

Scaling PHP Applications

By Steve Corona

Scaling PHP

Steve Corona

This book is for sale at <http://leanpub.com/scalingphp>

This version was published on 2014-05-23



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

©2012 - 2014 Steve Corona

Contents

Preface	1
How this book came to be	1
How this book is designed	2
Who should read this book?	2
What you need for this book	3
What you'll learn	3
Why you need this book	3
Rule Numero Uno	3
Ditching the LAMP Stack	4
What's wrong with LAMP?	4
The Scalable Stack	4
DNS: Why you need to care	7
DNS Load Distribution	8
DNS Resolution	10
Load Balancing with HAProxy	14
HAProxy	14
Choosing the best hardware	20
Automatic failover with keepalived	21
Tuning linux for a network heavy load	22
Issues at scale	24
App Server: Horizontal Scaling with Nginx and PHP-FPM	26
Choosing the right version of PHP	26
Nginx and PHP-FPM	28
Choosing the best hardware for your application server	33
Using a PHP Opcode cache	34
Tuning Linux for PHP	37
Scaling session handling	40
Database Server: Ultimate MySQL Tuning Guide	46
Getting NoSQL Performance out of MySQL	46
Dealing with libmysql's inability to set timeouts	49
Tuning your MySQL Configuration	53
Tuning Linux for an intensive database	63
Load balancing MySQL Slaves	67

CONTENTS

Accepting the Master as a SPOF	72
Understanding issues with NUMA	75
Choosing the best hardware	78
Online Schema Changes	81
Further Topics	83
Cache Server: Using Redis	85
Choosing between Redis, Memcached, and APC	85
Redis Commands	88
The importance of Atomic Operations	91
Performance Limitations of Redis	92
Installing Redis from Dotdeb	93
Installing the <code>phpredis</code> C extension	93
Tuning Redis for Performance	94
Scaling Redis to Multiple Servers	97
“The Dogpile”	99
Russian Doll Caching	102
Redis Bitmaps	106
Redis Notification Feeds	108
Worker Server: Asynchronous Resque Workers	109
Why use Queues?	109
Getting started with Resque and <code>php-resque</code>	111
Writing a Resque Job	115
Alternatives to Resque	119
Things you should put in a queue	120
Coding and Debugging at Scale	121
Scaling your code	121
Capistrano for code deployment	125
Live Debugging PHP with <code>strace</code>	127
Parting Advice	131
Want to upgrade?	131
Sending Feedback and Testimonials	132
In too deep? Need help ASAP?	132
Updates?	132
About the Author	132
Case Study: Horizontally scaling user uploads	133
Case Study: How DNS took down Twitpic down for 4 hours	138
Case Study: Scaling MySQL Vertically (10 to 3 MySQL Servers w/ SSDs)	139
Case Study: Hot MySQL Backups w/ Percona	141
Moving from <code>mysqldump</code> to Percona XtraBackup	143
Case Study: Async APIs w/ Kestrel saved my butt	145

CONTENTS

Case Study: The missing guide to Memcached	148
Choosing Memcached over Redis?	148
Installing Memcached	149
Hooking up PHP to Memcached	150
Protocols	151
Compression and Serialization	151
Server Pools	152
Atomic Counters	156
Avoiding the Rat Race: CAS Tokens	157
Watch out for timeouts	159
Using Memcached for Sessions	160
Debugging Memcached from Telnet	161
My Memcached Setup	162
Further Reading and Tools	164
Case Study: HTTP Caching and the Nginx Fastcgi Cache	166
Caching Static Assets	167
Caching Proxy	169
Case Study: CakePHP Framework Caching	173
CakePHP, you devilish dessert	173
The life and death of a CakePHP request	174
If it smells like a race condition...	174
Busting the assumptions	175
Now what? I'm out of ideas	177
The solution(s)	178
Case Study: Optimizing image handling in PHP	181
The naive "PHP Tutorial" way	181
ImageMagick vs GraphicsMagick	189
Allowing Large File Uploads	191
Stripping EXIF Metadata	192
Autorotation	193
Case Study: Benchmarking and Load Testing Web Services	196
Setting up a Test Environment	196
The famous ab (Apache Benchmark)	196
Siege, a more modern ab	198
Bees with machine guns	199
Sysbench	201
Sponsors	204

Preface

How this book came to be

In 2009, I met Noah Everett, founder and CEO of Twitpic through a post he made on Twitter. Noah is a smart, self-taught programmer, quite literally thrown into scaling when Twitpic launched into overnight success after the first picture of the “Miracle on the Hudson”, [US Airways Flight 1549](#)¹ was posted on [Twitpic](#)².

