one database to another shouldn't be a problem.*

Here's a list of currently supported databases and their DBD modules, all of which are accessible from the Perl DBI homepages at http://www.symbolstone.org/technology/perl/DBI/:

- `DBD::ADO`
  The interface to Microsoft's **Active Data Objects** data access technology. The driver itself is installed with DBI, but in order to pass requests, you'll also need ADO (version 2.1 or later) and the Win32::OLE module installed as well. You can find more about ADO at http://www.microsoft.com/data/ado/.

- `DBD::Adabas`
  The driver for **Adabase** database servers.

- `DBD::Altera`
  The driver for **Altera** database servers.

- `DBD::CSV`
  The driver to access Comma-Separated Value files. These files can survive outside and so don't require a running database server. This makes them a good choice for creating simple SQL-driven databases with a view to migrating to a proper server later on. In addition to `DBD::CSV` however, you'll also need to install the `Text::CSV_XS` modules to read and write to CSV files and also the `SQL::Statement` module to parse SQL statements and emulate a real SQL server.

- `DBD::DB2`
  The driver for **DB2** database servers, as built by IBM. See http://www.softwate.ibm.com/data/db2 for more information.

- `DBD::Empress`
  The driver for **Empress** database servers and EmpressNet distributed databases. See http://www.empress.com for more information.

- `DBD::Illustra`
  The driver for **Illustra** database servers.

- `DBD::Informix`
  The driver for **Informix Online** and **Informix SE** database servers from version 5 onwards. Note that in order to work, this driver requires the presence of a licensed copy of the Informix Client SDK prior to installation. See http://www.informix.com for more information.

- `DBD::Ingres`
  The driver for Computer Associates' **Ingres 6.4** and **OpenIngres** (all versions) database servers. See http://www.cai.com/products/ingres.htm for more information.

- `DBD::Interbase`
  The driver for **Interbase** database servers. See http://www.interbase.com for more information.

- `DBD::ODBC`
  The driver for Microsoft's **ODBC** database connectivity protocol, versions 2 and 3 on Win32 and Unix systems. Note that in order for this driver to access a database through ODBC, an underlying ODBC driver for the chosen platform and database is also required. See http://www.microsoft.com/data/odbc for more information.

- `DBD::Oracle`
  The driver for **Oracle 7** and **Oracle 8/8i** database servers. It also includes an emulation mode for older 'legacy' Perl scripts written to use the Perl 4 `oraperl` library. See http://www.oracle.com for more information.

- `DBD::Pg`
  The driver for **PostgreSQL 6.4** and **6.5** databases. This is a freely available open source database, frequently bundled with open source operating systems like Linux. See http://www.postgresql.org for more information.

- `DBD::Proxy`
  The driver for communicating with **remote** DBI applications. However, it is not needed to access remote databases whose drivers already support remote access. It is useful though for propagating DBI requests through firewalls and can optionally cache networked DBI connections for CGI scripts. `DBD::Proxy` is bundled with the DBI package itself.

- `DBD::SearchServer`
  The driver for Fulcrum **SearchServer**/**PCDOCS**. See http://www.pcdocs.com for more information.
- `DBD::Solid`
  The driver for **Solid** database servers.
- `DBD::Sybase`
  The driver for **Sybase 10** and **Sybase 11** database servers. It also has a limited interaction with Sybase 4. Interstingly, with the addition of Sybase Open Client or the FreeTDS libraries, this driver can also support Microsoft MS-SQL servers. See http://www.sybase.com, http://www.freetds.org for more information
- `DBD::Unify`
  The driver for **Unify** database servers.
- `DBD::XBase`
  Contains drivers for **dBaseIII**, **dBaseIV**, and **Fox** databases.
- `Msql-MySQL-modules`
  A bundle of modules for **Msql** and **MySQL** databases, both popular and freely available, and very similar in ability. Includes the `DBD::mSQL` and `DBD::mysql` modules. For more information, see http://www.Hughes.com.au/products/msql/ and http://www.mysql.com/ respectively.

While all these modules work similarly and present the same basic interface to DBI, there are many subtle variations in the way that they work. It pays to read the included documentation for a given driver before using it – `perldoc DBD::<DriverName>` should produce some useful information.

Our DB of Choice – MySQL

For the rest of this chapter, we're going to be working with one specific database and its driver – MySQL. Why this one in particular? A number of reasons actually:

- It's available on the same platforms that DBI is - Solaris, Linux and Windows.
- It's fast enough to be run on almost any machine available at the moment.
- It's free!

You can of course choose to follow the rest of this chapter using another database driver. It would be quite understandable, for instance, if Windows users decided it best to use `DBD::ADO` so that they could work with already installed Access or SQL Server databases. The rest of this chapter won't even *try* to teach you everything – it will however teach you the basics of working with database servers, and will apply to any database you may happen to use. That said, let's get on and get MySQL up and running.

> *Note that if you do decide to use a different database than MySQL, each driver comes with its own set of methods on top of those in DBI. For example, in this chapter, we briefly use `NAME` and `NUM_OF_FIELDS` at the end of the chapter that are MySQL specific. Always check the drivers documentation for which methods they do and do not support beside those in DBI*

Installing on Windows

As usual, installing MySQL on Windows will be a lot simpler than installing it on Linux. You can download the shareware version of MySQL 3.22.34 from the MySQL homepage at http://www.mysql.com. It should come to you as a zipped file – `mysql-shareware-3.22.34-win.zip`. If you unzip that, you'll find a file called `setup.exe` which you should run. The standard installation wizard will run you through a couple of questions. The defaults (a Typical install in `C:\MySQL`) will do fine.

Once the server and clients have installed, we'll need to get the server itself up and running. Windows 95 users should note that MySQL uses TCP/IP to talk to the client, so you'll need to install that from your Windows CD and to download Winsock 2 from the Microsoft website.

To start MySQL running, you'll need to open a command prompt window, navigate to `C:\MySQL\bin`, and issue the following command:

```
>mysqld-shareware
```

Likewise, use the following command to shut it down:

```
>mysqladmin -u root shutdown
```

Windows NT/2000 users also have the option of running MySQL as a service. First though, you'll need to copy and rename `my-example` from `C:\MySQL` to `C:\my.cnf`. This holds global values for MySQL, which the service reads on startup. After that it's simply a case of install MySQL as a service with:

```
>mysqld-shareware --install
```

and to start and stop the service, just use:

```
>net start mysql
>net stop mysql
```

Installing on Linux

Just like when we installed Perl, Linux users have the choice of installing MySQL from a package or using the source. In either case, you can obtain the files you need from http://www.mysql.com/download_3.22.html.

If you're after RPMs, then make sure you download the server, the client, the include files and libraries, and the client-shared libraries, for the correct platform. You should end up with the following four files (the exact version number, and the platform, may vary):

- `MySQL-3.22.32-1.i386.rpm`
- `MySQL-client-3.22.32-1.i386.rpm`
- `MySQL-devel-3.22.32-1.i386.rpm`
- `MySQL-shared-3.22.32-1.i386.rpm`

If you're after the source, then you'll just need the tarball, `mysql-3.22.32.tar.gz`.

Installing MySQL Using RPMs

We install RPMs using the command:

```
> rpm -Uvh filename.rpm
```

If you install them in the order listed on page 457, you should have no trouble.

