CHAPTER 19

DBM and mod_perl

Some of the earliest databases implemented on Unix were Database Management (DBM) files, and many are still in use today. As of this writing, the Berkeley DB is the most powerful DBM implementation. Berkeley DB is available at *http://www.sleepycat.com/*. If you need a light database with an easy API, using simple key-value pairs to store and manipulate a relatively small number of records, DBM is the solution that you should consider first.

With DBM, it is rare to read the whole database into memory. Combine this feature with the use of smart storage techniques, and DBM files can be manipulated much faster than flat files. Flat-file databases can be very slow when the number of records starts to grow into the thousands, especially for insert, update, and delete operations. Sort algorithms on flat files can also be very time-consuming.

The maximum practical size of a DBM database depends on many factors, such as your data, your hardware, and the desired response times. But as a rough guide, consider 5,000 to 10,000 records to be reasonable.

We will talk mostly about Berkeley DB Version 1.x, as it provides the best functionality while having good speed and almost no limitations. Other implementations might be faster in some cases, but they are limited either in the length of the maximum value or the total number of records.

There are a number of Perl interfaces to the major DBM implementations, such as `DB_File`, `NDBM_File`, `ODBM_File`, `GDBM_File`, and `SDBM_File`. The original Perl module for Berkeley DB was `DB_File`, which was written to interface with Berkeley DB Version 1.85. The newer Perl module for Berkeley DB is `BerkeleyDB`, which was written to interface with Version 2.0 and subsequent releases. Because Berkeley DB Version 2.x has a compatibility API for Version 1.85, you can (and should) build `DB_File` using Version 2.x of Berkeley DB, although `DB_File` will still support only the 1.85 functionality.

Several different indexing algorithms (known also as *access methods*) can be used with DBM implementations:

- The HASH access method gives an O(1) complexity (see sidebar) of search and update, fast insert, and delete, but a slow sort (which you have to implement yourself). HASH is used by almost all DBM implementations.
- The BTREE access method allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree. This allows you to get a sorted sequence of data pairs in O(1) (see sidebar), at the expense of much slower insert, update, and delete operations than is the case with HASH. BTREE is available mostly in Berkeley DB.
- The RECNO access method is more complicated, and enables both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in HASH and BTREE. In this case the key will consist of a record (line) number. RECNO is available mostly in Berkeley DB.
- The QUEUE access method stores fixed-length records with logical record numbers as keys. It is designed for fast inserts at the tail and has a special cursor-consume operation that deletes and returns a record from the head of the queue. The QUEUE access method uses record-level locking. QUEUE is available only in Berkeley DB Version 3.0 and higher.

Big-O Notation

In math, complexity is expressed using *big-O* notation. For a problem of size N:

- A constant-time method is "order 1": O(1)
- A linear-time method is "order N": O(N)
- A quadratic-time method is "order N squared": O(N^2)

For example, a lookup action in a properly implemented hash of size N with random data has a complexity of O(1), because the item is located almost immediately after its hash value is calculated. However, the same action in the list of N items has a complexity of O(N), since on average you have to go through almost all the items in the list before you find what you need.

Most often you will want to use the HASH method, but there are many considerations and your choice may be dictated by your application.

In recent years, DBM databases have been extended to allow you to store more complex values, including data structures. The MLDBM module can store and restore the whole symbol table of your script, including arrays and hashes.

It is important to note that you cannot simply switch a DBM file from one storage algorithm to another. The only way to change the algorithm is to copy all the records one by one into a new DBM file, initialized according to a desired access method. You can use a script like the one shown in Example 19-1.

Example 19-1. btree2hash.pl

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

#
# This script takes as its parameters a list of Berkeley DB
# file(s) which are stored with the DB_BTREE algorithm.  It
# will back them up using the .bak extension and create
# instead DBMs with the same records but stored using the
# DB_HASH algorithm.
#
# Usage: btree2hash.pl filename(s)

use strict;
use DB_File;
use Fcntl;

# @ARGV checks
die "Usage: btree2hash.pl filename(s))\n" unless @ARGV;

for my $filename (@ARGV) {
    die "Can't find $filename: $!"
        unless -e $filename and -r _;

    # First back up the file
    rename "$filename", "$filename.btree"
        or die "can't rename $filename with $filename.btree: $!";