Subsequently, our first conversation was less of an “interview” and more me introducing Noah to memcache and how it could be setup within the next hour—up till that point the site crashed regularly, and was down during our phone call. I wasn’t there with him at the time, but I like to think he was sweating bullets, feverishly typing `apt-get install memcache` while hundreds of angry beiber-fans were sending negative energy (and tweets) his way. An hour of free consulting for a life changing opportunity to own the tech behind one of the top 100 sites on the internet? Sounds good to me. It was a great time to be at Twitpic.



There's a plane in the Hudson. I'm on the ferry going to pick up the people. Crazy. - Janis Krums (@jkrums)

When I started at Twitpic, I quickly learned how big of a mess it was. The infrastructure was a mix of different servers and FreeBSD versions, backed up by a steaming pile of PHP-inlined HTML. Remember, Twitpic was built in a Red Bull fueled weekend, a side project for Noah to share pictures with a couple of his friends. It wasn’t meant win the prettiest-code beauty pageant. But, wow, it was bad. Think of the type of PHP your 14 year old little-brother might write— spaghetti, `pagename.php` files, no MVC framework, and a handful of `include_once`’s at the top of each file.

¹http://en.wikipedia.org/wiki/US_Airways_Flight_1549

²<http://twitpic.com/135xa>

In the beginning, Noah and I had to schedule our lives around our laptops. Going to the grocery store? Bring the laptop. Headed out on a date? Don't forget the laptop.

"Hey, I know this date is going awesome, but I need to put it on pause and jump on my laptop really quick. Do you think they have free WiFi here?"

Twitpic was like a jealous girlfriend, waiting to ruin your day in a moments notice. I never knew when I'd have to pop into a Starbucks to restart Apache.

Mostly out of necessity, and because no one enjoys cuddling with their laptop at night, we learned. We scaled. We improved our infrastructure. By no means is our setup perfect, but we've learned an incredible amount, and after three years I've built a pretty solid architecture. What happens when servers go down in the middle of the night today? Nothing. I sleep peacefully through the night, because we've built a horizontal, replicated system that is designed for failure.

How this book is designed

This book, although PHP centric, can be applied to any language. Most of the tricks, techniques, and design patterns will apply cleanly to Ruby, Python, Java or really anything besides Visual Basic (ok, maybe even VB too).

Read it like a cookbook, or maybe even a choose your own adventure story— you don't need to read it in order, or cover to cover. Jump around to different chapters or sections, digging into the ones that pique your interest the most. Or, if you're like Noah, the one you need right now to get your site back online.

I've laid out the chapters so they move from the outside of the stack, level-by-level, subsequently getting deeper until we hit the code. Yea, the code is last. Why? Because it's usually the fastest part of your stack. I get it, you're not a sysadmin and you don't play one on TV either, but 80% of your scaling gains are to be had outside of your code and a little bit of Linux-fu can go a long way.

Who should read this book?

This book is designed for startups, entrepreneurs and smart people that love to hustle and build things the right way.

You should know PHP, your current stack and your way around Linux. The examples assume Ubuntu Server, but any linux distribution will get the job done. If you can use a package manager (`apt-get`, `yum`) and edit text files, you'll be able to implement most examples in this book.

My readers are smart and intelligent people that don't need their hands held. I explain complicated topics and provide thorough examples, but this is **NOT simply regurgitated documentation**. I've packed as much practical, real-world knowledge and case-studies that I could, but topics discussed aren't exhaustive. If you want the docs, you can grab them off Google, because there's nothing I hate more than buying a book that's filled with 150 pages of filler material.

What you need for this book

At a minimum, you need a PHP application that you want to learn how to scale. And you'll probably see the most benefit if you have a Linux server that you can test it with. Don't have one? I really like [Rackspace Cloud](http://www.rackspace.com/cloud/public/servers/)³ or [Amazon EC2](http://aws.amazon.com/ec2/)⁴ for testing because you can setup and tear down servers quickly, and launch multi-server test setups cheaply.

What you'll learn

You're going to get thrown into the depths of scaling everything from DNS to MySQL to Nginx. If you suddenly wake up one morning with 10 million new users, don't even sweat it— the material in this book has been proven to handle it with grace. Today, Twitpic has over 40 million users and is running the *exact* setup that I describe in the 10 chapters ahead, it's the book that I *wish* I had 4 years ago.

