



Using memcached

How to scale your website easily

Josef Finsel

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas

Useful Friday Links

- [Source code](#) from this book and other resources.
- [Free updates to this PDF](#)
- [Errata and suggestions](#). To report an erratum on a page, click the link in the footer.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

To see what we're up to, please visit us at

<http://www.pragmaticprogrammer.com>

Copyright © 2008 Josef Finsel.

All rights reserved.

This PDF publication is intended for the personal use of the individual whose name appears at the bottom of each page. This publication may not be disseminated to others by any means without the prior consent of the publisher. In particular, the publication must not be made available on the Internet (via a web server, file sharing network, or any other means).

Produced in the United States of America.

Lovingly created by gerbil #19 on 2009-4-20



Contents

1	Introduction	1
1.1	What is memcached?	2
1.2	What memcached Isn't	3
1.3	What Components Make Up memcached?	3
1.4	How do I Install the memcached Server?	4
1.5	How Do I Configure memcached?	7
1.6	How Do I Manipulate Data?	8
1.7	What Options Do I Have for Storing Data?	13
1.8	What Other Commands Can I Use?	15
1.9	Review	17
2	Using a memcached Client Library	18
2.1	How Do I Install a Windows Client?	19
2.2	Where do I get the memcached Linux client?	22
2.3	What are the benefits of Using a Client?	25
2.4	Review	29
3	The Basics of Implementing memcached	30
3.1	How Does memcached Fit in the Cache System?	30
3.2	What is the Basic Coding Pattern for Using memcached?	31
3.3	How Do I Update memcached When the Data Changes?	39
3.4	How Do I Prevent Multiple Clients Updating One NVP?	42
3.5	How Do I Determine the Optimal Expiration Time?	46
3.6	Can I Eliminate Caching Duplicate Data?	51
3.7	How do I Gauge Cache Efficiency?	56
3.8	Review	57

4	Best Practices	58
4.1	How Can I Secure memcached?	58
4.2	How Do I Determine What Gets Cached?	59
4.3	How Do I Name Keys?	60
4.4	Which Storage Command is Best?	60
4.5	How Do I Fill the Cache?	62
4.6	How Can I Minimize memcached Server Outages? .	63
4.7	What's the Future of memcached?	64
5	memcached Add Ons	66
6	Additional Resources	68
6.1	memcached Resources	68

Chapter 1

Introduction

memcached

What we're working on lately (this past week) is a hard-core distributed memory caching system. Basically, we're putting up a bunch of servers that do nothing but keep frequently-used LiveJournal objects in memory. Objects can be users, logins, colors, styles, journal entries, comments, site text, anything...

It was with these words in a post to the lj_maintenance community that *memcached* was first introduced into the general world. *memcached*, which stands for Memory Cache Daemon, was ready for its first big test. LiveJournal, one of the first large social blogging sites, had been going through a period of rapid growth and in the spring of 2003 users were complaining about things being very slow. In a post that outlined LiveJournal's infrastructure, Brad Fitzgerald, founder of LiveJournal and developer of *memcached*, pointed out that one of the big bottlenecks was the reading from the database. And so the creative developers of LiveJournal started work on *memcached*. When it was fully implemented in October, the statistics were phenomenal: almost 95% of the data reads were served up from *memcached* and the largest bottleneck in the system had been eliminated.

Today *memcached* is in use by many major sites; including Flickr, Slashdot, Wikipedia and Facebook, just to name a few. But what is it and how can a website benefit from it? We're going to explore those questions in these pages.

1.1 What is memcached?

memcached is a server that caches Name Value Pairs (NVPs) in memory. The value in the NVP can be anything that fits in memcached: rows of data, *HTML* fragments, binary objects. Retrieving the cached value from memory is more efficient than having to get it from disk, so applications implementing memcached are more scalable. For a demonstration of why this is so, let's take a look at a calendar of events web page and how implementing memcached can improve efficiency.

Without memcached, every time the web server gets a request for a list of upcoming events, it queries the database server for information. The database server retrieves the data from the disk and hands it back to the web server to format and finally send back to the web browser for display. With memcached, the web page checks memcached first and returns the data from there if it exists. If it doesn't, the web server queries the database and then stores the results in memcached so they are there for the next request. This adds a bit of extra overhead but the difference between the time it takes to read from memory and the time it takes to read from disk more than makes up for it, allowing the web server to deliver more pages than if it had to query the database every time.

cache: a temporary storage of values for more efficient retrieval than retrieving the value from its original location.

Caches should never be used as persistent data stores.

In its most basic form, that is how memcached is used and implementing it is almost that easy. But this simplicity is frequently misunderstood by people when they first start thinking about memcached and how they can use it in their site. So, before we get into the technical details, best practices, and examples of implementing memcached, let's take a quick look at what memcached *isn't*.

1.2 What memcached Isn't

First, memcached is not a persistent data store. You cannot query memcached and get a list of all the values it holds, nor can you dump all of the values in memcached to disk. The only way to know if something is in memcached is to query the server and find out. This is by design since memcached was optimized to be a caching server, not a persistent data storage server.

Second, there is no security mechanism built into memcached, but we will explore ways to secure the caches through other means in Section 4.1, *How Can I Secure memcached?*, on page 58

The final point to make is that memcached does not support any fail-over/high-availability mechanisms. If a memcached server goes down, all of that data is gone. But that's ok because memcached is a cache, not the original source of the data. The code will simply fail to find the data in memcached and get it out of the database. There are ways to minimize the problem if a memcached server goes down and we'll cover those in Section 4.6, *How Can I Minimize memcached Server Outages?*, on page 63.

1.3 What Components Make Up memcached?

memcached is made up of two components: the server and the client. In this chapter we'll focus on the server and in the next we'll focus on the client. The memcached server is best thought of as a *Name-Value Pair* (NVP) server, storing values by a lookup key (name) in memory. That's all the server does: store and retrieve data stored with a key. It is a very simple, very fast program with two limitations: the size of the key cannot exceed 250 characters and the size of any

You may be tempted to find some way to use memcached for something other than a cache. I recently found myself in this very situation. We designed a project that used memcached as a cheap way to handle expiration of logins. Whenever a login needed to be validated we would check memcached, see if the login had expired and update the current value to reflect the last time the login was validated. It was quick, easy, and not the way to use a memory cache, as I found out the first time we flushed the cache and everyone was forced to log back in because their session authentication had been lost! I had been treating memcached as a high-availability data store rather than a caching system. So we rewrote the application to use a rolling update to the database like we implement in Section 3.4, *How Do I Prevent Multiple Clients Updating One NVP?*, on page 42, checking memcached first.

The Moral of the Story: Any time you find yourself trying to use memcached as a data store, you probably should rethink your design.

Name-Value Pair

chunk of data you can store is 1 MB. Also, each memcached server is atomic. It neither knows nor cares about any other memcached server; knowing which server contains the NVP is the responsibility of the client. So you can add as many memcached servers as you'd like.

1.4 How do I Install the memcached Server?

The memcached server can be installed on either Windows or Linux and this section will show how to do both. While the steps to install memcached on both platforms are specific, once it's installed, there is no reason that you cannot use both Windows and Linux servers in a shared pool of memcached servers.

How Do I Install memcached on a Linux Server?

The best way to install the memcached server on a Linux distribution is to download and compile the source code. The following instructions should help you install and configure the server. You will need to have a developer box with gcc installed and you'll need root privileges to properly compile and install memcached.

There is a supported install package for Debian that can be retrieved with `apt-get install memcached`. The current packaged release is 1.1.12, which is quite a bit older than the current 1.2.5 version.

Dependency: Getting libevent

memcached uses the libevent API. libevent provides a mechanism to execute a callback function when a specific event occurs. A copy of it may already be installed on your computer but you will need the 1.3 version. The following steps should download the 1.3 version of libevent and create it:

[Download](#) Introduction/LibEventInstall.txt

```
cd /usr/local/src
wget http://monkey.org/~provos/libevent-1.3b.tar.gz
tar zxvf libevent-1.3b.tar.gz
cd libevent-1.3b
./configure
make && make install
```

Now we need to update `/etc/ld.so.conf.d/libevent-i386.conf` to add the path information for libevent. Use your favorite editor to edit `/etc/ld.so.conf.d/libevent-i386.conf` and add the following line if it doesn't exist: `/usr/local/lib/`

The last step is to run `ldconfig`. Now we're ready to get and build memcached.

Getting memcached

Now that we have libevent created, we can download and build memcached. You can check the memcached distribution page (see Resources) to find the latest version. The latest version currently available at the time of this writing is 1.2.5. So go ahead and run the following steps:

[Download](#) Introduction/memcachedInstall.txt

```
cd /usr/local/src
wget http://danga.com/memcached/dist/memcached-1.2.5.tar.gz
tar zxvf memcached-1.2.5.tar.gz
cd memcached-1.2.5
./configure
make && make install
```

Now you should have a working copy of the latest memcached server. The install should provide you with a memcached shell script in

/etc/init.d. You can modify this script to run memcached using the runtime options so it will automatically start whenever your server reboots.

Installing memcached Server on Windows

Installing memcached on Windows is as easy as downloading the binaries from the link in the Resources section. You can also download and compile the source, but either way, the end result is memcached.exe. When you run `memcached -d install`, it will install the program as a service. You can start and stop the service by running `memcached -d` followed by `start`, `stop`, `shutdown` or `restart`; or from Services in the Administrative Tools.

To modify any of the parameters for memcached you will need to use `regedit`. Drill down to `HKEY_LOCAL_MACHINE\Software\System\Services\memcached` and modify the `ImagePath` entry (see Figure 1.1, on the next page).

Running the server under Windows as a service only allows one instance. If you want to run multiple instances you will need to actually run `memcached` multiple times with different port addresses. This can be done by adding keys to the Registry or by using a tool such as `AutoRuns`¹ to do that for you. We'll look at the how to specify a different port in Section 1.5, *How Do I Configure memcached?*, on the following page.

Making a compiled version of memcached under windows is best left to people with lots of C++ experience. But, if you want to compile a version, you'll need to download a copy of the source code from <http://code.sixapart.com/svn/memcached/branches/memcached-win32> using a SubVersion tool like TortoiseSVN.

You will also need a copy of `libevent`. Links for that can be found in the resources section. The details of compiling the Windows version are more complex than those of the Linux version in the Linux installation in the following section but, if you regularly use C++ you should be able to follow the directions given in that section to build a copy here.

¹<http://www.microsoft.com/technet/sysinternals/Security/Autoruns.msp>

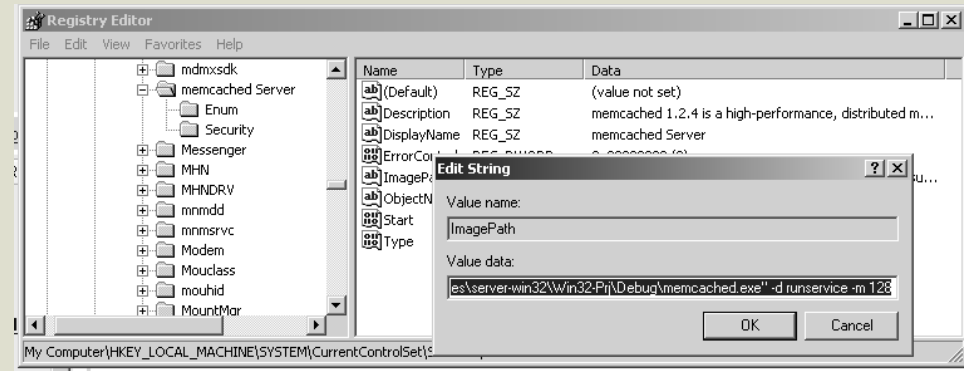


Figure 1.1: Use Regedit to change the service's parameters in Windows

1.5 How Do I Configure memcached?

Both the Windows and Linux versions of the memcached server use command line arguments to customize the server instance. Three key arguments are `-p`, `-m` and `-d`. By default, memcached listens in on port 11211. You can change the port it listens on with `-p` followed by the port number. The amount of memory memcached will use is set with `-m`, defaulting to 64MB. Finally, `-d` will run memcached as a daemon. If you have a multi-processor machine or a lot of memory, you may want to set up multiple instances of memcached running with large chunks of memory.