    # tie both DBs (db_hash is a fresh one!)
    tie my %btree , 'DB_File',"$filename.btree", O_RDWR|O_CREAT,
        0660, $DB_BTREE or die "Can't tie $filename.btree: $!";
    tie my %hash ,  'DB_File',"$filename" , O_RDWR|O_CREAT,
        0660, $DB_HASH  or die "Can't tie $filename: $!";

    # copy DB
    %hash = %btree;

    # untie
    untie %btree;
    untie %hash;
}
```

Note that some DBM implementations come with other conversion utilities as well.

mod_perl and DBM

Where does mod_perl fit into the picture? If you need read-only access to a DBM file in your mod_perl code, the operation is much faster if you keep the DBM file open (tied) all the time and therefore ready to be used. We will see an example of this in a moment. This will work with dynamic (read/write) database accesses as well, but you need to use locking and data flushing to avoid data corruption.

It's possible that a process will die, for various reasons. There are a few consequences of this event.

If the program has been using external file locking and the lock is based on the existence of the lock file, the code might be aborted before it has a chance to remove the file. Therefore, the next process that tries to get a lock will wait indefinitely, since the lock file is dead and no one can remove it without manual intervention. Until this lock file is removed, services relying on this lock will stay deactivated. The requests will queue up, and at some point the whole service will become useless as all the processes wait for the lock file. Therefore, this locking technique is not recommended. Instead, an advisory `flock( )` method should be used. With this method, when a process dies, the lock file will be unlocked by the operating system, no matter what.

Another issue lies in the fact that if the DBM files are modified, they have to be properly closed to ensure the integrity of the data in the database. This requires a flushing of the DBM buffers, or just untying of the database. In case the code flow is aborted before the database is flushed to disk, use Perl's `END` block to handle the unexpected situations, like so:

```
END { my_dbm_flush() }
```

Remember that under mod_perl, this will work on each request only for `END` blocks declared in scripts running under `Apache::Registry` and similar handlers. Other Perl handlers need to use the `$r->register_cleanup( )` method:

```
$r->register_cleanup(\&my_dbm_flush);
```

as explained in Chapter 6.

As a rule, your application should be tested very thoroughly before you put it into production to handle important data.

Resource Locking

Database locking is required if more than one process will try to modify the data. In an environment in which there are both reading and writing processes, the reading processes should use locking as well, since it's possible for another process to modify the resource at the same moment, in which case the reading process gets corrupted data.

We distinguish between shared-access and exclusive-access locks. Before doing an operation on the DBM file, an *exclusive* lock request is issued if a read/write access is required. Otherwise, a *shared* lock is issued.

Deadlocks

First let’s make sure that you know how processes work with the CPU. Each process gets a tiny CPU time slice before another process takes over. Usually operating systems use a “round robin” technique to decide which processes should get CPU slices and when. This decision is based on a simple queue, with each process that needs CPU entering the queue at the end of it. Eventually the added process moves to the head of the queue and receives a tiny allotment of CPU time, depending on the processor speed and implementation (think microseconds). After this time slice, if it is still not finished, the process moves to the end of the queue again. Figure 19-1 depicts this process. (Of course, this diagram is a simplified one; in reality various processes have different priorities, so one process may get more CPU time slices than others over the same period of time.)

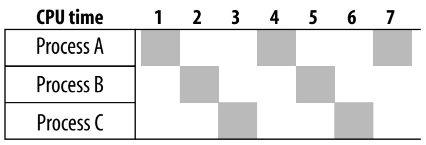

Figure 19-1. CPU time allocation

Now let’s talk about the situation called *deadlock*. If two processes simultaneously try to acquire exclusive locks on two separate resources (databases), a deadlock is possible. Consider this example:

```
sub lock_foo {
    exclusive_lock('DB1');
    exclusive_lock('DB2');
}

sub lock_bar {
    exclusive_lock('DB2');
    exclusive_lock('DB1');
}
```

Suppose process A calls `lock_foo()` and process B calls `lock_bar()` at the same time. Process A locks resource `DB1` and process B locks resource `DB2`. Now suppose process A needs to acquire a lock on `DB2`, and process B needs a lock on `DB1`. Neither of them can proceed, since they each hold the resource needed by the other. This situation is called a deadlock.

Using the same CPU-sharing diagram shown in Figure 19-1, let’s imagine that process A gets an exclusive lock on `DB1` at time slice 1 and process B gets an exclusive lock on `DB2` at time slice 2. Then at time slice 4, process A gets the CPU back, but it cannot do anything because it’s waiting for the lock on `DB2` to be released. The same thing happens to process B at time slice 5. From now on, the two processes will get the CPU, try to get the lock, fail, and wait for the next chance indefinitely.

Deadlock wouldn't be a problem if `lock_foo()` and `lock_bar()` were atomic, which would mean that no other process would get access to the CPU before the whole subroutine was completed. But this never happens, because all the running processes get access to the CPU only for a few milliseconds or even microseconds at a time (called a *time slice*). It usually takes more than one CPU time slice to accomplish even a very simple operation.

For the same reason, this code shouldn't be relied on:

```
sub get_lock {
    sleep 1, until -e $lock_file;
    open LF, $lock_file or die $!;
    return 1;
}
```

The problem with this code is that the test and the action pair aren't atomic. Even if the `-e` test determines that the file doesn't exist, nothing prevents another process from creating the file in between the `-e` test and the next operation that tries to create it. Later we will see how this problem can be resolved.

Exclusive Locking Starvation

If a shared lock request is issued, it is granted immediately if the file is not locked or has another shared lock on it. If the file has an exclusive lock on it, the shared lock request is granted as soon as that lock is removed. The lock status becomes `SHARED` on success.

If an exclusive lock is requested, it is granted as soon as the file becomes unlocked. The lock status becomes `EXCLUSIVE` on success.

If the DB has a shared lock on it, a process that makes an exclusive lock request will poll until there are no reading or writing processes left. Lots of processes can successfully read the file, since they do not block each other. This means that a process that wants to write to the file may never get a chance to squeeze in, since it needs to obtain an exclusive lock.

Figure 19-2 represents a possible scenario in which everybody can read but no one can write. ("pX" represents different processes running at different times, all acquiring shared locks on the DBM file.)

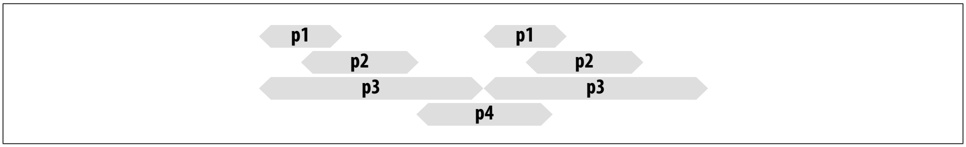

Figure 19-2. Overlapping shared locks prevent an exclusive lock

The result is a starving process that will time out the request, which will fail to update the DB. Ken Williams solved this problem with his `Tie::DB_Lock` module, discussed later in this chapter.

There are several locking wrappers for `DB_File` on CPAN right now. Each one implements locking differently and has different goals in mind. It is worth knowing the differences between them, so that you can pick the right one for your application.

Flawed Locking Methods

The suggested locking methods in the first and second editions of the book *Programming Perl* (O'Reilly) and the `DB_File` manpage (before Version 1.72, fixed in 1.73) are flawed. If you use them in an environment where more than one process can modify the DBM file, it can be corrupted. The following is an explanation of why this happens.

You cannot use a tied file's file handle for locking, since you get the file handle after the file has already been tied. It's too late to lock. The problem is that the database file is locked *after* it is opened. When the database is opened, the first 4 KB (for the Berkeley DB library, at least) are read and then cached in memory. Therefore, a process can open the database file, cache the first 4 KB, and then block while another process writes to the file. If the second process modifies the first 4 KB of the file, when the original process gets the lock it now has an inconsistent view of the database. If it writes using this view it may easily corrupt the database on disk.

This problem can be difficult to trace because it does not cause corruption every time a process has to wait for a lock. One can do quite a bit of writing to a database file without actually changing the first 4 KB. But once you suspect this problem, you can easily reproduce it by making your program modify the records in the first 4 KB of the DBM file.

It's better to resort to using the standard modules for locking than to try to invent your own.

If your DBM file is used only in the read-only mode, generally there is no need for locking at all. If you access the DBM file in read/write mode, the safest method is to tie the DBM file after acquiring an external lock and untie it before the lock is released. So to access the file in shared mode (`FLOCK_SH`[*]), follow this pseudocode:

```
flock $fh, FLOCK_SH <===== start critical section
tie...
read...
untie...
flock $fh, FLOCK_UN <===== end critical section
```

[*] The `FLOCK_*` constants are defined in the `Fcntl` module; `FLOCK_SH` for shared, `FLOCK_EX` for exclusive, and `FLOCK_UN` for unlock.

Similarly for the exclusive (EX) write access:

```
flock FLOCK_EX <===== start critical section
tie...
write...
sync...
untie...
flock FLOCK_UN <===== end critical section
```

You might want to save a few tie()/untie() calls if the same request accesses the DBM file more than once. Be careful, though. Based on the caching effect explained above, a process can perform an atomic downgrade of an exclusive lock to a shared one without retying the file:

```
flock FLOCK_EX <===== start critical section
tie...
write...
sync...
               <===== end critical section
flock FLOCK_SH <===== start critical section
read...
untie...
flock FLOCK_UN <===== end critical section
```

because it has the updated data in its cache. By atomic, we mean it's ensured that the lock status gets changed without any other process getting exclusive access in between.

If you can ensure that one process safely upgrades a shared lock to an exclusive lock, you can save the overhead of doing the extra tie() and untie(). But this operation might lead to a deadlock if two processes try to upgrade from shared to exclusive locks at the same time. Remember that in order to acquire an exclusive lock, all other processes need to release *all* locks. If your OS's locking implementation resolves this deadlock by denying one of the upgrade requests, make sure your program handles that appropriately. The process that was denied has to untie the DBM file and then ask for an exclusive lock.

A DBM file always has to be untied before the lock is released (unless you do an atomic downgrade from exclusive to shared, as we have just explained). Remember that if at any given moment a process wants to lock and access the DBM file, it has to retie this file if it was tied already. If this is not done, the integrity of the DBM file is not ensured.

To conclude, the safest method of reading from a DBM file is to lock the file before tying it, untie it before releasing the lock, and, in the case of writing, call sync() before untying it.

Locking Wrappers Overview

Here are the pros and cons of the DBM file-locking wrappers available from CPAN:

`Tie::DB_Lock`
: A `DB_File` wrapper that creates copies of the DBM file for read access, so that you have a kind of multiversioning concurrent read system. However, updates are still serial. After each update, the read-only copies of the DBM file are recreated. Use this wrapper in situations where reads may be very lengthy and therefore the write starvation problem may occur. On the other hand, if you have big DBM files, it may create a big load on the system if the updates are quite frequent. This module is discussed in the next section.

`Tie::DB_FileLock`
: A `DB_File` wrapper that has the ability to lock and unlock the database while it is being used. Avoids the tie-before-flock problem by simply retying the database when you get or drop a lock. Because of the flexibility in dropping and reacquiring the lock in the middle of a session, this can be used in a system that will work with long updates and/or reads. Refer to the `Tie::DB_FileLock` manpage for more information.

`DB_File::Lock`
: An extremely lightweight `DB_File` wrapper that simply flocks an external lock file before tying the database and drops the lock after untying. This allows you to use the same lock file for multiple databases to avoid deadlock problems, if desired. Use this for databases where updates and reads are quick, and simple `flock()` locking semantics are enough. Refer to the `DB_File::Lock` manpage for more information.

On some operating systems (FreeBSD, for example), it is possible to lock on `tie`:

```
tie my %t, 'DB_File', $DBM_FILE, O_RDWR | O_EXLOCK, 0664;
```

and release the lock only by untying the file. Check if the `O_EXLOCK` flag is available on your operating system before you try to use this method!

Tie::DB_Lock

`Tie::DB_Lock` ties hashes to databases using shared and exclusive locks. This module, written by Ken Williams, solves the problems discussed earlier.

The main difference with this module is that `Tie::DB_Lock` copies a DBM file on read. Reading processes do not have to keep the file locked while they read it, and writing processes can still access the file while others are reading. This works best when you have lots of long-duration reading processes and a few short bursts of writing.

The drawback of this module is the heavy I/O performed when every reader makes a fresh copy of the DB. With big DBM files this can be quite a disadvantage and can slow down the server considerably.

An alternative would be to have one copy of the DBM image shared by all the reading processes. This would cut the number of files that are copied and put the responsibility of copying the read-only file on the writer, not the reader. However, some care would be required to make sure that readers are not disturbed when a new read-only copy is put into place.

Examples

Let's look at a few examples that will demonstrate the theory presented at the beginning of the chapter.

tie( )-ing Once and Forever

If you know that your code accesses the DBM file in read-only mode and you want to gain the maximum data-retrieval speed, you should tie the DBM file during server startup and register code in the child initialization stage that will tie the DBM file when the child process is spawned.

Consider the small test module in Example 19-2.

Example 19-2. Book/DBMCache.pm