Put what you learn on your Resume. Seriously. If you're interviewing with startups, there's no doubt that being an expert in the topics discussed will help you land the job. Already get the job? Or, use some of the techniques to scale your site successfully? Shoot me an email! I really want to hear about it.

Why you need this book

Ok Steve, I get it, this book is filled with great topics but what if I don't need to scale yet?

System Admin's are a dying breed, and they're slowly converging with [DevOps](#)⁵. The future of web programming involves being able to setup your own Nginx server, being able to tune MySQL, and get your production server in order.

When would you rather master these techniques? When you're still building, still learning, and have the extra time? Or when you're scrambling around at 2am after you got some media coverage and have been down for the past 24 hours? Learning to scale is a cheap insurance policy for success.

Rule Numero Uno

There's first rule of *Scaling PHP Applications* is that **you will not run the examples in production without testing them first**. That's all. If you agree, turn the next page. If you don't agree, well, boo, but if you burn down your servers complain to me. Get yourself a cheap test server on [DigitalOcean](https://www.digitalocean.com/?refcode=c5806f4bb04f)⁶ or AWS. It's easy.

³<http://www.rackspace.com/cloud/public/servers/>

⁴<http://aws.amazon.com/ec2/>

⁵<http://en.wikipedia.org/wiki/DevOps>

⁶<https://www.digitalocean.com/?refcode=c5806f4bb04f>

Ditching the LAMP Stack

What's wrong with LAMP?

LAMP (Linux, Apache, MySQL, PHP) is the most popular web development stack in the world. It's robust, reliable and everyone knows how to use it. So, what's wrong with LAMP? Nothing. You can go really far on a single server with the default configurations. But what happens when you start to really push the envelope? When you have so much traffic or load that your server is running at full capacity?

You'll notice tearing at the seams, and in a pretty consistent fashion too. MySQL is always the first to go—I/O bound most of the time. Next up, Apache. Loading the entire PHP interpreter for each HTTP request isn't cheap, and Apache's memory footprint will prove it. If you haven't crashed yet, Linux itself will start to give up on you—all of those sane defaults that ship with your distribution just aren't designed for scale.

What can we possibly do to improve on this tried-and-true model? Well, the easiest thing is to get better hardware (scale vertically) and split the components up (scale horizontally). This will get you a little further, but there is a better way. Scale intelligently. Optimize, swapping pieces of your stack for better software, customize your defaults and build a stack that's reliable and fault tolerant. You want to spend your time building an amazing product, not babysitting servers.

The Scalable Stack

After lots of trial and error, I've found what I think is a generic, scalable stack. Let's call it **LHNMPRR**... nothing is going to be as catchy as LAMP!

Linux

We still have Old Reliable, but we're going to tune the hell out of it. This book assumes the latest version of Ubuntu Server 12.04, but most recent distributions of Linux should work equally as well. In some places you may need to substitute `apt-get` with your own package manager, but the kernel tweaks and overall concepts should apply cleanly to RHEL, Debian, and CentOS. I'll include kernel and software versions where applicable to help avoid any confusion.

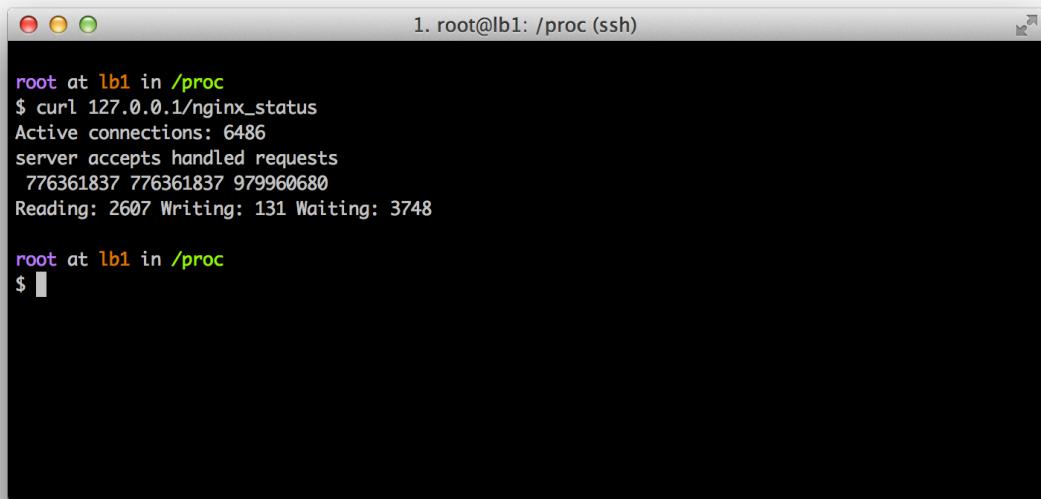
HAProxy

We've dumped Apache and split its job up. HAProxy acts as our load balancer—it's a great piece of software. Many people use nginx as a load balancer, but I've found that HAProxy is a better choice for the job. Reasons why will be discussed in-depth in Chapter 3.