You can see a complete listing of the arguments available by running `memcached -h` (see Figure 1.2, on the next page for an example). For more information on how to use other arguments, see the memcached website (<http://www.danga.com/memcached/>).

```
jfinzel@deb01:/usr/local/bin$ memcached -h
memcached 1.2.2
-p <num>      TCP port number to listen on (default: 11211)
-U <num>      UDP port number to listen on (default: 0, off)
-s <file>      unix socket path to listen on (disables network support)
-l <ip_addr>   interface to listen on, default is INADDR_ANY
-d           run as a daemon
-r           maximize core file limit
-u <username> assume identity of <username> (only when run as root)
-m <num>      max memory to use for items in megabytes, default is 64 MB
-M           return error on memory exhausted (rather than removing items)
-c <num>      max simultaneous connections, default is 1024
-k           lock down all paged memory
-v           verbose (print errors/warnings while in event loop)
-vv          very verbose (also print client commands/reponses)
-h           print this help and exit
-i           print memcached and libevent license
-b           run a managed instance (mnemonic: buckets)
-P <file>     save PID in <file>, only used with -d option
-f <factor>   chunk size growth factor, default 1.25
-n <bytes>    minimum space allocated for key+value+flags, default 48
```

Figure 1.2: memcached options

1.6 How Do I Manipulate Data?

memcached only has four basic commands related to storing and retrieving data; in this section we are going to explore each of them to see how they work. These commands are outlined in the Protocol document.² Interaction with the server normally takes place over a TCP or UDP connection. We can actually interact with a server using telnet, which is exactly what we're going to do now as we explore the basic commands available to us. Let's look at using telnet to store and retrieve data using these four simple commands. When you attempt to put something into the cache, you'll get either a confirmation that it was STORED or told that it was NOT_STORED.

²<http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt?rev=HEAD>

Connecting to a memcached server using telnet is as simple as executing `telnet servername port`. So, if you are running memcached on your machine and it's using the default port you could use `telnet 127.0.0.1 11211`.

SET: add a new item to memcached or replace an existing one with new data

```
⇒ set test1 0 0 10
testing001
STORED
```

ADD: only store the data if the key doesn't exist. If the key exists, we get NOT_STORED, as the test1 shows below. Otherwise we get STORED.

```
⇒ add test1 0 0 10
testing002
NOT_STORED
add test2 0 0 10
testing002
STORED
```

REPLACE: only store the data if the key already exists. If the key does not exist, we get NOT_STORED, as the test3 shows below. Otherwise we get STORED.

```
⇒ replace test1 0 0 10
testing003
STORED
replace test3 0 0 10
testing003
NOT_STORED
```

GET: return the data. When you get data out of the cache it tells you the name of the key, the value of the flag that you can pass in and the number of bytes on one line, the actual data on the next line and finally returns END on a line of its own. If the key doesn't exist, it returns END on the first line.

```
⇒ get test1
VALUE test1 0 10
testing003
END
```

```
get test4
END
get test1 test2
VALUE test1 0 10
testing003
END
```

The first command, using `set`, stored the value `testing001` under the key `test1`. That's because `set` either creates a new NVP if the key doesn't exist or replaces the value if the key exists. Then, when we try to use `add` with the key `test1`, the server returns `NOT_STORED` because `add` requires that the key not exist. Finally, we use `replace` to store data to the key `test1` and the key `test2`. Since the key `test1` exists, the server stores the data but it doesn't store anything for the key `test2` because the key doesn't exist and `replace` requires it to exist.

When we use the `get` command, `memcached` returns two lines of data. First is the line that starts with `VALUE`, to indicate that it is returning data. This is followed by the name of the key that was requested, the flag value (that we will cover in Section 1.7, *What Options Do I Have for Storing Data?*, on page 13) and the number of bytes `memcached` will be returning. Next comes the requested data followed by `END` on a line by itself. If there is no data then all that gets returned is `END` on a line by itself, as you see when we try to get the value for the key `test2`. Finally, you can include multiple keys in a request by separating them with spaces. When you ask for multiple keys, you only get confirmation of the keys that have data. If an NVP exists on the server, it will return the name and value. `memcached` won't, however, return an empty name for NVPs it doesn't contain. If you ask for multiple keys and there is no data, all you will get is the `END`.

More Advanced Data Manipulation

There are two additional, related data command: `gets` and `cas`, which stands for Check And Set. `gets` has the same command format as `get` but returns an extra bit of data, a 64 bit integer that uniquely identifies this data. As you can see in the following example, the first `gets` returns a CAS value of 5. Then, when the NVP is updated using `set`, the next `gets` returns a different value. Where this is useful is in using the CAS value along with `cas`. The first attempt to save the data, using the original CAS value of 5 returns an EXISTS message, indicating we didn't successfully store the data because the CAS value was different then the one that currently exists. When we used the correct CAS value with the `cas`, we are told that the data was successfully STORED.

```
⇒ gets test1
VALUE test1 0 10 5
testtest01
END
set test1 0 0 10
test01test
STORED
gets test1
VALUE test1 0 10 6
test01test
END
cas test1 0 0 10 5
test02test
EXISTS
cas test1 0 0 10 6
test03test
STORED
get test1
VALUE test1 0 10
test03test
```

END

Removing Data From memcached

One way to remove data from memcached is to delete it. The format of the delete command is simple: delete key You can type the commands below in telnet while connected to memcached for a demonstration.

```
set test1 0 0 10
testing001
get test1
delete test1
get test1
```

When an object expires or is removed using the delete command, all memcached really does is invalidate the key. In order to make memcached as fast as possible, items are not deleted from the memory until the memory they occupy is needed. This is much more efficient than trundling through all of the items in the cache to physically delete them all the time.

Items are also deleted from the cache when memcached needs memory and determines that an item hasn't been used in a while and probably serves no purpose remaining in cache. When memcached needs memory and there are no expired or deleted areas that can be reused, it uses a *Least Recently Used (LRU)* algorithm to remove items that are the oldest things in memory that it determines are no longer worth caching.

Least Recently Used (LRU)

1.7 What Options Do I Have for Storing Data?

Now let's take a moment and look at the parameters of the storage commands, which we've used so far in this chapter. `set`, `add`, `replace` and `cas` all have four required parameters, outlined in the table below.

keyname	the name of the name value pair
flags	an integer that is passed through with the key and is transparent to memcached. This is used by the client if it wants to store some information about the key and generally access to this parameter is not exposed through the client.
expiration time	0 means never delete, 1-2,592,000 ³ is the number of seconds to keep, > 2,592,000 equals the Unix time.
bytes	The number of bytes to be stored.

So far, we've been passing in 0 for the flag and the expiration time, but let's set a key that will expire by passing in an expiration value. We will set something to expire in 10 seconds by passing in 10 as the second option. An example can be found in the sample code below. After storing the data you can do `get test1` and it will return data until the 10 seconds have passed.

```
set test1 0 10 10
testing001
```

Interestingly enough, `cas` supports `noreply`. Which seems to defeat the purpose of using the command in the first place.

In addition to these required options, there is an optional one: `noreply`. This option suppresses the `STORED` or `NOT_STORED` indicator and is designed for folks who are really pushing memcached and clients to

³Number of seconds in 30 days: 60*60*24*30

the limit because it means the client doesn't need to wait around for an answer and the server doesn't need to use cycles to return one. This makes sense if you think about the original design behind memcached, as a memory based cache. One of two things can happen when you request data from the cache: it will be there or it won't. It may not be there because it expired out of the cache, it may not be there because memcached needed the memory location that was being used, it may not be there because it was deleted on purpose, or it may not be there because it was never put there. Any program using memcached doesn't really care *why* the data isn't there, only that it has to look somewhere else to get the data. And if that's the case, why does the program need to know that the data was successfully stored? So no return code is provided.

Increment/Decrement

memcached has two handy commands for using numeric counters: INCR (increment) and DECR (decrement). These two commands add or subtract values from an *NVP* in memcached so you can use a value as a simple counter.

Let's look at an example of this. On one site that I've implemented memcached on, we're storing several different types of data in the cache. When we set a debug flag to true, it starts using INCR on two keys for each type of data, one for cache hits and one for cache misses. That enables us to query the cache periodically to see how specific types of data are being cached and to help us determine where we might have issues related to what we have defined as cacheable.

The one downside to INCR and DECR is that they require the *NVP* to exist before they can be used. When you attempt to increment

If you flush the cache so the system is empty and store a 1 byte value with a 1 byte key, how many bytes will that take? Well, that depends. If you try this on a default memcached server, odds are that you will find that one key takes 45 bytes. One of the parameters for memcached (see Figure 1.2, on page 8) is `-n`, which determines the minimum number of bytes that a key, flag and value will take up. Unless you *know* that you're going to be storing lots of key/flag/values that will total less than the default, it doesn't matter. As you can see from Figure 1.3, on the following page, the average size of items stored in this cache is about 160 bytes (`bytes / curr_items`).

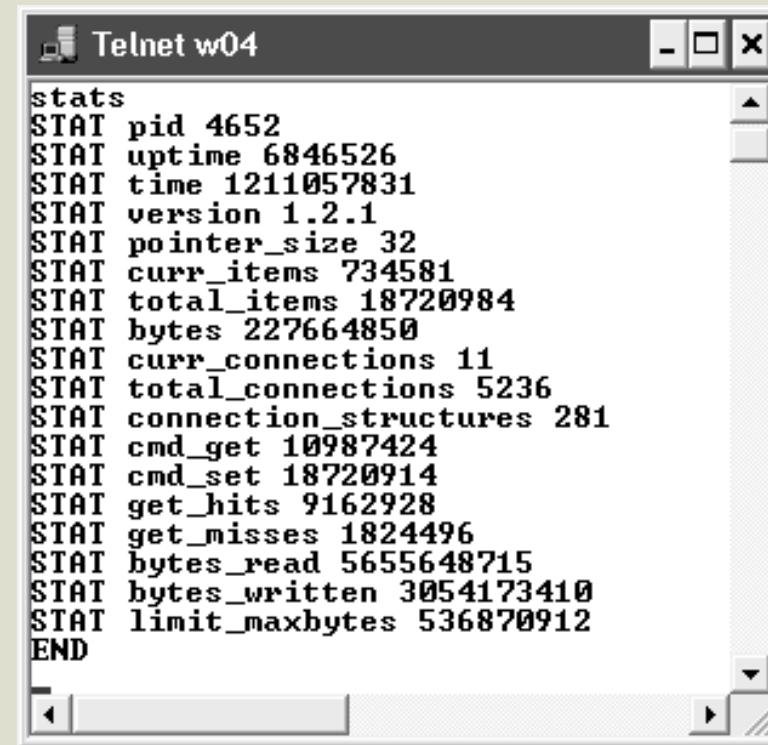
a value that doesn't exist, it will return `ERROR`. The format of the command is `INCR/DECR key value`.⁴

1.8 What Other Commands Can I Use?

In addition to the commands to manipulate data in the cache, there are two commands for helping to manage memcached. The simplest is `flusha_all`. This will reset the cache; it's handy for testing and debugging. `stats`, on the other hand, is a useful tool for getting information about the memcached server and is primarily used to determine your cache's effectiveness. Let's take a look at the output of `stats` in Figure 1.3, on the next page.