When you install the server, you'll see the following documentation appear on the screen:

```
PLEASE REMEMBER TO SET A PASSWORD FOR THE MySQL root USER !
This is done with:
/usr/bin/mysqladmin -u root password 'new-password'
See the manual for more instructions.

Please report any problems with the /usr/bin/mysqlbug script!

The latest information about MySQL is available on the web at http://www.mysql.com
Support MySQL by buying support/licenses at http://www.tcx.se/license.htmy.

Starting mysqld daemon with databases from /var/lib/mysql
```

However, the `mysqladmin` program is one of the client tools, so we'll have to wait until after installing the client package first. Note, though, that the RPM immediately starts up the MySQL server program, `mysqld` (for MySQL daemon). It also creates the startup and shutdown script `/etc/rc.d/init.d/mysql`, which will ensure that MySQL starts whenever your computer is booted and shuts down conveniently whenever it is halted. You can use this script to start and stop `mysqld`, with the commands:

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

Now, install the `MySQL-client`, `MySQL-devel`, and `MySQL-shared` packages, and we're done.

Installing MySQL from Source

The installation procedure for the sourcecode should be fairly simple:

```
> tar -zxvf mysql-3.22.32.tar.gz
> cd mysql-3.22.32
> ./configure --prefix=/usr
> make
```

If make fails it is often because of a lack of memory, even on fairly high-spec machines. In this case, try:

```
> rm -f config.cache
> make clean
> ./configure --prefix=/usr --with-low-memory
> make
```

Now we simply run:

```
> make install
> mysql_install_db
```

Now, we'll need to set up some scripts to start and stop the MySQL server, `mysqld`. A typical startup script might read:

```
#!/bin/bash

/usr/bin/safe_mysqld &
```

And a script to shut the server down might be:

```
#!/bin/bash

kill `cat /usr/var/$HOSTNAME.pid`
```

Setting up the Root Account

Once the server and clients are installed, and the server's up and running, we can do the sensible thing and set the root password:

```
> mysqladmin -u root password elephant
```

choosing a much safer password than 'elephant', obviously.

Testing Our MySQL Server

With our setup complete, it just remains for us to test our MySQL installation with the following commands:

```
>mysqlshow
>mysqlshow -u root mysql
>mysqladmin version status proc
```

This should echo back to you the current MySQL configuration of both databases and TCP/IP connections.

Installing DBD::MySQL

Now that MySQL is installed, we just need the database driver for DBI to talk to. Again, the driver is available from http://www.symbolstone.org/technology/perl/DBI/.

Note that PPM's install command is a little different from usual, to cover both CPAN and the MySQL homespaces:

```
PPM> install "DBD::mysql" "http://www.mysql.com/Contrib/ppd/DBD-mysql.ppd"
```

Source-code downloaders and CPAN shell users, remember that `DBD::MySQL` comes in a bundle called `Msql-MySQL-modules` and not by itself. You need to make (or nmake) the whole package. When you first run perl makefile.pl, you'll get asked some questions about your MySQL configuration. If you left MySQL to the defaults when you installed it, you be able to leave the makefile to the defaults as well. If not, just answer the questions as appropriate.

What's Available Again?

Now we've got everything installed, we should be able to run `available.plx` again and see our MySQL driver and database appear on the list of the `@INC` directories. Sure enough, we get.

```
>perl available.plx
Available DBI Drivers and Data Sources:

ADO
        No known data sources
ExampleP
        dbi:ExampleP:dir=.
Proxy
        Error: install_driver(Proxy) failed: Can't locate RPC/PlClient.pm ...
mysql
        DBI:mysql:mysql
        DBI:mysql:test
>
```

All's well, so let's get down to work.

First Steps - The Database Cycle

Working with relational databases in DBI has three fundamental steps.

- Connecting to the database using `DBI->connect()`.
- Interacting with the database using SQL queries.
- Disconnecting from the database with `DBI-> disconnect()`.

The second step is a little more involved than the other two, and we must be wary of errors occurring throughout. From a high level though, this is what working with databases boils down to.

Connecting to a Database

Naturally enough, before we can start talking to a database server, we need to connect to it. Perl must know which driver to use to talk to which database, who wants to access it if asked and any operational nuances we're going to work under. We can do all of this using `DBI->connect()`.

One of DBI's class methods (so called as it affects all drivers), `DBI->connect()` has two forms:

```
$db_handle = DBI->connect (dbi:$data_source, $user, $password)
        || die $DBI::errstr;
$db_handle = DBI->connect (dbi:$data_source, $user, $password, \%attribs)
        || die $DBI::errstr;
```

Both versions return a handle to a database (more accurately, a DBI connection object) with which we can query our data in exchange for three basic pieces of information (and a fourth, optional one, which we'll return to later):

- The DBD name and that of the database (henceforth known as the **Data Source Name** or **DSN**) combined in the form `dbi:<driver>:<data source>` and supplied as a single parameter. For example, `dbi:mysql:test`.
- The user accessing the database server.
- The identifying password for the accessing user.

For example, to connect to the MySQL `test` database with user 'anonymous' and password 'guest', we would use:

```
my $dbh=DBI->connect('dbi:mysql:test','anonymous','guest') ||
   die "Error opening database: $DBI::errstr\n";
```

Once called, and providing all's well, we'll get a handle, `$dbh`, to read from and write to the database. We can either start working with `test` now or create more handles to different servers and databases. If the connection fails for whatever reason, `connect` returns the undefined value and tells us what went wrong in the variable `$DBI::errstr`, as shown above. However, it doesn't set `$!`. For example, if we'd misspelled `test` as `tess` and tried to connect, we'd get the message:

```
Error opening database: Unknown database 'tess'
```

It's possible that a given database server doesn't require (or support) a user and password, in which case these two parameters can be omitted. For most serious applications though, it's a good idea to have access restrictions in place, especially if we intend to make the database accessible from the web.

Connecting to a Remote Database

The actual process of connecting to remote (that is, based on your network or on the internet) databases is remarkably easy. Many (but not all) drivers allow a remote database to be specified in the Data Source Name. This takes the same form as the normal DSN but also includes a hostname and, optionally, a port number at which to reach the remote database. For example, here's how we might access our MySQL `test` database from a different host:

```
my $dbh=DBI->connect
             ('dbi:mysql:test:db.myserver.com:3077','anonymous','guest')
             || die "Error opening database: $DBI::errstr\n";
```

You'll have noticed earlier that we said some drivers allow `connect()` to specify a remote hostname. For those that don't, we can use the `DBD::Proxy` driver (which came with DBI), so long as there's a DBI driver running on the remote machine (the `DBI::Proxyserver` module is provided to allow such a server to be easily implemented). To use the proxy, we just modify the format of the DSN in the `connect` call:

```
my $host='db.myserver.com';
my $port='8888';
my $dsn='dbi:mysql:test';
my $user='anonymous';
my $pass='guest';

my $dbh=DBI->connect
             ("dbi:Proxy:hostname=$host;port=$port;dsn=$dsn",$user,$pass)
             || die "Error opening database: $DBI::errstr\n";
```

An alternative way to use the proxy is to define the environment variable `DBI_AUTOPROXY` to a suitable value in the form `hostname=<host>;port=<port>` If this variable is seen by DBI, it automatically modifies a normal `connect` call into a call that passes via the proxy. We can set the variable in our code, too:

```
$ENV{DBI_AUTOPROXY}="host=$host;port=$port";
my $dbh=DBI->connect($dsn,$user,$pass)
             || die "Error opening database: $DBI::errstr\n";
```

`DBD::Proxy` also supports a number of other features, such as the ability to cache and encrypt network connections, but that's a little bit beyond the scope of this introduction. If you're curious though, you can look in the perldocs for `DBI` and `DBD::Proxy`.

Connecting with the Environment

When you're working with lots of data, making mistakes on your own is bad enough, but having other people making them deliberately is even nastier. We've already helped secure our root user a little more (by setting a password for him while installing MySQL) and now we can take it a step further.