nginx

Nginx is an incredible piece of software⁷. It brings the heat without any bloat and does web-serving extremely well. In addition to having an incredibly low memory and CPU footprint, it is extremely reliable and can take an ample amount of abuse without complaining.

One of our busiest nginx servers at Twitpic handles well over 6,000 connections per second while using only 80MB of memory.

A screenshot of a terminal window titled "1. root@lb1: /proc (ssh)". The window shows the output of the command "curl 127.0.0.1/nginx_status". The output displays the following statistics: Active connections: 6486, server accepts handled requests 776361837 776361837 979960680, Reading: 2607 Writing: 131 Waiting: 3748. The terminal prompt is "\$".

```
root at lb1 in /proc
$ curl 127.0.0.1/nginx_status
Active connections: 6486
server accepts handled requests
776361837 776361837 979960680
Reading: 2607 Writing: 131 Waiting: 3748

root at lb1 in /proc
$
```

An example of a server with over 6000 active connections

PHP 5.5 / PHP-FPM

There are several ways to serve PHP applications: mod_php, cgi, lighttpd fastcgi??. None of these solutions come close to PHP-FPM, a FastCGI Process Manager written by the nginx team that's been bundled with PHP since 5.3. What makes PHP-FPM so awesome? Well, in addition to being rock-solid, it's extremely tunable, provides real-time stats and logs slow requests so you can track and analyze slow portions of your codebase.

MySQL

Most folks coming from a LAMP stack are going to be pretty familiar with MySQL, and I will cover it pretty extensively. For instance, in Chapter 5 you'll learn how you can get NoSQL performance out of MySQL. That being said, this book is database agnostic, and most of the tips can be similarly applied to any database. There are many great databases to choose from: Postgres, MongoDB, Cassandra, and Riak to name a few. Picking the correct one for your use case is outside the scope of this book.

⁷Apache is great, too. It's extremely popular and well supported, but Apache + mod_php is not the best tool for the job when you're dealing with high volume PHP apps. It's memory intensive and doesn't offer full tuning capabilities.

Redis

Redis⁸ can be used as a standalone database, but its primary strength is as a datatype storage system. Think of it as memcache on steroids. You can use memcache (we still do), but Redis performance is just as good or better in most scenarios. It is persistent to disk (so you don't lose your cache in the case of a crash, which can be catastrophic if your infrastructure can't deal with 100% cache misses), and is updated extremely frequently by an opinionated, vocal and smart developer, @antirez.

Resque

Doing work in the background is one of the principal concepts of scaling. Resque⁹ is one of the best worker/job platforms available. It uses Redis as a backend so it's inherently fast and scalable, includes a beautiful frontend to give you full visibility into the job queue, and is designed to fail gracefully. Later we'll talk about using PHP-Resque, along with some patches I've provided, to build elegant background workers.

Should I run on AWS?

AWS is awesome. Twitpic is a huge AWS customer—over 1PB of our images are stored on Amazon S3 and served using CloudFront. Building that kind of infrastructure on our own would be extremely expensive and involve hiring an entire team to manage it.

I used to say that running on EC2 full-time was a bad idea—that it was like using your expensive iPhone to hammer a nail into the wall, it works and it'll get the job done, but it's going to cost you much more than using a hammer.

I don't think this is the case any longer. There is a certainly a premium to using AWS, but it's much cheaper than it used to be and the offering has become much better (200GB+ memory, SSDs, provisioned IOPS). If I was starting up a new company, I would build it on AWS, hands down. Reserved Instances will get your monthly cost down to something that's reminiscent of Dedicated Server Hosting (albeit still 10-20% more expensive), but being able to bring up a cluster of massive servers instantly or start up a test cluster in California in less than 5 minutes is powerful.