Most of the values are defined in the `Protocol.txt` file (Section 1.6, *How Do I Manipulate Data?*, on page 8) but different implementations on different machines may have additional statistics returned, so I'm not going to go through each of them in detail. Instead, I'm going to look at some of the key statistics and how they can help determine utilization. The first set of statistics includes `curr_items`, `total_items`, and `bytes`. These tell something about what's in the cache. `total_items` defines the total number of items in the cache, including active, expired, and deleted items. Deleted items are only those marked as deleted. When an item is actually physically deleted so that the memory can be reused, the `total_items` value is decremented. `curr_items` is the number of active items in the cache. `bytes` is the total number of bytes being used by active items.

⁴The latest version of memcached, 1.2.4, allows `INCR` and `DECR` to use a 64 bit counter rather than the 32 bit counter used in earlier versions.



```
stats
STAT pid 4652
STAT uptime 6846526
STAT time 1211057831
STAT version 1.2.1
STAT pointer_size 32
STAT curr_items 734581
STAT total_items 18720984
STAT bytes 227664850
STAT curr_connections 11
STAT total_connections 5236
STAT connection_structures 281
STAT cmd_get 10987424
STAT cmd_set 18720914
STAT get_hits 9162928
STAT get_misses 1824496
STAT bytes_read 5655648715
STAT bytes_written 3054173410
STAT limit_maxbytes 536870912
END
```

Figure 1.3: Sample Stats

cache hit ratio

The one statistic that everyone wants to know about their memcached usage is their *cache hit ratio*, defined as how often memcached delivers something from cache compared to how often it was asked for something. This data is actually derived using `cmd_get`, `get_hits`, and `cmd_misses`. To determine the cache hit ratio take the number of `get_hits` divided by the number of `cmd_get`. According to the numbers displayed in Figure 1.3, the server has been able to deliver a little more than 83% of the requests from cache. That's a low ratio, mostly because the application using this development instance of memcached is still being optimized for caching.

Another good statistic to look at is percentage of gets to sets. For a well-tuned application, there should be more gets than sets. In this development application we can see that we are setting 70% more objects than we are getting. This indicates that a great deal of information is being placed in memcached that is never being accessed, which may mean that we need to take a closer look at what we are caching. The higher the number of gets compared to the number of sets, the better utilization is being made of memcached. And a key item to remember is that you really want to total statistics across all of your memcached servers to determine your aggregate cache hit ratio, because multiple servers for an application will be storing different sets of data. See Section 2.3, *What are the benefits of Using a Client?*, on page 25 for more information about multiple servers.

1.9 Review

We've covered the basics of what the memcached server is, an in-memory Name Value Pair *NVP* cache. We've also covered how to install it and the basic commands used to interact with it. You should be able to interact with an instance of memcached using telnet to store, retrieve and delete values. Next we'll cover clients available to programatically interact with the server, so to prepare us for Chapter 3, *The Basics of Implementing memcached*, on page 30.

Every time you reinvent the wheel, you run the risk of inventing something that doesn't roll as well.

► Josef Finsel

Chapter 2

Using a memcached Client Library

In the last chapter, we introduced the memcached server and discussed how to install it and interact with it using a telnet client. In the real world, however, using a telnet client is an extremely impractical way to interact with the memcached server. And, while it's possible to write your own code to use TCP or UDP to interact with the server, we are fortunate that much of this has already been done and there are many different client libraries ¹ that we can use to programmatically interact with the server. The current list includes:

- Perl
- PHP
- Python
- Ruby
- Java
- .NET/C#
- C
- Postgres
- Chicken

We're going to take a look at two clients, the Enyim .NET client for Windows and the Perl client for Linux. We'll talk about installing each client and look at the basics of how to connect to the server and store and retrieve data for both of the clients. We'll finish by demonstrating some of the benefits built into many of the existing

¹There is a fine distinction between the client library that contains the code that encapsulates our access to the memcached server and the actual client we write that uses that library. For the rest of the book I will use client to refer to both.

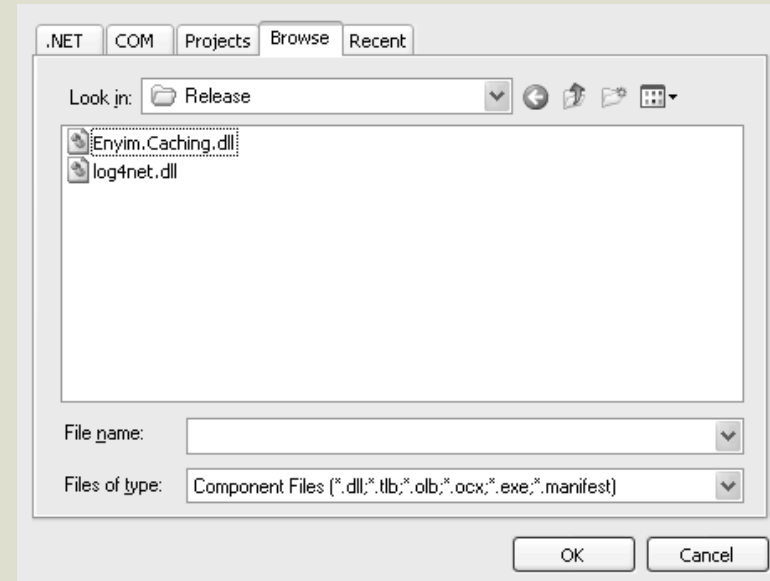


Figure 2.1: Adding memcached to a Project

memcached clients today.

There are a couple of reasons for using Perl for the sample code throughout most of the book. The original user of memcached, LiveJournal, was writing in Perl and the Perl client has been around since the beginning. So if you want to do something with memcached that falls outside the scope of this book, you'll probably find a Perl sample showing how to do it. In addition, Perl is an easy language to follow, is relatively easy to install and use, even on Windows, making the examples more universally accessible.

2.1 How Do I Install a Windows Client?

There are currently two clients available for .NET development. One is a *port* of the Java client² and the other was written for .NET from the ground up. We'll be working with that second client for the sample code within this book. You can download both a compiled DLL version as well as the source code for the Enyim memcached client

²You can download this client from <https://sourceforge.net/projects/memcacheddotnet/>

The Enyim client also uses Log4Net for logging purposes. This is a .NET version of Apache log4j tool to make logging simple. <http://logging.apache.org/log4net/>

from <http://www.codeplex.com/EnyimMemcached>. Once you've gotten the client downloaded, you're ready to start using it. The first thing to do is to create a project and add a reference to the DLL (see Figure 2.1, on the preceding page). Once you've added a reference, you need to add the configuration information to your App.Config file.

Download WindowsInstall/memcachedHashTest/memcachedHashTest/App.config

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <sectionGroup name="enyim.com">
      <section name="memcached"
        type="Enyim.Caching.Configuration.MemcachedClientSection,
        Enyim.Caching" />
    </sectionGroup>
  </configSections>
  <enyim.com>
    <memcached>
      <servers>
        <!-- put your own server(s) here-->
        <add address="127.0.0.1" port="11211" />
        <add address="127.0.0.1" port="11212" />
      </servers>
      <socketPool minPoolSize="10" maxPoolSize="100"
        connectionTimeout="00:10:00" deadTimeout="00:02:00" />
    </memcached>
  </enyim.com>
</configuration>
```

How Do I Use the Windows Client?

With configuration out of the way, we just need to start using the client. This client implements the three storage commands as enums on the first argument of the store method. The second object is any

serializable object. For this sample we're going to use strings.

[Download](#) [WindowsInstall/memcachedHashTest/memcachedHashTest/memcachedHashTest.cs](#)

```
using System;
using System.Collections.Generic;
using System.Text;
using Enyim.Caching;
using Enyim.Caching.Memcached;

namespace memcachedHashGet
{
    class memcachedHashGet
    {
        static void Main(string[] args)
        {
            MemcachedClient mc = new MemcachedClient();

            mc.FlushAll(); // Flush the cache for this example
            mc.Store(StoreMode.Set, "73EB7DE0-9452-4607-AAFC-8F7B625E75A3",
                "BEE9B83A-2A9B-4071-8D1B-90C46D2BDB30");
            mc.Store(StoreMode.Set, "D9632738-43C2-4D92-A0D3-A9E8DB2EE6E2",
                "B3879EFE-543F-4771-A243-1EA9B29E4B0A");
            mc.Store(StoreMode.Set, "74B42DC4-78E6-4649-B511-5D43F0A0E995",
                "645A9A2F-3119-4E29-92F1-5653AD3274E4");
            mc.Store(StoreMode.Set, "0D75C88B-1FDE-42A2-A7A5-17C7D0D7C03D",
                "04448CF0-DA7B-474D-B8D8-B1A3D9416793");
            mc.Store(StoreMode.Set, "210618B8-DA69-49C7-9349-16A51721913A",
                "8A580C0D-8C75-4260-A268-3D58200A6BA1");
            mc.Store(StoreMode.Set, "B31C9348-944D-491B-820A-5A1D055715C0",
                "93D53F85-236C-468B-9A0C-EE7848E95DF8");
            mc.Store(StoreMode.Set, "D6F90F35-A870-4435-BA8F-939DDB6812C6",
                "A8FBAB5E-F23C-4A01-A726-2F47324094E8");
            mc.Store(StoreMode.Set, "8682DE07-8EA2-485E-B654-B93281ACA5CA",
                "4FF12E32-B523-4DDD-AE23-B8FAACFF0C41");
            mc.Store(StoreMode.Set, "69E7D203-4483-4BC8-9912-D0F8FB639989",
                "5A01CAA6-EBE3-4628-8C60-005F06037F52");
```

```

mc.Store(StoreMode.set, "50D20618-9CE3-44E0-A0D4-B74B6486E04B",
                    "2EB046FB-D548-4B03-ACEE-A96B8FB67879");
mc.Store(StoreMode.Set, "B51AA1E9-FF4C-400B-A633-C2F2B686B417",
                    "EDF9D59B-A98F-42D6-8475-58978308E37D");
mc.Store(StoreMode.Set, "63E45F49-63C3-4BA1-81B6-36D6FD5D0E23",
                    "D2733881-3BFE-4C02-90B9-98A6C87358F9");
mc.Store(StoreMode.Set, "20F3396E-7374-4861-8E52-F81958464F50",
                    "3EBBABB1-E3EC-48B6-AC62-324A3B75031C");
mc.Store(StoreMode.Set, "A158E28A-EC6F-47EE-9C3C-E38FA0CD88BB",
                    "A8C6199C-9BF5-4B03-B274-7F7E45C94A8C");
    }
}
}

```

Similarly, getting data out of the cache is done using the `Get`, which takes a key and returns an object.

While the remaining code samples in the book are Perl samples, every Perl sample has a corresponding .NET sample.

Before we leave the Windows client section, I want to point out that if you are programming in a Windows environment but using another programming language such as Perl or PHP for Windows rather than C# or VB.NET, you can use the appropriate client for your environment. You aren't limited to using just the .NET client for Windows programming.

2.2 Where do I get the memcached Linux client?

There are a number of clients available for use on Linux systems. For this book we're going to install the Perl memcached client,³ available on *CPAN*, the Comprehensive Perl Archive Network. To

³Links to other clients are available at the memcached resources page. There's a link to that in the Resources section

IP Addresses Are Faster!

The Perl client *can* take a hostname, so I could have used "deb01:11211","laptop:11211" but that's not recommended because it takes an extra step to resolve the hostname to an IP address.

install it you will need to be logged in as a super-user and have Perl installed. Invoke the interface with the following command: perl -MCPAN -e shell.

Once you are at the CPAN shell, enter `install Cache::Memcached`. Once the CPAN interface has finished downloading and making the module you can quit and now you're ready to use memcached.

How do I use the Perl Client?

The first step to using the memcached Perl client is to tell Perl to include it with `use Cache::Memcached`. With that out of the way, we need to define a list of servers to use. This is handled by defining a new variable using a hashref of servers. Line 6 of the code below defines two servers for our Perl program to use, in the format of `Server:PortNumber`. Then we load the data.

[Download](#) `LinuxInstall/memcachedHashTest.pl`