The trouble with the call to `connect()` is that all the information about your database and the account you're using to access it is there for prying eyes to see – and potentially abuse. However, we can hide that information within system environment variables, and this allows us to write database scripts without explicitly coding database details or user identities, as follows:

Variable Name	Replaces
`DBI_DRIVER`	The driver to use – for example, 'mysql'.
`DBI_DSN`	The data source name – for example, 'test'.
	The variable `DBNAME` is an alias for this value for older scripts that use it, but will likely be removed from DBI in the future.
`DBI_USER`	The user name – for example, 'anonymous'.
`DBI_PASS`	The user password – for example, 'guest'.
`DBI_AUTOPROXY`	If defined, the server and port number to access a remote DBI server.
	See 'Connecting to a Remote Database' above.

If, somewhere in our script then, we set the above variables, we could replace our original call to `connect()`, which looked like this:

```
my $dbh=DBI->connect('dbi:mysql:test','anonymous','guest') or
   die "Error opening database: $DBI::errstr\n";
```

to

```
my $dbh=DBI->connect('dbi::') or
   die "Error opening database: $DBI::errstr\n";
```

while the code (for example) that sets those variables:

```
...
local $ENV{"DBI_DRIVER"} = "mysql";
local $ENV{"DBI_DSN"} = "test";
local $ENV{"DBI_USER"} = "anonymous";
local $ENV{"DBI_PASS"} = "guest";
...
```

is located somewhere else. How many parameters you choose to leave out in favor of setting environment variables is up to you. When perl reads the call to `connect()`, and finds that the parameters it's looking for are missing, it will search for values in the environment variables above. If, finally, it still can't find them, it will return an error – something like:

```
Can't connect(dbi::), no database server specified and DBI_DSN env var not set at your_file.plx line xx.
```

The Fourth Parameter – Connection Flags

Bet you didn't think that saying "Hi" to a database would be so involved! Our mysterious optional fourth parameter is a hash reference holding a number of flags, which control the way our connection to a database operates. It's optional, which means that each of these flags has defaults already, but of those that you can set, there are three in particular worth being aware of:

AutoCommit - Transaction Control

The `AutoCommit` flag provides a good way to briefly look at transactions. Consider the classic situation where two banks are looking at the same person's account at the same time, both wanting to add money to his account. They both take the same original value and both update it with the new total according to their own calculations. If the fellow is unlucky, one of those deposits will have just got lost in the system. Unless, that is, the banking system was transaction enabled.

Not going into too much detail, 'enabling transactions' implies that whatever data is being accessed by a client is isolated from everyone else until they're done with it. Furthermore, that data is not altered unless *every* change that the client (in this case, one of the banks) wanted to make can be made and **committed** to the database. If one or more changes cannot be made, then those that have occurred so far (at least, since the last commit) are **rolled back** to the point where the data's in its original state.

`AutoCommit` affects the behavior of databases supporting transactions. If enabled, changes to the database have immediate effect. Otherwise, changes only have an effect on the *next* call to the DBI `commit` method and can be undone with the `rollback` method. Both `commit` and `rollback` will return an error if used with `AutoCommit`. Databases that don't support transactions won't allow you to disable `AutoCommit` (since `commit` and `rollback` don't do anything on them anyway).

Setting Error Urgency

We've already seen that when an error occurs in DBI, `$DBI::errstr` is populated with a description of the error and `$DBI::Err` the error's numeric value, but that the program itself continues. However, if we were to enable `RaiseError`, errors would cause DBI to call `die`.

By contrast, if we set `PrintError` (on by default), DBI will raise warnings (as generated by the `warn` operator) when errors occur, instead of `die`ing. Compare these situations with the norm, where `$DBI::errstr` is set if an error occurs, but `DBI` itself remains silent.

To set these flags at connect time, we just call `connect`, specifying the flags we want to set and the values we want them set to, like this:

```
my $dbh=DBI->connect('dbi:mysql:test','anonymous','guest',{
   PrintError=>0,
   RaiseError=>1
 }) || die "Error opening database: $DBI::errstr\n";
```

It's also possible to read and set these flags on a database handle once it has been created, but it requires a little cheat – accessing the flags directly instead of going through the object-oriented interface. For example,

```
my $auto=$dbh->{AutoCommit}; # are we auto-committing?
$dbh->{PrintError}=0; # disable PrintError
$dbh->{RaiseError}=1; # enable RaiseError
```

Actually, this is one of the shortcomings of DBI – there's no method call that will alter these flags individually.

Disconnecting from a Database

There are a lot of different things to consider when we're *making* a connection to a database, as we've seen, if only briefly. It's perhaps surprising therefore, that there's just one, simple way to break them all. We just need to use the `disconnect` method on the database handle we want to shut down, and down it goes:

```
$dbh->disconnect();
```

We can also tell DBI to disconnect *all* active database handles with the `disconnect_all` method,

```
DBI->disconnect_all();
```

However, if you're just using one database handle (which will usually be the case), then there's really nothing to be gained by using the class method.

Try It Out - Talking with the Database

It's only fitting that now we know how to connect and disconnect from a database, we at least try it out before going any further – so here we go:

```
#!\usr\bin\perl
#connect1.plx

use warnings;
use strict;
use DBI;

my $dbh=DBI->connect('dbi:mysql:test','root','elephant') ||
   die "Error opening database: $DBI::errstr\n";
print "Hello\n";
$dbh->disconnect || die "Failed to disconnect\n";
print "Goodbye\n";
```

The results are less than earth-shattering:

```
>perl connect1.plx
Hello
Goodbye
>
```

On the other hand, if we get this output, we know that we have successfully hooked up and then disconnected from the database. We can also try the different variations of `connect()` in this framework as well.

Interacting with the Database

So we've just one key thing to look at – how we interact with the database and what exactly we can do with it. Answer: everything we do with a database, we do by querying it in SQL.

Virtually all current database systems support database queries made in **Structured Query Language**. These **SQL queries**, also called **SQL statements**, fall into two distinct groups:

- Queries that do not return results, for example, creating a new table in an airport database for passengers who are checking in before a flight.
- Queries that do return results, for example, querying that check-in table for those passengers who have not yet checked in order to page them.

DBI offers us a four-step plan when putting forward SQL queries to our databases:

- we `prepare` a handle to a SQL query (**or statement handle**)
- we `execute` the query on the database
- assuming success, we `fetch` the results
- we `finish`, telling the database we're done

Try It Out - The First SQL Query

Right then, let's do a quick example on MySQL's `test` database and see what we're going to look through:

```
#!\usr\bin\perl
#querytest.plx

use warnings;
use strict;
use DBI;

my ($dbh, $sth, $name, $id);

$dbh=DBI->connect('dbi:mysql:test','root','elephant') ||
   die "Error opening database: $DBI::errstr\n";

$sth=$dbh->prepare("SELECT * from testac;") ||
   die "Prepare failed: $DBI::errstr\n";
```

```
$sth->execute() ||
   die "Couldn't execute query: $DBI::errstr\n";

while (( $id, $name) = $sth ->fetchrow_array) {
   print "$name has ID $id\n";
}

$sth->finish();

$dbh->disconnect || die "Failed to disconnect\n";
```

```
>perl querytest.plx
test has ID 162
>
```

How It Works

Once we've connected to the `test` database, our first step is to create a statement with `prepare`. We extract information from a SQL database with a `SELECT` statement, which selects and returns complete records or selected columns that match our criteria. In this case, the `*` is a wildcard character – so we're asking for *all* the information in the `testac` table in `test`, grouped by rows.

```
$sth=$dbh->prepare("SELECT * from testac;") ||
   die "Prepare failed: $DBI::errstr\n";
```

This process creates and returns a statement handle ready to be executed. A return value of `undef` indicates that there was an error, in which case we `die`.