For a bigger, more established startup, I would maintain a fleet of dedicated servers at SoftLayer or Rackspace, and use EC2 for excess capacity, emergencies, and on-demand scaling. You can do this seamlessly with a little bit of network-fu and Amazon VPC. Sudden increase in traffic? Spin up some EC2 instances and handle the burst effortlessly. Using Amazon VPC, your EC2 instances can communicate with your bare-metal servers without being exposed to the public internet.

⁸<http://redis.io/>

⁹<https://github.com/defunkt/resque>

DNS: Why you need to care

The first layer that we are going to unravel is DNS. DNS? What!? I thought this was a book on PHP? DNS is one of those things that we don't really think about until it's too late, because when it fails, it fails in the worst ways.

Don't believe me? In 2009 Twitter's DNS was hijacked and redirected users to a hacker's website for an hour. That same year, SoftLayer's DNS system was hit with a massive DDoS attack and took down their DNS servers for more than six-hours. As a big SoftLayer customer, we dealt with this firsthand because (at the time) we also used their DNS servers.

The problem with DNS downtime is that it provides the worst user experience possible—users receive a generic error, page timeouts, and have no way to contact you. It's as if you don't exist anymore, and most users won't understand (or likely care) why.

As recently as September 2012, GoDaddy's DNS servers were attacked and became unreachable for over 24-hours. The worst part? Their site was down too, so you couldn't move your DNS servers until their website came back up (24-hours later).

Too many companies are using their domain registrar or hosting provider's DNS configuration—that is **WRONG!** Want to know how many of the top 1000 sites use GoDaddy's DNS? None of them.

So, should I run my own DNS server?

It's certainly an option, but I don't recommend it. Hosting DNS "the right way" involves having many geographically dispersed servers and an Anycast network. DDoS attacks on DNS are extremely easy to launch and if you half-ass it, your DNS servers will be the Achilles' heel to your infrastructure. You could have the best NoSQL database in the world but it won't matter if people can't resolve your domain.

Almost all of the largest websites use an external DNS provider. That speaks volumes as far as I'm concerned—the easiest way to learn how to do something is to imitate those that are successful.

- Reddit: [Akamai](http://www.akamai.com/html/solutions/enhanced_dns.html)¹⁰
- Twitter: [Dynect](http://dyn.com)¹¹
- Amazon: [UltraDNS](http://ultradns.com)¹² and Dynect
- LinkedIn: UltraDNS

¹⁰http://www.akamai.com/html/solutions/enhanced_dns.html

¹¹<http://dyn.com>

¹²<http://ultradns.com>

You should plan to pay for a well known, robust DNS SaaS. At Twitpic, we use [Dynect Managed DNS](#)¹³. It has paid for itself—we've had no downtime related to DNS outages since switching. Make sure you choose a DNS provider that has a presence close to your target audience, too, especially if you have a large international userbase.

Here's what you should look for in a DNS provider:

- Geographically dispersed servers
- Anycast Network
- Large IP Space (to handle network failures)



What is Anycast?

Anycast is a networking technique that allows multiple routers to advertise the same IP prefix—it routes your clients to the “closest” and “best” servers for their location. Think of it as load balancing at the network level.

In addition to the providers listed above, [Amazon Route53](#)¹⁴ is a popular option that is incredibly cheap, offers a 100% SLA, and has a very user-friendly web interface. Since it's pay-as-you-go, you can easily get started and adjust for exact usage needs as you grow.

Name	Type	Value	TTL
scalingphpbook.com.	A	174.129.25.170	300
scalingphpbook.com.	NS	ns-1846.awsdns-38.co.uk. ns-540.awsdns-03.net. ns-1490.awsdns-58.org. ns-158.awsdns-19.com.	172800
scalingphpbook.com.	SOA	ns-1846.awsdns-38.co.uk. awsd	900
www.scalingphpbook.co	CNAME	d3dzhg6nialwmr.cloudfront.net	300

DNS Load Distribution

Besides scaling DNS, we can use DNS to help us scale, too. The HAProxy load balancer is integral to our scalable stack—but what happens when it loses network connection, becomes overloaded, or just plain crashes? Well it becomes a single point of failure, which equals downtime—it's a question of when, not if it will happen.

Traditional DNS load balancing involves creating multiple A records for a single host and passing all of them back to the client, letting the client decide which IP address to use. It looks something like this.