```

Line 1  #memcachedHashTest.pl
-      #!/usr/bin/perl
-      use Cache::Memcached;
-
5      # Set the server list up
-      my $memd = new Cache::Memcached {'servers' => [ "deb01:11211",
-                                                       "192.168.0.80:11211"],};
-
-      #flush the servers to remove any unwanted data before testing.
10     $memd->flush_all();
-     $memd->set( "73EB7DE0-9452-4607-AAFC-8F7B625E75A3",
-                "BEE9B83A-2A9B-4071-8D1B-90C46D2BDB30");
-     $memd->set( "D9632738-43C2-4D92-A0D3-A9E8DB2EE6E2",
-                "B3879EFE-543F-4771-A243-1EA9B29E4B0A");
15     #disconnect from all servers
-     $memd->disconnect_all();

```

Now that we have defined the servers,⁴ we execute a flush on line 9 to clean the system before loading data. Normally we wouldn't flush the cache but, for demonstration purposes we want the cache to be empty. Now we load the data using:

```
$memd->set(Name, Value);
```

In this example I am just showing two of the twelve *NVPs* I am loading but it shows how it's done. Now that we've stored the data, let's retrieve it. There are two methods to retrieve data from memcached using the Perl client. One is using `get`. You can test to see if data was returned by checking the value retrieved:

```
$val = $memd->get("73EB7DE0-9452-4607-AAFC-8F7B625E75A3");
if ($val)
{ print $val;}
else {print "73EB7DE0-9452-4607-AAFC-8F7B625E75A3 not found!";}
```

The second way to get data using the Perl Client is to use `multi-get`. In Section 1.6, *How Do I Manipulate Data?*, on page 8 I discussed the fact that you could submit multiple keys separated by spaces after the `GET` command and memcached would return the values for the keys it had cached. The Perl client allows you to take advantage of using that multi-get functionality by passing in an array of keys to look for.

[Download](#) `LinuxInstall/memcachedHashGetMulti.pl`

```
Line 1 #memcachedHashGetMulti.pl
- #!/usr/bin/perl
- use Cache::Memcached;
-
5 # Set the server list up
- my $memd = new Cache::Memcached {'servers' => [ "127.0.0.1:11211",
-                                                "192.168.0.80:11211" ],};
```

⁴This corresponds to the `App.Config` in the .NET example.

```

- my @arrayref =("73EB7DE0-9452-4607-AAFC-8F7B625E75A3",
-               "D9632738-43C2-4D92-A0D3-A9E8DB2EE6E2",
10              "74B42DC4-78E6-4649-B511-5D43F0A0E995",
-               "0D75C88B-1FDE-42A2-A7A5-17C7D0D7C03D",
-               "210618B8-DA69-49C7-9349-16A51721913A",
-               "B31C9348-944D-491B-820A-5A1D055715C0",
-               "D6F90F35-A870-4435-BA8F-939DDB6812C6",
-               "8682DE07-8EA2-485E-B654-B93281ACA5CA",
15              "69E7D203-4483-4BC8-9912-D0F8FB639989",
-               "50D20618-9CE3-44E0-A0D4-B74B6486E04B",
-               "B51AA1E9-FF4C-400B-A633-C2F2B686B417",
-               "63E45F49-63C3-4BA1-81B6-36D6FD5D0E23",
-               "20F3396E-7374-4861-8E52-F81958464F50",
20              "A158E28A-EC6F-47EE-9C3C-E38FA0CD88BB");
- my $hashref = $memd->get_multi(@arrayref);
- #disconnect from all servers
- $memd->disconnect_all();
25 foreach $arrayref (@arrayref){
-     if($hashref->{$arrayref}){
-         print $arrayref," = ", $hashref->{$arrayref},"\\n";}
-     else {print $arrayref," DOES NOT EXIST!\\n";}
- }

```

As you can see in Line 7, we assemble an array of keys and use the `get_multi` subroutine to retrieve a hashref of values. Then, in Line 17 we can loop through the hashref and determine if any value was returned.

2.3 What are the benefits of Using a Client?

With the basics of how to use a client covered, let's talk a bit about what the client is doing for us. Aside from encapsulating our access to memcached for a consistent interface, a client provides the following benefits:

Server hashing is handled by the client so you may wonder why I'm talking about it at all. The fact of the matter is that this section will benefit you when it comes to understanding how the temporary loss of a memcached server will affect performance. You can safely skip this and implement memcached but you should now it's here so you can come back to it to better understand how to get the most from your memcached implementation.

server hashing

1. It uses a hashing algorithm to determine which server the *NVP* should be stored on or retrieved from.
2. It can provide the ability to compress data so that we can store *NVPs* with values that exceed 1MB in size.

Server Hashing

Server hashing is the act of determining where to store and retrieve *NVPs*. When we first started discussing memcached, we talked about the fact that every server is atomic, knowing only about itself and the data stored in it, totally oblivious to any other memcached servers that may exist. It's the job of the client to determine how to use multiple servers and it does that through *server hashing*. The best way to explain this is to demonstrate using a simple hashing algorithm that determines which server to use based on the first character of a key. This simple example allows only 36 possible values of A-Z and 0-9 and ignores case so the algorithm divides 36 by the number of servers available to come up with a dividing point for where to allocate and look for keys. The table below shows which server the key would be hashed on for 4 servers, 3 servers and 2 servers.

Server	4 Server Hash	3 Server Hash	2 Server Hash
S1	0-8	0-B	0-H
S2	9-H	C-N	I-Z
S3	I-Q	O-Z	
S4	R-Z		

There are two things to consider regarding how a client implements server hashing. The first is whether or not the order of the servers matters. In this simple example, if one program is using S1, S2, S3, S4 and another is using S2, S3, S4, S1 then the two programs will

never be able to share the cache. While we can minimize the impact this would have on any program we write by making sure we always use the same list of servers in the same order for any programs that are designed to share the cache, consider the impact if one of the servers in the list is no longer available.

If we have a client that uses this hash and we lose a server, taking us from 4 servers to 3 servers, you can see that all of the values that had been stored on S2 from 9-B are now going to be looked for, not found, and stored on S1. The same goes for the values stored on S3 from I-N, they are now going to resolve to S2. All of which will lead to extra overhead as the servers have to adjust to the new server hash based on three servers.

Data Compression

Not every client supports compression. The Enyim .NET client is one of those but it should be in the next release.

There is one other aspect of clients that we're going to touch upon before we wrap up this chapter: data compression. The memcached server does nothing except store, retrieve and expire *NVPs* with a limit of 1MB for the size of a value it will store. If you want to be able to store items larger than 1MB in memcached, then either you need to compress them before passing them to the client or the client needs to compress and expand them for you, which the Perl client does. While there are a number of settings you can change that are defined on the CPAN site,⁵ we are going to look at two: `set_compress_threshold` and `enable_comprss`. The first setting tells the client how large a value has to be before it gets compressed and the second will turn compression on or off. `enable_comprss` only works, however, if the compression threshold has been set. For the

⁵<http://search.cpan.org/dist/Cache-Memcached/lib/Cache/Memcached.pm>

last example in this chapter, we need a large amount of data. You can download a copy of Shakespeare's *Tempest*, formatted in HTML, using the following command:

```
wget http://ih.27south.com/shakespeare/tempest/full.html6
```

Now we're ready to test compression in memcached.

Download `LinuxInstall/memcachedCompressionExample.pl`

```
#memcachedCompresionExample.pl
#!/usr/bin/perl
use Cache::Memcached;
# Set the server list up
my $memd = new Cache::Memcached {'servers' => [ "127.0.0.1:11211" ],};

#flush the servers to remove any unwanted data before testing.
$memd->flush_all();
#set compression level
$memd->set_compress_threshold(1_000);
$memd->enable_compress(1);
open(HANDLE, "full.html");
undef $/;
    $raw_data = <HANDLE>;
close(HANDLE);
$memd->set("Compressed", $raw_data);
$memd->enable_compress(0);
$memd->set("Uncompressed", $raw_data);
#disconnect from all servers
$memd->disconnect_all();
```

First we load the entire file into a variable and then we turn compression on and store the data in memcached. Next we turn com-

⁶wget is a tool built into many Linux systems that downloads files from the web. There are versions of this available for Windows as well, one being wget for Windows <http://gnuwin32.sourceforge.net/packages/wget.htm>

pression off and store it under a different name in memcached. If we telnet into the memcached server to look at the results get Uncompressed we see the plain-text HTML scroll past us. If we try to execute get Compressed, however, the equivalent of line noise scrolls past us because the text has been compressed.

2.4 Review

In this chapter we've covered downloading and installing the memcached server on both Windows and a Linux distribution, downloading and installing a Windows client and the Perl client, and looked at the basics of using the Perl client to store and retrieve data from the server. We also looked at how to compress the data to fit more data in the server. And we talked about the importance of understanding how client's hash data across servers and what steps we need to take if a server doesn't consistently hash in order to minimize the impact of hashing on our code.

In the next chapter we'll talk more in-depth about using memcached by taking code that starts out querying the database every time and working through the steps to implement a more scalable system.



The Basics of Implementing memcached

Now that we've covered the basics of memcached, it's time to implement it in an application.

In this chapter, we're going to take a MySQL based calendar of events implemented in Perl and walk through implementing memcached in its simplest form. Then, step by step we'll refine that implementation, overcoming common obstacles along the way until we've reached our final goal: an implementation that will provide a pattern you can use to implement memcached in your own applications. While all of the examples here are in Perl, there are C# examples available in the downloadable code with the same solution name as the Perl program. But, before we dig into programming, let's talk a bit about where memcached fits in the overall caching scheme.

3.1 How Does memcached Fit in the Cache System?

memcached does not exist in a vacuum. And there may be other caching systems built into your existing application that you don't even realize exist. One key to using memcached effectively is understanding what other caches may be involved so that you are not duplicating effort. For example, many web servers implement some form of caching, perhaps by storing the end result of a dynamically generated web page in memory and serving it up from there without you having to do anything at all. Thus it makes no sense to store the *completed* web pages in memcached if they are being cached else-

where, but creating HTML fragments that are used by multiple web pages in their creation is a perfect example of utilizing memcached.

Databases may also have a cache element built in. MySQL, for instance, will cache the results of a SELECT query and will return them from memory. In a very simple example, if we test getting data from memcached versus getting the data directly out of MySQL where nothing else is using the database, it may almost seem like memcached is extra overhead that is not necessary, since MySQL's caching will return the data just as quickly as memcached will. In a real world scenario, however, MySQL will be handling many data requests that will be changing what data it is caching, meaning MySQL is much more likely to have to go to disk to get the data than to get it from its internal cache. So caching data requests from MySQL in memcached results in a more scalable site.

3.2 What is the Basic Coding Pattern for Using memcached?

Before we can implement memcached, we need to have an application to work with. The following program builds an HTML table containing a calendar of events for a website based on data stored in a database.¹

[Download](#) LinuxBasics/GetMySQLCalendarOfEvents.pl

```
Line 1  #!/usr/bin/perl
-      use strict;
-      use DBI;
-      # DBI configuration
```

¹The code to create this MySQL database is available in the downloadable code examples.