Now that we have a statement handle, we can execute it. This sends the query to the database using the underlying database driver module. Again, this will return `undef` and (in this example) `die` if there were any kind of problem – for example, if the statement has already been executed, and `finish` has not been called. Otherwise, we can retrieve the results of the statement:

```
$sth->execute() ||
   die "Couldn't execute query: $DBI::errstr\n";
```

There are several ways to retrieve results, including the `fetch` family of methods and **bound columns** (both of which we discuss in more detail later). The function we've used here is one of the simplest, `fetchrow_array`, which returns the values for a single matching row in the database as an array. `testac` only defines two fields per row, so we assign the two values to `$id` and `$name`.

This only retrieves one result though, corresponding to one matched record in the database. Usually, there'll be more than one, so to get all our results, we need to call `fetchrow_array` several times, once for each result. When there are no more rows to retrieve, `fetch` will return the undefined value, so we can write our loop like this:

```
while (( $id, $name) = $sth ->fetchrow_array) {
   print "$name has ID $id\n";
}
```

Once we've finished retrieving all the results (or all the ones we want – we're not obliged to retrieve all the matches if we only want some of them), we call `finish` to inform the database that we're done and then disconnect:

```
$sth->finish();

$dbh->disconnect || die "Failed to disconnect\n";
```

The only drawback to this example is that the one table in the test database, `testac` is quite small and only has one entry. How can we learn very much using that? We're going to have to build our own, and use that instead.

Creating a Table

Creating a table within a database is actually no different from retrieving data from one. It remains simply a matter of building the correct SQL statement and executing it, although in this case there are no actual results to retrieve.

Earlier on, we gave the example of a check-in desk at an airport that uses a database to keep tabs on who has and hasn't checked in. Let's start building up that functionality up now, beginning with a table that all our passengers' records will be stored on. Our check-in table is going to be quite simplistic. We're going to need to keep track of:

- passengers' first and last names
- destination
- whether or not they've checked in yet
- how many bags they've checked in.

Let's go straight ahead and do that - we'll figure out how we did it in a minute.

Try It Out - Creating The Check-in Desk Table

The code to create the table is very similar to the code we queried the test table with:

```
#!\usr\bin\perl
#create.plx

use warnings;
use strict;
use DBI;

my ($dbh, $sth);

$dbh=DBI->connect('dbi:mysql:test','root','elephant') ||
   die "Error opening database: $DBI::errstr\n";

$sth=$dbh->prepare("CREATE TABLE checkin (
          id        INTEGER AUTO_INCREMENT PRIMARY KEY,
         firstname     VARCHAR(32) NOT NULL,
         lastname     VARCHAR(32) NOT NULL,
         checkedin     INTEGER,
         numberofbags   INTEGER,
         destination     VARCHAR(32) NOT NULL)");
```

```
   $sth->execute();           # execute the statement

   $sth->finish();            # finish the execution
   print "All done\n";
   $dbh->disconnect || die "Failed to disconnect\n";
```

All being well, we'll get a little note saying All done from `create.plx`, but to verify that it's done what we wanted it to, we need to go and talk to MySQL directly.

From a command prompt window then, go to `MySQL_root_directory\bin` and type `mysql` to start the MySQL monitor. You should get a welcome message telling you your connection id and to type 'help' for help.

Now our aim has been to create a table called `checkin` in the test database, so we'll target that database and see if it's there:

```
mysql> use test
Database changed
mysql> show tables;
+----------------+
| Tables in test |
+----------------+
| checkin        |
| testac         |
+----------------+
2 rows in one set (0.01 sec)
mysql>
```

Success! Our check-in table has been created, but does it contain the fields we specified?

```
mysql> show columns from checkin;
+--------------+-------------+------+-----+---------+----------------+
| Field        | Type        | Null | Key | Default | Extra          |
+--------------+-------------+------+-----+---------+----------------+
| id           | int(11)     |      | PRI | 0       | auto_increment |
| firstname    | varchar(32) |      |     |         |                |
| lastname     | varchar(32) |      |     |         |                |
| checkedin    | int(11)     | YES  |     | NULL    |                |
| numberofbags | int(11)     | YES  |     | NULL    |                |
| destination  | varchar(32) |      |     |         |                |
+--------------+-------------+------+-----+---------+----------------+
6 rows in set (0.02 sec)
mysql>
```

Yes, they are there, too. Type `quit` to quit MySQL monitor and let's look at why we've done what we've done.

How It Works

The bulk of the program is pretty much the same as we've seen before, so let's jump straight to the SQL statement. This consists of one call to `CREATE TABLE`, which as you may imagine, does exactly that in the database we're currently connected to.

```
   $sth=$dbh->prepare("CREATE TABLE checkin (
```

The full syntax for `CREATE TABLE` is somewhat overwhelming at this level, so we'll cut it down somewhat:

```
CREATE TABLE table_name (
   column1_name  data_type   notes,
   column2_name  ...,
   ...
   columnx_name  data_type   notes)
```

Both `table_name` and `column_names` can be defined as you want, but just as it is when we give names to variables, something descriptive is always of help. It's easier to figure what a table called 'Horse_Feed' holds than a table called 'A13'. Note that the valid characters in a table or column name are also the same as for a scalar\array\hash\etc. name.

As it would suggest, `data_type` specifies exactly what kind of data each column can hold. MySQL actually recognizes 27 different data types – for our purposes, five are worth mentioning:

- `INTEGER(`*`max_length`*`)` – hold integers only.
- `FLOAT(`*`max_length, number_of_decimals`*`)` – holds floating-point numbers with a given number of decimal places to it.
- `VARCHAR(`*`max_length`*`)` – holds a string of up to *`max_length`* characters.
- `DATE` – holds date values in the form `YYYY-MM-DD`.
- `TIME` – holds time values in the form `(-)HHH:MM:SS`.

We'll come to the 'notes' part of the column declarations as we meet them. If you're interested, the full syntax for `CREATE TABLE` can be found in section 7.7 of the MySQL manual.

For each table, it's recommended that you specify one column in which the value is unique for every single record, thus avoiding the confusion of having two records with exactly the same value for all fields. This column is known as the **primary key** for the table and we use `id` for this purpose:

```
id      INTEGER AUTO_INCREMENT PRIMARY KEY,
```

Rather than having to worry about the next unique value to put in this column however, we've told MySQL that we want it to `AUTO_INCREMENT` from its default of 1. From now, MySQL will keep track of the next value to give `id` whenever we add a new record into our table.

The next two columns represent the name of our passenger who is due to turn up at check-in. These are both variable length strings of no more than 32 characters in length and have both been given the value `NOT NULL`.

```
firstname   VARCHAR(32) NOT NULL,
lastname    VARCHAR(32) NOT NULL,
```

`NULL` is a special kind of value in the world of databases, in that it represents no value at all, in a similar fashion to a variable that hasn't been given a value being undefined. No matter what else you've declared a column to be, you can always declare whether or not it can contain `NULL` values.

Next, we have the columns representing whether our passengers have checked in or not and how many bags they brought with them:

```
checkedin    INTEGER,
numberofbags    INTEGER,
```

Both of these fields have a default value of `NULL` – representing the period during which we know they're due to check in but haven't yet arrived. Later we can check against this default value to see who is late for boarding.

Last, but not least, we have the column representing the passenger's destination:

```
destination    VARCHAR(32) NOT NULL)");
```

Populating a Table with Information

Okay, we've now created our `checkin` table, but it's not going to be any use without some information inside it, so we turn to another SQL command, `INSERT`. This is SQL's way of saying to a database, "Create a new record in the table I've given you and fill in the values as I've specified." If fields that have a default value aren't given a value, then they'll be filled automatically.