```
package Book::DBMCache;

use DB_File;
use Fcntl qw(O_RDONLY O_CREAT);

use vars qw(%dbm);

sub init {
    my $filename = shift;
    tie %dbm, 'DB_File', $filename, O_RDONLY|O_CREAT,
        0660, $DB_BTREE or die "Can't tie $filename: $!";
}
1;
```

This module imports two symbols from the `Fcntl` package that we will use to tie the DBM file. The first one is `O_RDONLY`, as we want the file to be opened only for reading. It is important to note that in the case of the `tie( )` interface, nothing prevents you from updating the DBM file, even if the file was tied with the `O_RDONLY` flag. The second flag, `O_CREAT`, is used just in case the DBM file wasn't found where it was expected—in this case, an empty file will be created instead, since otherwise `tie( )` will fail and the code execution will be aborted.

The module specifies a global variable, `%dbm`, which we need to be global so that we can access it directly from outside of the `Book::DBMCache` module. Alternatively, we could define this variable as lexically scoped to this module and write an accessor (method), which would make the code cleaner. However, this accessor would be called every time we wanted to read some value.

When `Book::DBMCache::init( )` is called with a path to the DBM file as its argument, the global variable `%dbm` is tied to this file. We want the `tie` operation to happen before the first request is made, so we do it in the `ChildInitHandler` code executed from *startup.pl*:

```
use Book::DBMCache;
Apache->push_handlers(PerlChildInitHandler => sub {
                        Book::DBMCache::init("/tmp/foo.db");
                    });
```

Assuming */tmp/foo.db* is already populated with data, we can now write the test script shown in Example 19-3.

Example 19-3. test_dbm.pl

```
use Book::DBMCache;
use strict;

my $r = shift;
$r->send_http_header("text/plain");

my $foo = exists $Book::DBMCache::dbm{foo} ? $Book::DBMCache::dbm{foo} : '';
print "The value of foo: [$foo]";
```

When this is executed as an `Apache::Registry` script (assuming the DBM file was populated with the `foo`, `bar` key/value pair), we will see the following output:

```
The value of foo: [bar]
```

There's an easy way to guarantee that a tied hash is read-only: use a subclass of the tie module you're using that prevents writing. For example, you can subclass `DB_File` as follows:

```
package DB_File::ReadOnly;

use strict;
require DB_File;
$DB_File::ReadOnly::ISA = qw(DB_File);

sub STORE  { }
sub DELETE { }
sub CLEAR  { }

1;
```

As you can see, the methods of the tie() interface that can alter the DBM file are overriden with methods that do nothing. Of course, you may want to use warn() or die() inside these methods, depending on how you want to flag writes. Any attempts to write probably should be considered serious problems.

Now you can use DB_File::ReadOnly just like you were using DB_File before, but you can be sure that the DBM file won't be modified through this interface.

Read/Write Access

This simple example will show you how to use the DBM file when you want to be able to safely modify it in addition to just reading from it. As mentioned earlier, we are running in a multiprocess environment in which more than one process might attempt to write to the file at the same time. Therefore, we need to have a lock on the DBM file before we can access it, even when doing only a read operation—we want to make sure that the retrieved data is completely valid, which might not be the case if someone is writing to the same record at the time of our read. We are going to use the DB_File::Lock module from CPAN to perform the actual locking.

The simple script shown in Example 19-4 imports the O_RDWR and O_CREAT symbols from the Fcntl module, loads the DB_File::Lock module, and sends the HTTP header as usual.

Example 19-4. read_write_lock.pl

```
use strict;
use DB_File::Lock;
use Fcntl qw(O_RDWR O_CREAT);

my $r = shift;
$r->send_http_header("text/plain");

my $dbfile = "/tmp/foo.db";
tie my %dbm, 'DB_File::Lock', $dbfile, O_RDWR|O_CREAT,
    0600, $DB_HASH, 'write';
# assign a random value
$dbm{foo} = ('a'..'z')[int rand(26)];
untie %dbm;

# read the assigned value
tie %dbm, 'DB_File::Lock', $dbfile, O_RDWR|O_CREAT,
    0600, $DB_HASH, 'read';
my $foo = exists $dbm{foo} ? $dbm{foo} : 'undefined';
untie %dbm;

print "The value of foo: [$foo]";
```

The next step is to tie the existing */tmp/foo.db* file, or create a new one if it doesn't already exist. Notice that the last argument for the tie is 'write', which tells DB_File::Lock to obtain an exclusive (write) lock before moving on. Once the exclusive lock is acquired and the DBM file is tied, the code assigns a random letter as a value and saves the change by calling `untie()`, which unlocks the DBM and closes it. It's important to stress here that in our example the section of code between the calls to `tie()` and `untie()` is called a critical section, because while we are inside of it, no other process can read from or write to the DBM file. Therefore, it's important to keep it the execution time of this section as short as possible.

The next section is similar to the first one, but this time we ask for a shared (read) lock, as we only want to read the value from the DBM file. Once the value is read, it's printed. Since the letter was picked randomly, you will see something like this:

```
The value of foo: [d]
```

then this (when reloading again):

```
The value of foo: [z]
```

and so on.

Based on this example you can build more evolved code, and of course you may choose to use other locking wrapper modules, as discussed earlier.

Storing Complex Data Structures

As mentioned earlier, you can use the `MLDBM` module to store complex data structures in the DBM file (which apparently accepts only a scalar as a single value). Example 19-5 shows how to do this.

Example 19-5. mldbm.pl

```
use strict;
use MLDBM qw(DB_File);
use DB_File;
use Data::Dumper ();
use Fcntl qw(O_RDWR O_CREAT);

my $r = shift;
$r->send_http_header("text/plain");

my $rh = {
          bar => ['a'..'c'],
          tar => { map {$_ => $_**2 } 1..4 },
         };

my $dbfile = "/tmp/foo.db";
tie my %dbm, 'MLDBM', $dbfile, O_RDWR|O_CREAT,
    0600, $DB_HASH or die $!;
# assign a reference to a Perl datastructure
$dbm{foo} = $rh;
untie %dbm;
```

Example 19-5. mldbm.pl (continued)

```
# read the assigned value
tie %dbm, 'MLDBM', $dbfile, O_RDWR|O_CREAT,
    0600, $DB_HASH or die $!;
my $foo = exists $dbm{foo} ? $dbm{foo} : 'undefined';
untie %dbm;

print Data::Dumper::Dumper($foo);
```

As you can see, this example is very similar to the normal use of `DB_File`; we just use `MLDBM` instead, and tell it to use `DB_File` as an underlying DBM implementation. You can choose any other available implementation instead. If you don't specify one, `SDBM_File` is used.

The script creates a complicated nested data structure and stores it in the `$rh` scalar. Then we open the database and store this value as usual.

When we want to retrieve the stored value, we do pretty much the same thing as before. The script uses the `Data::Dumper::Dumper` method to print out the nested data structure. Here is what it prints:

```
$VAR1 = {
          'bar' => [
                     'a',
                     'b',
                     'c'
                   ],
          'tar' => {
                     '1' => '1',
                     '2' => '4',
                     '3' => '9',
                     '4' => '16'
                   }
        };
```

That's exactly what we inserted into the DBM file.

There is one important note, though. If you want to modify a value that is a reference to a data structure, you cannot modify it directly. You have to retrieve the value, modify it, and store it back.

For example, in the above example you cannot do:

```
tie my %dbm, 'MLDBM', $dbfile, O_RDWR|O_CREAT,
    0600, $DB_HASH or die $!;
# update the existing key
$dbm{foo}->{bar} = ['a'..'z']; # this doesn't work
untie %dbm;
```

if the key `bar` existed before. Instead, you should do the following:

```
tie my %dbm, 'MLDBM', $dbfile, O_RDWR|O_CREAT,
    0600, $DB_HASH or die $!;
# update the existing key
```

```
my $tmp     = $dbm{foo};
$tmp->{bar} = ['a'..'z'];
$dbm{foo}   = $tmp;       # this works
untie %dbm;
```

This limitation exists because the perl TIEHASH interface currently has no support for multidimensional ties.

By default, MLDBM uses Data::Dumper to serialize the nested data structures. You may want to use the FreezeThaw or Storable serializer instead. In fact, Storable is the preferred one. To use Storable in our example, you should do:

```
use MLDBM qw(DB_File Storable);
```

at the beginning of the script.

Refer to the MLDBM manpage to find out more information about it.

References

- Chapter 14 in *Perl Cookbook*, by Tom Christiansen and Nathan Torkington (O'Reilly)
- Chapter 17 in *Learning Perl*, Second Edition, by Randal L. Schwartz and Tom Christiansen (O'Reilly)
- Chapter 2 in Prog*ramming the Perl DBI*, by Alligator Descartes and Tim Bunce (O'Reilly)
- The Berkeley DB web site: *http://www.sleepycat.com/*