```

5  my $dsn = "DBI:mysql:memcachedTesting:localhost";
-  my $user = "mcd";           #Notice, not root
-  my $pw = "n0pass";         #Notice, password
-  my ($id, $password);
-  # Connect to server and database
10 my $dbh = DBI->connect($dsn, $user, $pw);
-  # Define Select query
-  my $myquery = $dbh->prepare(qq{select EventID, EventDate, Description, Details
-      from calendarofevents where WebSiteID = 1});
-  my $table = "";
15 # Execute the query
-  $myquery->execute();
-  # Start an HTML Table
-  $table = "<table border='1'><tr><th>EventDate</th><th>Details</th></tr>";
-  while (my ($EventID, $EventDate, $Description, $Details, $WebSiteID) =
20      $myquery->fetchrow_array()) # keep fetching until there's nothing left
-  {      $table = $table . "<tr><td>"
-          . $Description . "</td><td><a href=BookReservation.html?EventID="
-          . $EventID . ">". $Description . "</a><br />"
-          . $Details . "</td></tr>\r\n"; }# End While
25 #Close query table
-  $myquery->finish();
-  #Close HTML table
-  $table = $table . "</table>";
-  print $table;

```

A very simple program, it connects to the database and gets a set of data containing the EventID, EventDate, Description, and Details for the WebSiteID 1 and then loops through the data set to create the table with rows and hyperlinks to pages with more detail for the event. The problem with this code is that it has to query the database everytime it gets called, so we'll begin our implementation of memcached by storing the requested data in memcached.

The most basic pattern of memcached implementation is to look in memcached for the data first and, if it isn't not found, get the

data from the database and load it into memcached. How you do that depends on which client you are using. While it would be nice to take and just store `$myquery` from line 16 in the code above in memcached, the Perl client won't let us. Instead we need to put the data in a hashref and store that, something that we can easily do using `selectall_hashref`, which returns the data as a hashref. This does mean we need to make some minor changes to the code that creates the table in the course of implementing memcached. Let's look at the code and then step through the changes.

[Download](#) `LinuxBasics/memcachedCalendarOfEvents-1.pl`

```

Line 1  #!/usr/bin/perl
-      use strict;
-      use DBI;
-      use Cache::Memcached;
5
-      # Set the server list up
-      my $memd = new Cache::Memcached {'servers' => [ "127.0.0.1:11211"],};
-      $memd->set_compress_threshold(1_000);
-      $memd->enable_compress(1);
10     # DBI configuration
-      my $dsn = "DBI:mysql:memcachedTesting:localhost";
-      my $user = "mcd";           #Notice, not root
-      my $pw = "n0pass";         #Notice, password
-      my $table = "";
15     #Check memcached
-      my $data = $memd->get("CalendarOfEvents");
-      if (not $data) {
-          # Connect to server and database
-          my $dbh = DBI->connect($dsn, $user, $pw);
20     my $statement = "select EventID, EventDate, Description, Details
-                   from calendarofevents where WebSiteID = 1";
-      my $key_field = "EventID";
-      $data = $dbh->selectall_hashref($statement, $key_field);
-      $memd->set("CalendarOfEvents", $data);

```

```

25     }
-     #Whether from memcached or from the database, I have a hashref to parse
-     $table = "<table border=1>";
-     while( my ($k, $v) = each %$data ) {
-         $table = $table . "<tr><td>"
30         . $v->{EventDate} . "</td><td><a href=BookReservation.html?EventID="
-         . $v->{EventID} . ">". $v->{Description} . "</a><br />"
-         . $v->{Details} . "</td></tr>\r\n";
-     }
-     $table = $table . "</table>";
35 print $table

```

The Enyim .NET client requires a serializable object to store in memcached. For that reason, all of the .NET examples change the data into an XML string before storing in memcached.

The first change is the addition of memcached, in line 4 and the definition of the servers in lines 7-9. Then, in lines 16-25, we handle the actual implementation of memcached in this application. First, we attempt to get data out of the cache in line 16. If the data doesn't exist in memcached, then `$data` will fail to resolve and we will go to the database to get the information. Here, in lines 22 and 23, we have to implement changes to set the data into a format that the memcached client can handle. In line 22 we define the key field that we can use to retrieve data from the hashref, `EventID`. Then we execute `selectall_hashsref`, passing in that key field and get a hashref in return, which we store in memcached in line 24.

Whether from memcached or the database, `$data` now contains a hashref that represents rows of data from the database. Since this is in a format different from the array in our original code, we have had to modify the code that builds the HTML table in lines 28-34. Rather than loading the values into individual variables, we are populating two variables, the key field defined in line 22 and an array of values associated with that key field. And now we have memcached in place. And we run into our first problem.

What do I store?

A couple of fundamental questions come up in every memcached implementation. The first being "What do I store?" Take another look at the code above and see if anything jumps out at you as a further candidate for caching. The goal of using a cache is to limit the amount of work done somewhere. It's faster to deliver data from memory than it is from disk, so that's an obvious thing to cache. But everytime we execute this section of code, we are going to process a half dozen lines of code to build an HTML table from it. Since this is a fragment of a web page and not the whole thing, it probably won't be cached by our webserver, so it makes sense to cache the HTML fragment and save ourselves the trouble of building it at all. So the question becomes, should we store the HTML fragment or the data we use to create the HTML fragment? But it may not be an either/or question. In a large website, we may format the same data differently for use in different places. So it makes sense to cache *both* the underlying data and the HTML fragment it creates.

With that in mind, we'll rewrite the code to check for the formatted table in memcached and deliver that if it exists. If the formatted table is not there, check for the data in memcached. If it exists, create and store the formatted table in memcached and deliver it up. If the data doesn't exist, query the database, store the data in memcached, build the formatted table, store the formatted table in memcached, and deliver the formatted table. But, before we start changing the code to handle this better strategy, we need to talk about keys.

How Should I Name Keys?

It's been said that there are only two difficult problems to solve in programing: caching invalidation and naming. And here we get to deal with both. While `CalendarOfEvents` is an easy to read name, it's a bit limited in terms of usefulness. If we have multiple web sites running with different `WebSiteIDs` and each of them uses `CalendarOfEvents` as the key in memcached, it won't take long before the web site with ID 1 stores it's calendar of events in memcached and that's the data that gets retrieved and displayed by the web site with ID 2. One way to get around this is to format the name of the key to reflect not just what the data is but what it contains. One suggestion is a three part key separated with colons, something in the form of `ObjectName:ObjectType:Key`. In the example outlined above we might use `CalendarOfEvents:TableOutput:1` for the HTML fragment and `CalendarOfEvents:Data:1` to store the data from the database.

Another useful way to generate a key for *data* calls is to use the entire SQL request as the key. There are two challenges to overcome with this plan. The first is that memcached limits the key size to 256 bytes. While our simple `SELECT` in this example is only 90 bytes, more complex SQL statements quickly grow beyond that limit. Second, even though memcached doesn't list the keys stored in it, if someone *did* connect to your memcached client and knew some basics of your data structures, they could start making requests in attempting to get cached data. Fortunately, both of these challenges can be overcome by using a Secure Hash of our original key to store the data. Using SHA512, we can convert:

```
select EventID, EventDate, Description, Details from calendarofevents where Web-
SiteID = 1
```

There are a number of standard algorithms used for hashing data. The National Security Agency came up with five cryptographic hash functions whose goal is to take the contents of a message of any length and come up with a fixed length digital representation of the data unique enough that any change in the original data will result in a new hash. For our purposes, we don't necessarily need the encryption as much as a reliable way to turn some unknown length of data into a unique and reproducible value.

More good information on these and more hashes can be found at http://en.wikipedia.org/wiki/SHA_hash_functions.

into:

```
+AxCIPywhgxGcdUblad0eTYxKomKu/z9856dnr0tLqB
LUzdBBmQCKs4/8AZ6lLgd1Fe5tA4fIY305HhVkBb7SQ
```

The main point of key names is to be consistent, regardless of the scheme you settle on. For the rest of the demonstrations, I'm going to use the actual SQL as the key for any data objects I store and I'll use the `ObjectName:ObjectType:Key` format for anything else. Additionally, I'll hash the key for general security purposes.

Implementing Better Keys and Caching

Now that we've talked about what we should cache and come up with a more reasonable naming convention, let's implement that in code.

[Download](#) `LinuxBasics/memcachedCalendarOfEvents-2.pl`

```
Line 1  sub GetCalendarOfEvents
-      {
-      my $WebSiteID = shift;
-      my $tableKey = sha512_base64("CalendarOfEvents:HTMLTable:".$WebSiteID);
5      my $table = $memd->get($tableKey);      #Look for the HTML Table
-      my $statement = "select EventID, EventDate, Description, Details ".
-          "from calendarofevents where WebSiteID = ".$WebSiteID;
-      my $dataKey = sha512_base64($statement);
-
-
10     if (not $table) {
-         print "going to memcached for data object\n";
-         my $data = $memd->get($dataKey);
-         if (not $data) { # Connect to server and database
-             print "going to database\n";
15         my $dbh = DBI->connect($dsn, $user, $pw);
-         my $key_field = "EventID";
-         $data = $dbh->selectall_hashref($statement, $key_field);
-         $memd->set($dataKey, $data);
```

```

-   } #end not $data
20
-   #Whether from memcached or from the database, I have a hashref to parse
-   $table = "<table border=1>";
-   while( my ($k, $v) = each %$data ) {
-       $table = $table . "<tr><td>"
25       . $v->{EventDate} . "</td><td><a href=BookReservation.html?EventID="
-       . $v->{EventID} . ">". $v->{Description} . "</a><br />"
-       . $v->{Details} . "</td></tr>\r\n";
-   } #end while loop through %data
-   $table = $table . "</table>";
30   $memd->set($tableKey, $table);
-   } #end not $table
-   $table;
-   }

```

The first thing you may notice is that I've wrapped all of the functionality of building the table of events HTML fragment into a subroutine that can be called. Now the code can be called with any valid website id as a parameter:

```

Line 1  my $table = "";
-   $table = &GetCalendarOfEvents(1);
-   print $table;

```

We put the database id into a variable in Line 3. Then we create a string to hold the hash of our request—CalendarOfEvents:HTMLTable:1—and look in memcached to see if the HTML fragment exists. If it doesn't, we create a variable that contains the SQL we will use to query the database and another variable to store the hash of that SQL so we can first query memcached to see if the data is stored there. Finally we build the table if we need to and return it. This is a much more useful implementation of memcached than our first one because it caches the HTML fragment in addition to the data and it reduces key name conflicts within memcached.

3.3 How Do I Update memcached When the Data Changes?

So, now we can put data into memcached and we've got a reasonable pattern for getting data out, whether it's an actual database result set or formatted HTML, but what do we do when the data is no longer valid? When someone updates the calendar of events database table for the website we are caching, our cache is now stale. There are a couple of ways to handle this. The easiest one would be to flush the cache, which would guarantee that the freshest data would be delivered but it would also force the reloading of all cached data, something that should generally be avoided as it will cause a lot of database activity. A much better solution is to proactively clear the cache.

Proactively Clearing the Cache

The first method we're going to explore is proactively clearing the cache when an update or insert is made. The following subroutine is a standard update but, once we've finished the update we delete the HTML table and the data rowset from memcached.

[Download](#) LinuxBasics/memcachedUpdateCalendarOfEvents-1.pl

```

Line 1  sub UpdateCalendarOfEvents
-      { #load parameters
-          my ($EventID, $EventDate, $Description, $Details, $WebSiteID) = @_;
-          my $dbh = DBI->connect($dsn, $user, $pw);
5       #update the database
-          my $update_handle = $dbh->prepare_cached('UPDATE calendarofevents
-              set EventDate=?, Description=?, Details=?
-              WHERE EventID=?');
-          $update_handle->execute($EventDate, $Description, $Details, $EventID);
10

```

```

-   #remove from memcached
-   my $tableKey = sha512_base64("CalendarOfEvents:HTMLTable:".$WebSiteID);
-   my $statement = "select EventID, EventDate, Description, Details ".
-       "from calendarofevents where WebSiteID = ".$WebSiteID;
15  my $dataKey = sha512_base64($statement);
-   $memd->delete($tableKey);
-   $memd->delete($dataKey);
-   }

```

Lines 1-9 are the basic update routine. Line 3 loads variables with the arguments passed in, Line 5 defines the SQL command to update the database and Line 9 actually executes the update. It's from lines 11 on that we are interested in, however. Line 12 defines the hashed key for the HTML fragment and Lines 13-15 define the hash key for the SQL command. Using those keys, we delete the data from memcached in lines 16 and 17.

Since the code in the `GetCalendarOfEvents` subroutine prints whether it has to go to memcached or the database we can test that the update does what we expect. If memcached hasn't been cleared from previous runs, the first call to `GetCalendarOfEvents` in the code below gets data directly from the cache. Then we update the database and the next call to get the HTML table has to build the table from the database. The following section of code demonstrates updating the database by calling the `UpdateCalendarOfEvents` routine to update the database and clear the cache of the now stale data.

[Download](#) `LinuxBasics/memcachedUpdateCalendarOfEvents-1.pl`