For example, let's add in our first passenger.

Try It Out - Simple Inserts

John Smith is soon to be boarding the plane to New York. Before he checks in, we need to add him to the `checkin` table:

```
#!\usr\bin\perl
#insert1.plx

use warnings;
use strict;
use DBI;

my ($dbh, $rows);

$dbh=DBI->connect('dbi:mysql:test','root','elephant')
          || die "Error opening database: $DBI::errstr\n";

$rows=$dbh->do
             ("INSERT INTO checkin (firstname, lastname, destination)
               VALUES              ('John',    'Smith',  'Glasgow')")
           || die "Couldn't insert record : $DBI::errstr";

print "$rows row(s) added to checkin\n";

$dbh->disconnect || die "Failed to disconnect\n";
```

Again, we won't see much from this program unless something goes wrong:

```
>perl insert1.plx
1 row(s) added to checkin
>
```

So we need to go back into the MySQL monitor and check our table as its new entry. Assuming that we've just started it up then:

```
mysql> use test
Database changed
mysql> select * from checkin;
+----+-----------+----------+-----------+--------------+-------------+
| id | firstname | lastname | checkedin | numberofbags | destination |
+----+-----------+----------+-----------+--------------+-------------+
|  1 | John      | Smith    |      NULL |         NULL | Glasgow     |
+----+-----------+----------+-----------+--------------+-------------+
1 row in set (0.09 sec)
mysql>
```

Again, success.

How It Works

We've done things a little differently here. Once we've connected to `checkin` in the usual manner, there's no sign of the familiar `prepare()`, `execute()`, and `finish()` – just something called `do()` in their place:

```
$rows=$dbh->do
```

Now for any SQL statements that don't return a value – in this chapter we'll see `CREATE`, `INSERT`, `UPDATE`, `DELETE` and `DROP` – DBI provides the `do` method, which prepares and executes a statement all in one go. We don't need to call `finish` either because this query doesn't return values. Why didn't we use this for `create.plx` then? Well, okay. I admit we could have done, but let's take things one step at a time….

`do` doesn't return a statement handle either, since it's not going to be reused. Instead, its return value is either undefined (when an error occurs), the number of rows affected, or the value `0E0` if the statement just didn't affect any rows.

> *This rather strange looking number evaluates to zero numerically but as true in a string context, It therefore won't be confused with `undef` and avoids the possibility of causing errors to be reported in examples like the one above.*

Of course, within our call to `do`, we have our SQL `INSERT`:

```
      ("INSERT INTO checkin (firstname, lastname, destination)
        VALUES              ('John',    'Smith',  'Glasgow')")
    || die "Couldn't insert record : $DBI::errstr";
```

As you can see, the syntax is quite straightforward. We state which table columns we wish to assign entries to, and then supply the respective values. As noted earlier, the other three entries in our table all have default values and are filled in automatically:

```
print "$rows row(s) added to checkin\n";
```

Finally, we can print out the number of rows in the table that our `INSERT` has affected. Of course, this is just one.

A Note on Quoting

A word of warning here before we carry on: If we try to put our SQL statements into strings before preparing them, we can run into problems, especially if the values contain quotes or other characters that are significant to Perl. The trouble is, we've now got two sets of quotes to deal with – Perl's and SQL's. If we try to run the following, for example, we'll get an error:

```
my $last="O'Connor";
my $sth=$dbh->prepare("INSERT INTO checkin (firstname, lastname, destination)
                       VALUES              ('John',    '$last',   'Glasgow')");
```

The problem is that the value of `$last` contains a single quote, which is illegal within the single quote delimiters of our SQL statement, which now ends:

```
VALUES ('John',    'O'Connor',   'Glasgow')");
```

Perhaps we could use double quotes in the SQL statement and get:

```
VALUES ('John',    "O'Connor",   'Glasgow')");
```

which is (depending on your database) usually legal. This would be fine, except that we're already using double quotes in our Perl code, so we'd get a Perl syntax error instead. Could we replace the outer double quotes with `qq`? No we can't – as soon as we interpolate, we're back to a SQL syntax error. So, how do we handle variables when we can't control their contents?

Fortunately, DBI provides the `quote` method specifically to solve this problem. We can use `quote` to make our values safe from rogue quotes and other symbols, before interpolating them. Here's how we could use it to fix the problems in the code fragment above:

```
my $last=$dbh->("O'Connor");
my $sth=$dbh->prepare("INSERT INTO checkin (firstname, lastname, destination)
                       VALUES              ('John',    $last,   'Glasgow')");
```

The `quote` method makes our parameters safe and also deals with adding the surrounding quotes, so we do not want them in the actual SQL statement. Note that in the second example there are no quotes around `$last` in the SQL.

Do or Do Not

Let's go back to that `do` statement. In actual fact, `do` calls `prepare` and `execute` internally, so it's really just the same if we write the two separately:

```
my $sth=$dbh->prepare
             ("INSERT INTO checkin (firstname, lastname, destination)
               VALUES              ('John',    'Smith',  'Glasgow')");

$sth->execute() || die "Couldn't insert record : $DBI::errstr";
```

Which of them we choose depends on whether we intend to reuse the statement. If we create a table, the chances are that we'll only do it once. If we want to change records though, we might want to do it repeatedly; preparing and saving the statement handle will therefore be to our advantage. The example above is much too specific to be worth saving, but we can use **placeholders** to create a generic insert statement.

Placeholders are SQL's version of interpolation. While it isn't as powerful as Perl's interpolation (it can only substitute values, not arbitrary parts of the SQL string), it provides us with another way to avoid quoting problems. Better still, it lets us cache and reuse statements, because the substitution happens at execution time rather than during preparation. Instead of writing explicit values into our SQL statements, or using interpolation to substitute them, we replace the explicit parameter with a question mark (?). That's called the placeholder, because it's holding the place of a real value. We then supply the missing value as a parameter of the `execute` method:

```
my $sth=$dbh->prepare(
             ("INSERT INTO checkin (firstname, lastname, destination)
               VALUES              (?        , ?       , ?          )");

$sth->execute('John', 'Smith', 'Glasgow')
      || die "Couldn't insert record : $DBI::errstr";
```

Using placeholders allows us to prepare a statement once and then reuse it many times with different parameters – we can do this because the substitution takes place at the execution stage rather that the preparation stage. It also takes care of quoting problems for us, since we no longer need to do our own interpolation.

Note, however, that values that are substituted for placeholders have quotes added automatically (just as with the `quote` method discussed above), so we don't need to do it ourselves. Indeed, if we were to put quotes round the placeholder in the original SQL statement it wouldn't work, except to insert the string "?":

```
my $sth=$dbh->prepare(
             ("INSERT INTO checkin (firstname, lastname, destination)
               VALUES              ('?'      , ?       , ?          )");
              # this would insert "?" into firstname
```

There are limits to what we can and cannot use as a placeholder. In particular, we cannot substitute multiple values into a single placeholder. Different database servers provide different advanced query features, so it's worth checking what's available (and what complies with the SQL specification, if we want to make sure that our SQL is portable).

If we could guarantee a standard format for that information in a file, say, then we could parse that information and insert it into records in the database.

Suppose we had a text file holding all these peoples' details, one per line, in the format:

```
firstname:lastname:destination,
```

We could use this program, passing it the text file:

```
#!\usr\bin\perl
#insert2.plx

use warnings;
use strict;
use DBI;

my ($dbh, $sth, $firstname, $lastname, $destination, $rows);

$dbh=DBI->connect('dbi:mysql:test','root','elephant') ||
   die "Error opening database: $DBI::errstr\n";

$sth=$dbh->prepare
             ("INSERT INTO checkin (firstname, lastname, destination)
               VALUES              (?        , ?       , ?          )");

$rows=0;

while (<>) {
  chomp;
  ($firstname, $lastname, $destination) = split(/:/);
  $sth->execute($firstname, $lastname, $destination)
      || die "Couldn't insert record : $DBI::errstr";

  $rows+=$sth->rows();
}

print "$rows new rows have been added to checkin";

$dbh->disconnect || die "Failed to disconnect\n";
```

and sure enough, our data will appear in the table.