```

$table = &GetCalendarOfEvents(1);
&UpdateCalendarOfEvents(1, "2008-01-01", "New Year's Day 2008",
                        "Time to start anew!",1);
$table = &GetCalendarOfEvents(1);
print $table;

```

Resetting the Cache

The second method we'll explore is very similar to the first but rather than just deleting the keys and waiting for the next request for the data to refresh the cache, we reset the values after our update/insert. The easiest way to do this is to add a flag to our existing `GetCalendarOfEvents` that forces going to the database. The following code fragment demonstrates the changes in the control structure:

Download `LinuxBasics/memcachedUpdateCalendarOfEvents-2.pl`

```
sub GetCalendarOfEvents
{
    my $WebSiteID = shift;
    my $ForceOverRide = shift;
    if (not $table or ($ForceOverRide)) { # Load the data from source
```

There is at least one instance where your cached data can go out of synch with the database even if we have implemented a proactive cache to keep it from expiring. That would be occasions where you update the database using direct database calls through a query browser or any other mechanism that doesn't know about memcached. In those cases you have several options:

- flush the cache
- have a program that knows how to generate and rebuild the data in memcached
- not use any mechanism that doesn't know about memcached

We'll talk more about this in Section 3.5, *How Do I Determine the Optimal Expiration Time?*, on page 46

3.4 How Do I Prevent Multiple Clients Updating One NVP?

Cache Stampedes

One advantage that resetting the cache has over clearing the cache is that it helps avoid *Cache Stampedes*. A cache stampede is created when multiple clients try to update the cache with the same information. Let's say that we have just updated something and that deleted both the data rows and the HTML fragment from memcached. And now, thirty clients are running the `GetCalendarOfEvents` code, all at the same time, so each and every one of them looks in memcached, finds the data missing and queries the database, stores the results in memcached, builds the HTML fragment and stores that in memcached. Or, instead of thirty clients, hundreds, all hitting the database for the same query, which is what we are trying to avoid by implementing memcached. On our extremely simple calendar query, that may not be an issue, but a more complex one that actually requires the database server to devote some serious resources to fulfilling it can have a detrimental impact on site performance. Refilling the cache on updates is a valuable way of helping to minimize cache stampedes, but it has no impact when the data being requested isn't in the system.

What we need in this instance is some way to tell our program that another program is working on fetching the data. The best way to handle that is by using another memcached entry as a lock. When our program queries memcached and fails to find data, the first thing it attempts to do is to write a value to a specific key. In our example where we are using the actual SQL request for the key name we can just append `":lock"` to the SQL to create our new key. What we do next depends on whether the client supports returning success messages on memcached storage commands. If it does,

then we attempt to ADD the value. If we are the first one to attempt this then we'll get a success message back. If the value exists then we get a failure indication and we know that another process is trying to update the data and we wait for some predetermined time before we try to get the data again. When the process that's updating the cache is done, it deletes the lock key.

If our client doesn't support returning status information then we can query the lock key and, if we get no data back, write some random value there to let other processes know we are working on filling the cache and delete the key when we are done. This version isn't as efficient as using ADD to generate a pseudo-lock because there are two memcached operations (GET and SET) instead of one, but it is still very efficient code.

The first step in implementing code to avoid cache stampedes is to separate the section that gets the data from the database into a subroutine. While not strictly necessary, it does make the code easier to read. The big change in this routine is the logic for checking to see if anyone else is attempting to read the data from the database.

[Download](#) LinuxBasics/memcachedUpdateCalendarOfEvents-3.pl

```

Line 1 sub GetCalendarOfEventsDB
- {
-     my $statement = shift;
-     my $dbh = DBI->connect($dsn, $user, $pw);
5     my $key_field = "EventID";
-     my $lockKey = sha512_base64($statement."lock");
-     my $key = sha512_base64($statement);
-     my $lockCount = 0;
-     my $returnValue = "";
10 while(1 > 0) {
-     #Check to see if anyone else is attempting to get data
-     if($memd->get($lockKey) && $lockCount < 3) #Someone is making the attempt

```

```

-      { $lockCount ++;
-        usleep('500_000'); # sleep a half second
15      if($memd->get($key)) #Can get from memcached
-        { $returnValue = $memd->get($key);
-          print "got value from someone else stuffing it in memcached\n";
-          last;}
-      }
20    else #we are going to make the attempt
-      {
-        $memd->set($lockKey, [gettimeofday]);
-        $returnValue = $dbh->selectall_hashref($statement, $key_field);
-        print "got data from database\nstoring in memcached:\n";
25      $memd->set (sha512_base64($statement), $returnValue);
-        $memd->delete($lockKey);
-        last;
-      }
-    }
30    $returnValue;
-  }

```

We create a hashed key of the SQL statement we are looking for plus :lock, to remain consistent with our naming convention. Then we do a quick get on the lock key in Line 12. If it returns anything then we know that someone is trying to get the data so we increment lockCount on Line 13 and go into a wait state for half a second on Line 14. When done waiting we check memcached to see if the data's been placed there yet (Line 15). If it is, we're done. If not, we go through the wait state two more times. Finally, if we have gotten through our loop the maximum number of times and there's no data in memcached we go straight to retrieve it from the database and store the results in memcached. In either case, whether the data is pulled from memcached because someone else found it or from the database, we return a rowset.

In this example, the longest we would wait before going to the database is about one and a half seconds. That's more than long enough for the simple query we have. But you may have more complex, longer running queries and you should adjust the time and number of times you wait accordingly. The goal is to minimize the impact of data queries and if that means that five clients are attempting to execute the same data request rather than fifty, we've met that goal.

To see how effective this is, I ran twelve versions of the code example from Section 3.3, *Proactively Clearing the Cache*, on page 39 simultaneously and flushed memcached to see what messages I got. This resulted in anywhere from 1 to 4 of the programs having to get their data out of the database, usually 2 of them. With 12 simultaneous versions of this code running, I got the same number of clients not finding the data in memcached but only one hit the database, the other programs waited and got the data from memcached after the first client put it there.

The only real problem left with this subroutine is that it's still too specialized. We'll need to repeat this code by cutting, pasting, and then modifying the SQL for every database call we want to make. But we can avoid that by adding a couple more arguments to the subroutine. We'll pass in the key field, the number of cycles to wait without the data in memcached before going to retrieve it, and how long each wait should be. The resulting code makes a good pattern to use for minimizing cache stampedes.

[Download](#) LinuxBasics/memcachedUpdateCalendarOfEvents-4.pl

```

Line 1 sub GetFromDB
- {
-     my ($statement, $keyField, $maxLockCount, $waitTime) = @_;
-     my $dbh = DBI->connect($dsn, $user, $pw);
5     my $lockKey = sha512_base64($statement." :lock");
-     my $key = sha512_base64($statement);
-     my $lockCount = 0;
-     my $returnValue = "";
-     while(1 > 0) {
10         #Check to see if anyone else is attempting to get data
-         if($memd->get($lockKey) && $lockCount < $maxLockCount) #Someone
-             # is making the attempt
-         { $lockCount ++;
-           usleep($waitTime); # sleep a half second

```

```

15         if($memd->get($key)) #Can get from memcached
-           {$returnValue = $memd->get($key);
-             last;}
-       }
-     else #we are going to make the attempt
20     {
-       $memd->set($lockKey, [gettimeofday]);
-       $returnValue = $dbh->selectall_hashref($statement, $keyField);
-       $memd->set ($key, $returnValue); #Stick in memcached
-       $memd->delete($lockKey);
25       last;
-     }
-   }
-   $returnValue;
- }

```

3.5 How Do I Determine the Optimal Expiration Time?

memcached can expire items from the cache based on either a fixed duration (the number of seconds up to 30 days) or a date (based on Unix time). Before you begin to set expiration times, however, you should ask whether you need to use them at all. One reason to set an expiration time is to keep items in the cache from getting stale, that is, representing old data that may no longer be the correct data. If memcached is implemented in a way that proactively updates the cache with the data, then this is less of a problem because the data in memcached should reflect the most current data known to your program. Still, if you want to use expiration times, there are some general rules for determining how best to set that expiration period.

The two key pieces of data required for determining expiration times are how frequently the cached data is requested and how frequently

the cached data is updated in the database. The more frequently a set of data is updated, the shorter the expiration date. Likewise, a piece of data that's rarely called may never need an expiration date because it very likely will get removed from the cache because space is needed before it ever gets updated. But why do expiration times matter if we're updating memcached whenever we update data in the database?

There are two reasons expiration dates matter. The first is when we have data that we don't want to update the cached data every time the backend data is updated. One example of that might be an RSS feed of forum topics. Since this may be a volatile list, with additions made every minute, we might want to just assemble an updated RSS feed every five minutes. In that case, we can set an expiration time of five minutes on the feed data and just rebuild it when it expires (or is deleted by the cache).

We can utilize the same type of scheme with data that is being updated outside of our programs. If we are getting feeds that continually update the database, it may be more efficient to have the data displayed on the web site expire and get rebuilt on a regular basis.

How Can I Proactively Fill Expired Items?

In Section 3.4, *How Do I Prevent Multiple Clients Updating One NVP?*, on page 42, we addressed one of the problems that can occur when data is not found in memcached. But even if we minimize cache stampedes, expiring data from the database can still lead to clients waiting for data while it gets refreshed. The fact is that whenever something expires or gets deleted from the cache you run the risk of having multiple clients try to fill the cache, or at least waiting around while one of the other clients does. Fortunately, there's a

way around that by implementing a proactive cache refill. What that does is associate an internal refresh time separate from the expiration time.

In the following example, we'll store a refresh value of 10 seconds with a 20 second expiration. The 20 second expiration will be automatically handled by memcached, but we write our code so that the first client to pull data from the cache, when the current time is later than the refresh time, takes the following steps:

1. Save the current data with a new expiration and refresh times. This keeps other clients using the current data while we query the database for the most current data.
2. Query the database to get the current information.
3. Store that in memcached with a new expiration and refresh time.

[Download](#) LinuxBasics/memcachedRollingCalendarOfEvents.pl

```

Line 1  sub GetFromDB
-      {
-      my ($statement, $keyField, $maxLockCount, $waitTime) = @_;
-      my $dbh = DBI->connect($dsn, $user, $pw);
5      my $lockKey = sha512_base64($statement."lock");
-      my $refreshKey = sha512_base64($statement."refresh");
-      my $key = sha512_base64($statement);
-      my $lockCount = 0;
-      my $returnValue = "";
10     while(1 > 0) {
-         #Check to see if anyone else is attempting to get data
-         if($memd->get($lockKey) && $lockCount < $maxLockCount) #Someone
-             #is making the attempt
-         { $lockCount ++;
15         usleep($waitTime); # sleep a half second
-         if($memd->get($key)) #Can get from memcached
-             {$returnValue = $memd->get($key);

```

```

-         last;}
-     }
20     else #we are going to make the attempt
-     {
-         $memd->set($lockKey, [gettimeofday]);
-         $returnValue = $dbh->selectall_hashref($statement, $keyField);
-         $memd->set ($key, $returnValue);
25         $memd->set ($refreshKey, time + 30);
-         $memd->delete($lockKey);
-         last;
-     }
- }
30 $returnValue;
- }
-
- sub GetCalendarOfEvents
- {
35     my $WebSiteID = shift;
-     my $ForceOverRide = shift;
-     my $statement = "select EventID, EventDate, Description, Details ".
-         "from calendarofevents where WebSiteID = ".$WebSiteID;
-     my $tableKey = sha512_base64("CalendarOfEvents:HTMLTable:".$WebSiteID);
40     my $tableRefreshKey =
-         sha512_base64("CalendarOfEvents:HTMLTable:".$WebSiteID.":Refresh");
-     my $hashref = $memd->get_multi(@mcdRequests);
-     my $table = $memd->get($tableKey);
-     my $refreshTime = $memd->get($tableRefreshKey);
45     if ((not $table) || ($refreshTime < time) )
-     {
-         print "rolling update\n";
-         # Update memcached to reflect a new rolling expiration
-         $memd->set($tableRefreshKey, time + 30);
50         $memd->set($tableKey, $table);
-         # Force over ride
-         $ForceOverRide = 1;
-     }
-     my $dataKey = sha512_base64($statement);

```

```

55  if (not $table || ($ForceOverride)) {
-    print "going to memcached for data object\n";
-    my $data = $memd->get($dataKey);
-    if (not $data or ($ForceOverride)) {
-        # Check to see if another process is getting data
60    print "going to database\n";
-    my $keyField = "EventID";
-    my $maxLockCount = 3;
-    my $waitTime = '500_000';
-    $data = &GetFromDB($statement, $keyField, $maxLockCount, $waitTime);
65    } #end not $data
-    #Whether from memcached or from the database, I have a hashref to parse
-    $table = "<table border=1>";
-    my $dataStored = @data[0];
-    while( my ($k, $v) = each %dataStored ) {
70        $table = $table . "<tr><td>"
-        . $v->{EventDate} . "</td><td><a href=BookReservation.html?EventID="
-        . $v->{EventID} . ">". $data->{Description} . "</a><br />"
-        . $v->{Details} . "</td></tr>\r\n";
-    } #end while loop through %data
75    $table = $table . "</table>";
-    $memd->set($tableKey, $table);
-    $memd->set($tableRefreshKey, time+10);
-    } #end not $table
-    $table;
80 }

```

The new functionality in this code sample begins on line 39, where we define a new hashed key to look for in memcached with the object type of :refresh. This contains the time value that defines when we want to refresh the data. Then, in Line 41, we retrieve that value from memcached. If either the HTML fragment is empty or the refresh time is past, then we start the process of rebuilding the data.

The first thing we do is update memcached with the existing HTML fragment, using a new expiration time, and update the Refresh value to be 30 seconds in the future. This prevents memcached from expiring the HTML fragment while we update it and it prevents another client from attempting to update the data as well. Then we go through all the steps necessary to get the data from the database and update memcached with the update information. Finally, in Lines 74-75, we update memcached to reflect the updated data and refresh times. Since this code is added to all of the refinements we've made, we have code that gives us the best pattern for implementing memcached.

Now that we have all of the basic patterns we can use for implementing memcached at the code level, it's time to talk about something more esoteric.

Even if you follow all of the best practices for utilizing memcached, there's no guarantee that the data will be there until the expiration time passes. If memcached needs the memory that an NVP is occupying *and* it hasn't been used recently enough that it falls through the LRU filter, it will not be there and you'll need to be ready to reload the data.

3.6 Can I Eliminate Caching Duplicate Data?

In Section 3.2, *What do I store?*, on page 35, we talked about storing not just the data but also the HTML fragment the data was being used to build. In this section we're going to take a different look at our data to review another way to cache data, beginning with another look at data in general.

Most data can be placed into one of two types: a list of data items or an individual item. Throughout our example we've been looking at retrieving and caching a list of items that include 5 columns of information. But there is another way to cache the data that may look to be less efficient at first glance; however, looks can be deceiving. If

you take a closer look at the table we've been building throughout this book, you'll notice that it builds a link to each individual event's detail page. If we cache the data used to build that detail page, we will probably have duplicate data loaded: individual event data that makes up the *list* of dates and individual event data stored as discrete objects used on the detail page. One way to get around that is to think of lists of data as being little more than pointers and use that list to get the individual items.

Let's take a look at putting that into practice and then discuss how it can be beneficial. To implement this data model, we're going to make a couple of changes. First, our `GetDataFromDB` procedure can be renamed to `GetSetFromDB`, more accurately reflecting that it retrieves lists of data from the database. Now we can add a new procedure that gets individual rows of data, as shown here:

[Download](#) `LinuxBasics/memcachedUpdateCalendarOfEvents-5.pl`

```

Line 1  sub GetRowFromDB
-      {
-      my ($statement, $maxLockCount, $waitTime) = @_;
-      my $dbh = DBI->connect($dsn, $user, $pw);
5      my $lockKey = sha512_base64($statement.':lock');
-      my $key = sha512_base64($statement);
-      my $lockCount = 0;
-      my $returnValue = "";
-      while(1 > 0) {
10         #Check to see if anyone else is attempting to get data
-         if($memd->get($lockKey) && $lockCount < $maxLockCount) #Someone
-             #is making the attempt
-         { $lockCount ++;
-           usleep($waitTime); # sleep a half second
15         if($memd->get($key)) #Can get from memcached
-             {$returnValue = $memd->get($key);
-             last;}

```



```

-     }
-     else #we are going to make the attempt
20  {
-         $memd->set($lockKey, [gettimeofday]);
-         $returnValue = $dbh->selectrow_hashref($statement);
-         $memd->set ($key, $returnValue);    #Stick in memcached
-         $memd->delete($lockKey);
25  last;
-     }
- }
- return $returnValue
- }

```

The main difference between `GetRowFromDB` and `GetSetFromDB` is that the latter now returns only a list of keys, while the former returns a single hashref of values rather than a collection the way that `GetDataFromDB` does. Next we need to modify our `GetCalendarOfEvents` procedure to use the list of `CalendarEventIDs` to drive the getting of individual calendar events:

[Download](#) `LinuxBasics/memcachedUpdateCalendarOfEvents-5.pl`

```

Line 1 sub GetCalendarOfEvents
- {
-     my $WebSiteID = shift;
-     my $ForceOverRide = shift;
5  my $statement = "select EventID from calendarofevents ".
-         "where WebSiteID = ".$WebSiteID." order by EventDate";
-     my $tableKey = sha512_base64("CalendarOfEvents:HTMLTable:".$WebSiteID);
-     my $table = $memd->get($tableKey);    #Look for the HTML Table
-     my $listDataKey = sha512_base64($statement);
10  if (not $table or ($ForceOverRide)) {
-     print "going to memcached for data object\n";
-     my $listData = $memd->get($listDataKey);
-     if (not $listData or ($ForceOverRide)) {
-         # Check to see if another process is getting data
15  print "going to database\n";

```