Note that in the code download that's available for this book at `http://www.wrox.com`*, you will find a file called* `passlist.txt` *that contains about twenty records to populate our sample table with. Just call:*

```
>perl insert2.plx passlist.txt
```

We'll be using this populated version of the database for the rest of the chapter.

Keeping the Table up to Date

Now we have our passengers in our table, what happens when one of them actually makes it to check-in (or the plane originally going to Glasgow gets diverted to Edinburgh)? Our data can't remain sacrosanct – we have to be able to change it. We can do this with the SQL `UPDATE` command.

Unlike the `INSERT` command (which adds new data to the table) and the `DELETE` command (which removes data), `UPDATE` exists solely to modify the information *already* stored in our tables. It's also one of the more powerful statements we'll see in this chapter, allowing you to change multiple rows of data in one fell swoop. Before we get to that level of complexity though, let's address one of the situations we've got above. What exactly *will* happen if one of the passengers, say, `Peter Morgan`, makes it to check-in?

Two things. We need to add the number of bags that he's checked in to his record and also some value to the `checkedin` field to denote that he has arrived. Any non-zero, non-`NULL` value would equate to `True` (did you notice there was no Boolean data type for a field?), but we could make it a useful value in some other way as well. Let's say the number of passengers, including this one, that have checked in since the desk opened. That will do nicely if we ever want to expand on the simple routines we're developing here.

The corresponding SQL statement is:

```
UPDATE checkin
SET    checkedin = 1,
       numberofbags = 2
WHERE  firstname = 'Peter' AND lastname = 'Morgan'
```

and once again, that would sit inside the same code harness as `update1.plx`. Because `UPDATE` doesn't actually return any values, we could also sit this statement inside a call to `$dbh->do()` – it would still work, but would lack the facility of using placeholders (and more besides). Learning to use `UPDATE` is not hard. Its syntax is:

```
UPDATE table-name
SET    column_name1 = expression1,
       column_namex = .....
       ...
                    = expressionz
[ WHERE condition_expressions ]
```

That's fairly straightforward. The `WHERE` clause at the end of the statement is optional, meaning that those updates omitting `WHERE` are applied to all the records in the table – but its real power comes from being able to `SET` fields to the results of a query, which we've written and embedded in the `UPDATE`. For example, let's suppose passenger `Bill Gates` has switched his destination to match that of `Henry Rollins`. Rather than pulling Henry's destination from the table first and then matching it in the UPDATE, we can combine the two as follows:

```
UPDATE checkin
SET    destination =
       (SELECT destination
        FROM checkin
        WHERE firstname='Henry' AND lastname='Rollins')
WHERE  firstname='Bill' AND lastname='Gates'
```

Pulling Values from the Database

So now we come the raison d'être of databases – the ability to pull arbitrary data from your tables, cross-referenced and validity backed-up if necessary. As we've seen already, we can extract information from a SQL database with a `SELECT` statement, which selects and returns complete records or selected columns that match our criteria.

At this low level, `SELECT` doesn't threaten to be too complex. The syntax for the statement is as follows:

```
SELECT column1, column2, ...., columnx
FROM table
WHERE condition_to_be_met
[ORDER BY column][GROUP by column]
```

We know that it's going to return a set of records as well, so we know in advance that we'll also need to `prepare()`, `execute()`,and `finish()` these statements. Let's design a test program for our `SELECT` statements then and continue to add in the code we need to retrieve and display the results in this section. In fact, we already have `querytest.plx` from a while ago to do that for us:

```
#!\usr\bin\perl
#querytest.plx

use warnings;
use strict;
use DBI;

my ($dbh, $sth);

$dbh=DBI->connect('dbi:mysql:test','root','elephant') ||
   die "Error opening database: $DBI::errstr\n";

$sth=$dbh->prepare("<SQL_SELECT_Statement_here>") ||
   die "Prepare failed: $DBI::errstr\n";

$sth->execute() ||
   die "Couldn't execute query: $DBI::errstr\n";

my $matches=$sth->rows();
unless ($matches) {
   print "Sorry, there are no matches\n";
} else {
   print "$matches matches found:\n";
        while (my @row = $sth ->fetchrow_array) {
            print "@row\n";
        }
}

$sth->finish();

$dbh->disconnect || die "Failed to disconnect\n";
```

There's also a quick routine in there that prints out how well our `SELECT` statements fared against the table. We've seen the `rows()` function before, but it's worth noting that while it does work against MySQL, it doesn't with some other database drivers. If we don't have `rows` available to us, we can still produce a `matches found` message like the one above, only we'll have to either make the SQL statement count matches for us with a `COUNT` (for example: `SELECT COUNT(*) FROM tablename WHERE ...`) or count the returned rows before displaying them.

One of the easiest queries we can make is one that will get our table to return every row that matches the criteria given after `WHERE`. For example, let's return all the details of those passengers who've visited the check-in so far. That is, all the passengers whose `checkedin` field is not `NULL`:

```
#!\usr\bin\perl
#selectstar.plx

use warnings;
use strict;
use DBI;

my ($dbh, $sth);
```

```
$dbh=DBI->connect('dbi:mysql:test','root','elephant') ||
   die "Error opening database: $DBI::errstr\n";

$sth=$dbh->prepare("SELECT * from checkin WHERE checkedin IS NOT NULL;") ||
   die "Prepare failed: $DBI::errstr\n";

$sth->execute() ||
   die "Couldn't execute query: $DBI::errstr\n";

my $matches=$sth->rows();
unless ($matches) {
   print "Sorry, there are no matches\n";
} else {
   print "$matches matches found:\n";
        while (my @row = $sth ->fetchrow_array) {
            print "@row\n";
        }
     }
}

$sth->finish();

$dbh->disconnect || die "Failed to disconnect\n";
```

Sure enough, we've only had one person come through check-in so far, and accordingly, we get:

```
>perl selectstar.plx
1 matches found:
6 Peter Morgan 1 2 Thailand
>
```

Similarly, if we just wanted to know the details of those people bound for Japan, our statement would read:

```
SELECT * from checkin WHERE destination='Japan';
```

Now putting this in `selectstar.plx` and running it results in quite a bit of a mess, thanks to the NULL fields in most of the table's records. However we can choose, for instance, just to retrieve the first and last names of the passengers, possibly to send out a tannoy message for them to hurry up:

```
SELECT firstname, lastname from checkin WHERE destination='Japan';
```

This is much tidier:

```
>perl selectstar.plx
4 matches found:
Richard Collins
Simon Cozens
Larry Wall
Brian Walsh
>
```

Now although it appears our little program has ordered this result by surname, in actual fact that's just the order in which those records appear in my test database (they might appear differently in yours). We can however ensure that returned data is properly ordered, by appending an `ORDER BY` clause to the `SELECT` statement.

```
SELECT firstname, lastname from checkin WHERE destination='Japan'
ORDER BY firstname;
```

We can now be sure that our return values will always be ordered:

```
>perl selectstar.plx
4 matches found:
Brian Walsh
Larry Wall
Richard Collins
Simon Cozens
>
```

Where Do the Records Go?

Until now, we've been concentrating on how to interact with the database and getting it to send us the information we want. But how exactly are we picking up that information? We've already been using one of the possibilities, `fetchrow_array()` in the background without really saying what it does, but now we'll look at it and its siblings in more detail.

Fetching Rows into Arrays

The simplest way to retrieve results from a `DBI` query is a row at a time, using the `fetchrow_array` method, which returns the requested columns in a row as an array, with the order of elements corresponding to the order in which they were asked for in the original `SELECT` statement:

```
@record  = $sth->fetchrow_array;
```

Taking our `selectstar` examples above then, the `firstname` field will always appear in `$record[0]` and the `lastname` field in `$record[1]`. If we ask for all the columns with `SELECT *`, then the order is the same as that of the table definition in the database. This is because `Selectstar` uses a very simple fetch loop to retrieve the rows from the query in turn and then simply prints them out:

```
while (@row = $sth ->fetchrow_array) {
   print "@row\n";
}
```

Of course, this isn't all that great if we need to process the information and then update the database on the basis of what we've found out. So here's an alternate loop, which stores all the results in an array of arrays, ready for further inspection:

```
my @results=();
while (my @result=$sth->fetchrow_array){
   push @results,\@result;
}
```

We can achieve a similar but subtly different effect with the `fetchall_arrayref` method.

Fetching Rows into References

If we want to use the results of `fetchrow_array` immediately and don't need to keep them (as `Selectstar` does), we can use `fetchrow_arrayref` instead. Rather than creating and returning a brand new array each time we call it, this returns a reference to an internal array, which it reuses for every subsequent row:

Reusing the array means that perl can save time and memory by not creating a fresh one each time. The difference in the `while` loop we're using to dump the results is likewise relatively small:

```
while (my $row_ref = $sth ->fetchrow_arrayref) {
    print "@{$row_ref}\n";
}
```

However, if we tried to use this technique to store the array in a bigger 'results' array as in the preceding example, we'd end up with an array containing multiple copies of the last result returned from `fetchrow_arrayref`, since we'd have just stored the same array reference multiple times. That is not what we had in mind.

Sometimes we may not know what columns we've asked for, for example, if we're accepting an arbitrary column list and inserting it into the query:

```
my @columns=qw(firstname lastname destination);
my $query = "SELECT ".join(',',@columns)." FROM checkin";
$sth=$dbh->prepare("$query");
```

If we're writing a subroutine that gets called by some other code and only need to return matches, we can let the caller worry about column orders. On the other hand, if we want to interpret the results ourselves, we have a problem, as we don't know the column names.

One way to get round this problem is to ask the statement itself, using the `NUM_OF_FIELDS` and `NAME` attributes (see 'Extracting Column Information from Statements') for details. However, not every database driver supports these.

Fortunately, we can use the `fetchrow_hashref` method to solve this problem for us, by returning a row as a hash (rather than an array), with the column names as the keys and the retrieved columns as the values:

```
foreach (my $href=$sth->fetchrow_hashref) {
   foreach (keys %{$href}) {
      print "$_ => $href->{$_}\n";
   }
}
```

Because of the extra work involved in creating a hash, this isn't as efficient as `fetchrow_array` (which may be significant if performance is an issue). `DBI` doesn't define whether the returned hash is unique or reused internally for each new row (currently a new hash is created each time), so we shouldn't rely on this. If we want to hang on to it, we must copy it to a new hash:

```
my @results=();
foreach (my $href=$sth->fetchrow_hashref) {
   my %result=%{ $href }; #copy returned hash
   push @results,\%result;
}
```

Of course, if we don't need to keep the results of `fetchrow_hashref` (perhaps because we're able to use them as we retrieve them), we don't need to resort to copying the returned values. This step's only really necessary if we plan to collect data later on, for processing in some other way.

We can also use the `fetchall_array` method to create an array of hashes instead of an array of arrays, as we'll see shortly.

Fetching a Single Value

If we want to retrieve a single value from a database, we don't need to write a loop. Instead, we can just retrieve the scalar value direct from the function call (by putting an array index of `[0]` after it) and return the result to the caller. This subroutine makes use of that fact to return the number of entries in the table of our choice in an efficient manner:

```
sub count_table {
   my ($dbh,$table,$sql,@values)=@_;

   $sql="" unless defined $sql; #suppress undef warnings
   my $sth=$dbh->prepare("SELECT COUNT(*) FROM $table $sql")
      or return undef;
   $sth->execute(@values) or return undef;

   # return the result of the count
   return ($sth->fetchrow_array())[0];
}
```

Because the table name can't be specified through a placeholder (SQL doesn't allow a placeholder here), we use interpolation instead. We also allow the caller to supply an arbitrary trailing SQL clause (the `$sql` parameter) to further narrow the criteria of the count and a values array (the `@values` parameter) to fill any placeholders present in the clause. Here are some examples of how we could call this subroutine with different parameters:

```
print count_table($dbh,"checkin");
print count_table($dbh,"checkin","WHERE destination='San Diego'");
print count_table($dbh,"checkin","WHERE destination=?","Japan");
```

Note that if we replaced `prepare` with `prepare_cached` in the subroutine, then the last example would allow us to cache a generic `WHERE` clause that would also work for `London`.

Binding Columns

As well as the `fetch` family of methods, `DBI` also allows you to '`bind`' variables to the results of a `fetch`, so that they automatically receive the columns of the result when `fetch` is called. There are two ways to bind variables to columns, either individually or all at once. This is how we might bind the columns for our location query one at a time, using the `bind_col` method:

```
$sth=$dbh->prepare("SELECT firstname, lastname
                    FROM checkin WHERE destination=?;")
                     ||   die "Prepare failed: $DBI::errstr\n";

$sth->execute('Japan') or die "Error...";
```

```
$sth->bind_col(1,\$first ); #bind column 1 to $first
$sth->bind_col(2,\$second); #bind column 2 to $second

print "Match: $second, $first\n" while $sth->fetch();
```

Binding columns doesn't provide us with any performance increase over doing the same with an array, but it does allow us to write more legible code. We can also bind all our columns at the same time using the `bind_columns` method.

As a more developed example, here's a subroutine that returns the first matching result. It takes an array reference to a list of columns, and an optional array reference to a list of scalar references. It also allows some arbitrary SQL (like the `count_table` subroutine we developed earlier) and appends a `LIMIT 1` clause, which returns only the *first* matching row:

```
sub get_first_row {
   my ($dbh,$table,$columns,$results,$sql,@values)=@_;
   my ($col_list, sth);

   $sql="" unless defined $sql; #suppress undef warnings

   $col_list = join(',',@{$columns});
   $sth=$dbh->prepare("
      SELECT $col_list
      FROM $table
      $sql
      LIMIT 1
   ") or return undef;
   $sth->execute(@values) or return undef;

   $sth->bind_columns(@{$results}) if defined $results;

   return $sth->fetchrow_array; #return array;
}
```

We can call this subroutine to return a conventional array:

```
my  @columns=('firstname','lastname');
my @result=get_first_row($dbh,"checkin",\@columns);
print "Match: $result[1], $result[0]\n";
```

We can also bind the columns by passing an array of scalar references, and then use the scalars for a more legible `print` statement:

```
my ($first,$last);
my @columns=('first','last');
my @return_values=(\$first,\$last);
get_first_row($dbh,"checkin",\@columns,\@return_values);

print "Match: $last, $first\n";
```

Fetching All Results

We've already mentioned the `fetchall_arrayref` method several times. This retrieves all the results of a query at one time, as one large array. The advantage of this is that we can do things like count the number of results *before* we use them (portably – unlike with `rows`, which may not exist) and access the results out of order. For example, we could sort them before printing them out. The principal disadvantage is that the array created by `fetchall_arrayref` can take up an awful lot of memory (possibly more than the machine actually has) if a lot of matches are returned, so we must use it with great caution.