```

-     my $keyField = "EventID";
-     my $maxLockCount = 3;
-     my $waitTime = '500_000';
-     $listData = &GetSetFromDB($statement, $keyField, $maxLockCount, $waitTime);
20 } #end not $listData
- #Whether from memcached or from the database, I have a hashref to parse
- $table = "<table border=1>";
- while( my ($k) = each %$listData ) {
-     #Need to resolve the individual EventIDs
25     $statement =
-         "select EventID, EventDate, Description, Details from calendarofevents "
-         "where EventID=".$k;
-     my $dataKey = sha512_base64($statement);
-     my $data = $memd->get($dataKey);
30     if (not $data) { $data = GetRowFromDB($statement, 3, "500_000");}
-     $table = $table . "<tr><td>"
-         . $data->{'EventDate'} . "</td><td><a href=BookReservation.html?EventID="
-         . $data->{'EventID'} . ">". $data->{'Description'} . "</a><br />"
-         . $data->{'Details'} . "</td></tr>\r\n";
35 } #end while loop through %dataList
-     $table = $table . "</table>";
-     $memd->set($tableKey, $table);
- } #end not $table
- $table;
40 }

```

The big change here can be seen in Lines 23-34, where we build the table fragment. Rather than looping through the hashref of keys and values, we now loop through a hashref of keys. We then need to make calls to get the data to use to build that row of the table. It seems like it adds extra overhead, but let's look at what this implementation buys us.

On objects that are used outside of a list, it brings us the ability to reuse cached data rather than having duplicate copies of the data in cache, one in list form and one in detail form. In addition, any

program that references the details of one row of the calendar of events now has a better chance of getting it out of cache. The second improvement this brings is in limiting the number of items that need to be expired out of the cache when a calendar item is updated.

In a version that stores the Event data in the list, any change to the title of an item in the list would require the removing or refreshing of not only the individual object, if it was stored, but also any lists that contain the item. If the cached lists are little more than pointers then they no longer need to be expired just because an item they contain has had data updated. Instead, they will display the current data because whenever an event is updated, we can update the individual item in cache and that's the data that gets used when the list is resolved.

Now for this simple application, coding something like this is simply overkill, so let's look at a more practical example: a photo sharing site that contains lists of photographs with links to detail pages. Many such sites display paged lists, so 100 pictures in a list might be displayed as thumbnails with basic information, ten per page. Based on all of the coding we've done, it might seem reasonable to go get a list of one page of data from the database, cache the resulting list and then get the second page and cache that list and so on. But there's another way to handle that.

Start by getting the list of all 100 items that make up the list and cache that. Build the paged lists from that information, rather than going back to the database for every page. Now the same list serves someone who only wants to see 5 items per page as well as someone who wants to see 20, which is better than having to query the database 20 times for one person and another 5 times for the other, when all we're doing is showing the same data paged differently.

The most important thing to remember is that this is not an all-or-nothing approach. You may have data that lends itself to storing nothing except lists of pointers, and then store the individual data items separately. But you might just as easily have data where it makes more sense to store lists that contain data. This is one of those things you need to experiment with and see which makes more sense for your application.

3.7 How do I Gauge Cache Efficiency?

In Section 1.8, *What Other Commands Can I Use?*, on page 15 we looked at using the `stats` command to be able to look at cache efficiencies, but it is a very broad gauge of effectiveness. When we want to know how well caching is working on a more granular level, you need to take a more proactive approach. One way to handle this is to have a debug flag that you can turn on and off. When it's on, use the `INCR` to track efficiencies. I use this on websites where I cache several different types of things, such as the name and format of the data calls as well as the actual data returned. When the debug flag is turned on, then every attempt to resolve the name and format of the data increments either `DataDefCached` or `DataDefMissed`, and every attempt to resolve a data request increments either `DataCached` or `DataMissed`. Over the course of time we can query those keys and see how well or poorly our caching is working.

This type of logging is not the kind of thing you want to be running on a production site. You might get away with incrementing a counter but logging every single request to file or database is always going to introduce a performance hit.

Sometimes we need to take it a step further and turn on actual logging to a file or database, in order to see which individual calls are being retrieved from cache and which are not, so we can adjust our caching accordingly. By logging every data request and whether it is served up from cache or the database, we get enough information

to help determine whether there are things that we feel should be cached that aren't, or that are falling out of the cache too quickly. We can use this information to tune what's being cached. We can also use that data to determine how caching HTML fragments in memcached impacts our database selections.

3.8 Review

In this chapter we covered a lot of the practical details of implementing memcached. We started with a piece of code that didn't implement any memcached and worked our way through implementing memcached in several variations that got progressively better. Along the way we discussed some of the obstacles encountered in implementing memcached and ended with a good working pattern for implementing memcached in your own code. In the next chapter we're going to step back and look at memcached from a more architectural level.



Best Practices

In the last chapter we covered a lot of the practical details of implementing memcached, but that was at a very technical level. We started with a piece of code that didn't implement any memcached and worked our way through implementing memcached in several variations that got progressively better. In this chapter we're going to take a broader look at implementing memcached on a more architectural level and review the best practices associated with memcached. Let's start with one of the issues that frequently comes up: Security.

4.1 How Can I Secure memcached?

The memcached server has absolutely no authentication model. Anyone who can connect to the server via the TCP or UDP port can get data. That's by design because authentication would slow down the processing. But that doesn't mean that memcached can't be secured, and that begins by limiting access to the servers. On a broad level, your memcached servers shouldn't be exposed to the internet. Those servers, or at least the ports memcached is operating on, should be blocked from outside access. The second way to secure the servers is to obfuscate your keys. That way, anyone who might be able to access the servers can't easily find data by doing gets and randomly replacing the where clause in SQL commands. Some clients already handle this by hashing your key and/or compressing data so it's not easily human readable.

The best practice is to limit access to the memcached server/port to the machines that require access. This can be done by utilizing

routing and firewalls to allow and disallow access to specific ports by IP Address or IP Range, a topic beyond the scope of this Friday but there is a link to some information in Section 6.1, *Firewall/Network Configuration resources*, on page 69.

4.2 How Do I Determine What Gets Cached?

If you want to implement memcached in an existing live application, then you've probably already defined the pain points; most often these are database queries that are being executed in the part of the system where your users complain the most about performance. While memcached may not be able to make a poorly designed SQL statement run any faster, it may be able to cache the results so that you don't always have to go to the database every time you want the information. So the first step is to determine what queries you have that run the most and longest and put those data results in memcached, so you can retrieve them from the cache rather than the database.

And, while data may be where caching starts, don't forget to look at the parts that make up the whole. While it doesn't make sense to cache the resulting HTML page (since there are other caching systems probably much closer to the web server that can store that data) it does make sense to store discrete, composed chunks of data that are used in multiple places. If you have a set of links containing updated headlines, it makes sense to deliver that HTML fragment containing the formatted anchor tags from memcached rather than recreating it with the data pulled from memcached every time you need to deliver it.

Best practices include implementing caching in a way that minimizes cache stampedes, just as we did by using locking and a rolling expiration time to try to minimize the number of clients trying to fill the cache with the same data.

4.3 How Do I Name Keys?

When it comes to best practices, you'll often find what amounts to almost theological debates related to naming conventions. Proper-Casing, camelCasing, object:key:type, etc. The two best practices are to be as granular as reasonable and to be *consistent* in how you name keys. Once you've determined what your keys are going to look like, make sure that you implement those names across your application. This is especially important if you will have more than one application sharing the cache. If one program stores data under `CalendarOfEvents-HTMLTable-1` and another uses `CalendarOfEvents:HTMLTable:1` for the same data then you'll have duplicate data stored in your cache, and a less efficient cache.

My recommendations are to store SQL statements using a hash of the SQL statement the way we did in our examples. If non-SQL items you are caching already have unique names, use a hash of those existing names. Otherwise implement an object:key:type convention.

4.4 Which Storage Command is Best?

Back in Section 1.6, *How Do I Manipulate Data?*, on page 8 we learned that there are three commands for storing an NVP in memcached: `add`, `set` and `replace`. In the case of `add` and `replace`, there's some additional overhead you need to worry about. If you attempt to

add a key that exists or replace a key that doesn't, you need to be prepared to get handle an error code and respond to it. This is different from using the `set` which will store the data regardless of whether the key already exists. So, what's the best command to store data in memcached?

The best way to answer that is to go back to the reason that `noreply` was implemented, suppressing the `STORED` or `NOT_STORED` indicator. memcached is a memory based cache. One of two things can happen when you request data from the cache: it will be there or it won't. It may not be there because it expired out of the cache, it may not be there because memcached needed the memory location that was being used, it may not be there because it was deleted on purpose, or it may not be there because it was never put there. Any program using memcached doesn't really care *why* the data isn't there, only that it has to look somewhere else to get the data. And if that's the case, why does the program need to know that the data was successfully stored? So no return code is provided.

In order to implement `noreply`, you must use a client that supports it.

Given that rationale, even if not taken to the extreme of using `NOREPLY`, best practices dictate that data storage should be done using `set`. There are exceptions to every rule, including this one, and we explored one situation in Section 3.4, *How Do I Prevent Multiple Clients Updating One NVP?*, on page 42 when we used `add` to set a lock to indicate that processing was going on.

If you find yourself using `add` and `replace` often, you may want to ask yourself what value you are getting from that code. At one point, I wrote a lot of code that used `add` and `replace`, based on a lack of understanding of how memcached worked. Once I realized that the code getting data out of memcached didn't care *why* the data wasn't in memcached, I cleaned up a lot of code to use `set`, giving me a

leaner, more efficient program.

4.5 How Do I Fill the Cache?

In this book, we've covered only one way to put data in the cache, reacting to requests for data. But there are many other ways to cache data, depending on what we are attempting to accomplish. If we have a set of data that is frequently requested by clients and frequently updated, then we may want to store that data without specifying an expiration date in memcached or using a rolling expiration. Instead, a separate program that runs on a regular basis is used to query the database and store information in memcached. The only time the normal client will go to the database is when the data isn't in the cache for some reason.

If you plan on proactively filling the cache, keep in mind that you still need to make sure you implement best practices that avoid cache stampedes. Just because you place something in the cache, doesn't mean that it will be there when you go to get it out.

If you are going to set up a job to refresh the cache separate from the application that is reading the cache, there are a couple of things that you have to keep in mind. First, make sure the memcached server list for both applications is the same. If it's not, it's possible that the reader application will update the cache from the database and that set of data will get delivered for a long time, even though it may have gone stale.

The second is to make sure the two client programs used by the reader and writer applications are either the same or compatible. A good example of why this is important can be found with the two clients used in this book. The Perl client stores data in memcached using the key provided to it. The C# client, on the other hand, hashes the key and uses that hash as the key for memcached. This doesn't affect our code since the C# client always knows what

it is looking for, but it does mean that any data stored by the Perl client will never be found by the C# client and vice versa.

4.6 How Can I Minimize memcached Server Outages?

memcached is a very stable tool. There are some folks who have said they consider their memcached servers to be like dial tone, it's just there and working. Be that as it may, however, one should always be prepared just in case something happens to one or more of your memcached servers. The beauty of having a distributed application like memcached is that losing a machine is not all that detrimental. There are, however, a couple of actions that can be taken to make life easier in the case of an outage, whether it's one memcached instance, half of your memcached instances, or all of them.

The first thing is to realize that even if you lose connectivity to all of your memcached servers, your program should still run, as long as it's using memcached as nothing more than a memory cache. That's not to say it won't impact the application. If one instance out of several goes down, there will be a temporary larger load on the database server while data that was in the lost instance is reloaded as the clients make requests. If your code has been written to minimize cache stampedes then this should be a minimal impact. Another way to handle this is to bring up an instance of memcached on a new machine using the lost machine's IP address so it can fill in.

Another option, and definitely a best practice, is to code so that you can easily change your memcached server list with minimal work. In the C# client, this is handled in the App.Config. In Perl you could

use a file that defines all of the servers and use `do` to include it. That way, should you lose some instances you can easily change your server list. You should also code so that you could turn off memcached if you needed to. Why would you need to do that?

Some memcached clients implement a timeout value. While this won't be an issue when everything is running smoothly, if all of your memcached servers go down, the client will keep trying until that time out limit is reached. If the timeout is one second and all of the memcached servers go down then your application will take that second every time it tries to connect to the servers. Having a convenient way to stop using memcached in those instances is the best way to survive the outage and still keep the application running.

4.7 What's the Future of memcached?

memcached has a fervent following and developers are still adding new features to it. Some of the features currently under discussion that are worth looking at include:

Binary Protocol

One of the most expensive pieces of code in memcached is the part that parses the text request. To help cut down on this expense there's a new Binary Protocol that's being implemented. If it is implemented in the client, you can use it and probably will not need to change your code at all, other than setting a flag to tell the client to use the binary protocol.

Tags

One of the complaints most often voiced by memcached users is the inability to invalidate a set of *NVPs* unless you know what all of their keys are. Tags will provide a way to identify a key as a part of a larger set. So, in our calendar of events sample we've been working on, we could tag the list of dates with each of the individual items. One example of this would be to add our hashed keys as tags to related items. So we might tag `CalendarOfEvents:HTMLTable:1` with the each of the `$dataKey` that we generated in Section 3.6, *Can I Eliminate Caching Duplicate Data?*, on page 51.

Then, when we update one of those individual items, we could invalidate the `$dataKey` which would invalidate all of the *NVPs* associated with that tag, both the individual data item and lists that contain that item.

memcached Add Ons

Not only is memcached popular as a memory cache you can implement into your application, but a number of ways to extend its functionality exist as well. This list will introduce you to some of these and offer a link to where to get more information. Before we get to listing applications and other uses of memcached, I want to reiterate, one more time, that memcached was designed as a *memory cache*, not a data store nor a high availability memory repository. Some of these links are to applications that work to add that functionality into memcached, either by starting with the memcached code to make their own program or by incorporating memcached into their application. I'm not using any of these applications so I can't judge how effective they are but if you find yourself thinking "wouldn't it be nice if I could modify memcached to do *fill-in-the-blank*" then you should research to see if anyone has already done it first.

memcached Storage Engine for MySQL

This project is designed to allow you to query memcached with a MySQL backend using standard SQL calls.

http://tangent.org/index.pl?node_id=506

Apache memcached Session

If you have a website, you've had to deal with session data. This application integrates memcached with Apache to enable storing session data in memcached.

<http://search.cpan.org/~enrys/Apache-Session-Memcached-0.03/>

repcached

While memcached isn't designed to be a high-availability application, repcached is a set of patches that is designed to make it one, complete with redundancy between a master and slave system with failover.

<http://repcached.lab.klab.org/>

memcachefs

The memcached file system. Allows you to view your cached data as though it were files on disk.

<http://memcachefs.sourceforge.net/>

memcachedb

That's a b at the end, making it memcached database; a persistent, distributed storage system that's a cross between memcached and Tugela.

<http://code.google.com/p/memcachedb/>

Additional Resources

This chapter contains links to resources mentioned throughout the book, interesting items that may help you as you implement memcached, and some resources for general information.

6.1 memcached Resources

The most important resource is, of course, the official memcached site, hosted by Danga Interactive at <http://www.danga.com/memcached/>. That's where you can find links to all of the current builds. It's also where you can join the memcached mailing list (<http://lists.danga.com/mailman/listinfo/memcached>), a handy resource for asking questions and getting answers from the folks who maintain and use memcached on a daily basis.

The most current Windows version of memcached (1.2.4) can be found at <http://www.splinedancer.com/memcached-win32/>. While it is a beta-release, it appears stable and is in production use in several places.

One critical component of memcached is libevent, the API that memcached uses for its event handling. The source code and more information can be found at <http://monkey.org/~provos/libevent/>

While you don't need to actually understand consistent hashing unless you plan on writing a client that interfaces directly with the memcached server rather than using one of the many available, it's still an interesting subject. You can check out <http://www.last.fm/user/RJ/journal/2007/04/10/392555/> to get a good general overview of the data or check out the links from that page to get into the nuts and bolts of the mathematics behind consistent hashing.

The following link is to the results of a study Jay Pipes did testing various caching mechanisms, including MySQL's cache compared to memcached: <http://www.mysqlperformanceblog.com/2006/08/09/cache-performance-comparison/>

Firewall/Network Configuration resources

How network security is implemented is dependent upon the hardware that makes up your network, but for a good general introduction to the subject check out <http://www.interhack.net/pubs/network-security/>.



Pragmatic Fridays

Timely and focused PDF-only books. Written by experts for people who need information in a hurry. No DRM restrictions. Free updates. Immediate download. Visit [our web site](#) to see what's happening on Friday!

More Online Goodness

Using memcached's Home Page

Source code from this book and other resources. Come give us feedback, too!

Free Updates

Visit the link, identify your book, and we'll create a new PDF containing the latest content.

Errata and Suggestions

See suggestions and known problems. Add your own. (The easiest way to report an errata is to click on the link at the bottom of the page.

Join the Community

Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy

Check out the latest pragmatic developments in the news.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragmaticprogrammer.com/catalog
Customer Service:	orders@pragmaticprogrammer.com
Non-English Versions:	translations@pragmaticprogrammer.com
Pragmatic Teaching:	academic@pragmaticprogrammer.com
Author Proposals:	proposals@pragmaticprogrammer.com