We've already seen how to create an array of arrays and an array of hashes for ourselves, in 'Fetching Rows'. This example does the same thing in one go using `fetchall_arrayref`:

```
#retrieve all results as an array of arrays
my @results=$sth->fetchall_arrayref();
my @results=$sth->fetchall_arrayref([]); # supply empty list reference

#retrieve all results as an array of hashes
my @results=$sth->fetchall_arrayref({}); # supply empty hash reference
```

We can limit the results returned in each row's array or hash by passing in an array slice or predefined list of hash keys:

```
#retrieve the first three columns of each row in an array of arrays
my @results=$sth->fetchall_arrayref([0..2]);

#retrieve the first and last two columns of each row in an array of arrays
my @results=$sth->fetchall_arrayref([0,-2,-1);

#retrieve the first and last name columns as a array of hashes
my @results=$sth->fetchall_arrayref({first=>1,last=>1});
```

Note that although these examples are all perfectly good, none are as efficient as phrasing the SQL query so as to only return the desired values in the first place – we're making the database do work that just isn't required.

If we're making many similar queries and the data we don't require is relatively small, then reusing a saved statement handle (with `fetchall_arrayref`) is probably more efficient. Otherwise we're likely to incur a performance loss, which we could avoid by rewriting our code to use SQL queries that retrieve less data.

One final thing to remember – the array created by `fetchall_arrayref` is reused on subsequent calls, so if we use the method more than once, we'll lose the original set of results. That is, unless we take steps to copy the results elsewhere first.

Extracting Column Information from Statements

Once statement handles have been created, we can extract information from them, to help us deal with the results that they return. Not all databases and database drivers support all of these features, but with those that do, they can be very convenient. Here are the most useful of the attributes that we can retrieve from a statement handle:

Finding the Number of Columns

We can find the number of columns in a set of results using the `NUM_OF_FIELDS` attribute:

```
my $columns=$sth->{'NUM_OF_FIELDS'};
```

Finding the Name of a Column

We can find the column name for a particular column in a statement using the `NAME` attribute:

```
my @column_names=$sth->{'NAME'};
my $first_column_name=$sth->{'NAME'}->[0];
```

Column names can vary in case, and unlike SQL, Perl cares about case (when a column name is used as a hash key, for example). So we can also retrieve case-adjusted column names with:

```
my @upper_cased_names=$sth->{'NAME_uc'};
my @lower_cased_names=$sth->{'NAME_lc'};
```

Note that this returns the name of the columns returned by this query. It has nothing to do with the order of columns in the queried table or tables, unless we selected all columns with `SELECT *`.

Finding the Number of Placeholders

We can find the number of placeholders in a statement with the `NUM_OF_PARAMS` attribute:

```
my $parameters=$sth->{'NUM_OF_PARAMS'};
```

This is the number of parameters that must be bound (using `bind_param`) or otherwise passed to the `execute` method. Failing to supply enough parameters to `execute` will likely result in an invalid SQL statement and cause `DBI` to return an error. This is mostly useful for debugging purposes.

Retrieving the Original Text of the Statement

We can get back the original text of the statement by using the `Statement` attribute:

```
my $sql=$sth->{'Statement'};
```

More Advanced Attributes

These include:

- `NULLABLE` – this returns an array of values describing whether the corresponding field number can be NULL (in which case, the array will contain a zero for that field) or must contain a defined value (in which case, the array will contain a non-zero value).
- `PRECISION` – this returns an array of values describing the precision of each column. The definition of `PRECISION` varies, depending on the column type and the database. For a string, it's usually the length; for a number, it's the width as it would be displayed.
- `SCALE` – this returns an array of values describing the scale of each column. The definition of `SCALE` varies, depending on the column type and the database; it is usually the number of decimal places for floating-point numbers and zero for any other column type.
- `TYPE` – this returns an array of values describing the type of each column – integer, string, floating-point number, date, etc.

We can also set flags on statement handles, just as we can with database handles, to switch tracing on or off, or change DBI's error handling, for example. See the discussion on flags and attributes earlier in the chapter for a description of these flags and what they do.

Removing Information from the Table

Just before we end this crash course in DBI and SQL databases, we need to deal with two simple questions:

- How do I remove data from a table?
- How do I remove a table from a database?

The latter question actually has one of the simplest answers to all the questions we've asked of SQL – frighteningly so. If you execute the SQL statement, `DROP TABLE table_name`, the relevant table will **disappear** without a trace and without any query for confirmation. It's a very easy way to do a potentially disastrous thing, so be *very* careful before you even consider using the word `DROP` anywhere in the vicinity of a DBI program.

The slightly less drastic action of removing records from tables is actioned by the keyword `DELETE`, which has the following syntax:

```
DELETE FROM table_name
[WHERE condition]
```

`DELETE` will search out the records whose fields match the condition and completely remove their entries from the table. Going back to our check-in desk example – if the plane to Edinburgh has just taken off, there's no need to retain any of the information about passengers on that plane, so by telling MySQL:

```
DELETE FROM checkin
WHERE destination = 'Edinburgh';
```

we'll lose all the entries in our table associated with that flight to Edinburgh.

Things We've Not Covered

So that's it for DBI in this book – there are still huge amounts of material that we've not covered here; many of the areas we've touched upon really can't be done justice without a whole book to themselves. A few examples are:

- **The rest of the SQL language**
  Most of the commands we've looked at in this chapter haven't been covered exhaustively – nor have we covered all the commands.
- **Working with relational data**
  Ironically, one of the big points for relational databases is that they can join disparate data over separate tables and relate many tables to each other. We concentrated on the basics of working with a single table rather than going straight for the really messy stuff.

- **Stored Procedures**
  Most database servers have the ability to save SQL code in a compiled format internally and run it on demand. These compiled (groups of) SQL statements are known as **stored procedures**.

- **DBI working with ODBC and ADO**
  Windows users will be most disappointed here I imagine. This section had to be universal to all DBI users.

- **Transactions**
  As mentioned earlier, transactions are what really make the world go round, and transaction-enabled databases, with their commits and rollbacks, are what makes transactions available.

- **Databases and CGI**
  Almost every large-scale website on the internet generates its content by querying a database, formatting the returned results and inserting them into HTML.

The easiest place to go and start reading up on any or all of these subjects is the Internet. Start with http://www.perl.com and other websites mentioned in the introduction and work out from there. Or, if you prefer paperware products, the O'Reilly range of Perl books is quite comprehensive. Look out in particular for *Programming The Perl DBI* by Tim Bunce and Alligator Descartes (O'Reilly, ISBN 1565926994).

Summary

In this chapter, we've had a crash course in working with data in flat files through DBM and tied hashes and also with DBI and relational databases.

In the first half of the chapter, we looked at how to install and apply Perl's DBM (DataBase Manager) modules, seeing how to specify and prioritize certain modules' use over others, and optimize the portability of our programs. We looked at access and manipulation of flat data files transparently via a tied hash, how to store more complex structures (by joining and splitting scalars) and finally saw how we can use the MLDBM module to extend transparency to those complex data structures.

Meanwhile, in the second half, we saw the difference between the files that DBM users work with and the full-blown commercial relational database servers that the majority of the world's data lives on. We were introduced to DBI, a universal translator to many such relational databases and the drivers to which it speaks on behalf of the programmer. We also learnt how to install, set up and work with MySQL, a free RDBMS that everyone can use.

Once everything was set up, we saw how to create new database tables and populate them with data, update, and delete, query, and retrieve that data, and finally how to delete tables as a whole. In greater depth, we looked at how we might deal with the data once we had retrieved it from the database, the shortcuts offered us by placeholders and statement caching and the problems to work around when working with many sets of quotes in a SQL statement.