¹³<http://dyn.com/dns/>

¹⁴<http://aws.amazon.com/route53/>

```

1  > dig A example.com
2  ; <>> DiG 9.7.3-P3 <>> A example.com
3
4  ;; QUESTION SECTION:
5  ;example.com.           IN      A
6
7  ;; ANSWER SECTION:
8  example.com.          287     IN      A      208.0.113.36
9  example.com.          287     IN      A      208.0.113.34
10 example.com.          287     IN      A      208.0.113.38
11 example.com.          287     IN      A      208.0.113.37
12 example.com.          287     IN      A      208.0.113.35

```

There are a few drawbacks to this, however. First of all, less-intelligent DNS clients will always use the first IP address presented, no matter what. Some DNS providers (Dynect and Route53, for example) overcome this by using a round-robin approach whereby they change the order of the IPs returned everytime the record is requested, helping to distribute the load in a more linear fashion.

Another drawback is that round-robin won't prevent against failure, it simply mitigates it. If one of your servers crashes, the unresponsive server's IP is still in the DNS response. There are two solutions that work together to solve this.

1. Use a smart DNS provider that can perform health checks. Dynect offers this feature and it's possible to implement it yourself on Route53 using their API. If a load balancer stops responding or becomes unhealthy, it gets removed from the IP pool (and readded once it becomes healthy again).
2. Even if you remove a server's IP from the DNS pool, users that have the bad IP cached will still experience downtime until the record's TTL expires. Anywhere from 60-300s is a recommended TTL value, which is acceptable, but nowhere near ideal. We'll talk about how servers can "steal" IPs from unhealthy peers using `keepalived` in Chapter 4.

Dynect has a very intuitive interface for load balancing and performing health checks on your hosts:

Service Statuses		
	Serve Count: 1	Status: ok ✓
Address	Health Status	Health Results
LB1 (50.23.200.230)	█ ↑	█ ↑ ↑ ↑
LB2 (50.23.200.238)	█ ↑	█ ↑ ↑ ↑
LB3 (50.23.200.239)	█ ↑	█ ↑ ↑ ↑

DNS Resolution

The last, and very often overlooked, part of scaling DNS is internal domain resolution. An example of this is when a PHP application calls an external API and has to resolve the domain of the API host. Let's say the application is using the Twitter API—every time you post something to Twitter, PHP has to look up and determine the IP of `api.twitter.com`.

PHP accomplishes this by using the `libc` system call `gethostbyname()`, which uses nameservers set in `/etc/resolv.conf` to look up the IP address. Usually this is going to be set to a public DNS resolver (like `8.8.8.8`) or a local resolver hosted by your datacenter.

So, what's wrong with this setup? Well, two things:

1. It's another server that you don't control. What happens when it's down? Slow? They block you? The lowest timeout allowed in `/etc/resolv.conf` is one second (and the default is FIVE!), which is too slow for a high-volume website and can cause domino-effect failure at scale.
2. Most Linux distributions don't provide a DNS cache by default and that adds extra network latency to every single DNS lookup that your application has to make.

The solution is to run a DNS cache daemon like `nsqd`, `dnsmasq`, or `bind`. I won't cover `BIND` because it's overkill to run it simply as a cache, but I will talk about `nsqd` and `dnsmasq`, which work in slightly different ways.

`nsqd`

`nsqd` (nameserver cache daemon) is the simplest solution to setting up your own internal DNS cache. Whether or not it's installed by default is dependent on your Linux distro (it's not on

Ubuntu or Debian). It's easy to install and needs zero-configuration. The main difference between `nscd` and `dnsmasq` is that `nscd` runs locally on each system while `dnsmasq` is exposed as a network service and can provide a shared DNS cache for multiple servers.

```
1 > apt-get install nscd
2 > service nscd start
```

Pros

- Extremely easy to setup
- Zero configuration

Cons

- Runs locally only, so each server needs to have its own install

dnsmasq

`dnsmasq` is a lightweight DNS cache server that provides nearly the same feature-set as `nscd`, but as a network service.

You setup `dnsmasq` on a server, let's call it 198.51.100.10, and set 198.51.100.10 as the nameserver for all of your other servers. `dnsmasq` will still go out onto the internet to look up DNS queries for the first time, but it will cache the result in memory for subsequent requests, speeding up DNS resolution and allowing you to gracefully deal with failure.

Additionally, you can use `dnsmasq` as a lightweight internal DNS server with the `addn-hosts` configuration option, allowing you to use local hostnames without having to hardcode IP addresses in your code (i.e. `$memcache->connect('cache01.example')` instead of `$memcache->connect('198.51.100.15')`).

Assuming a network setup based on the table below, here is how we'd setup `dnsmasq` and point our servers to it:

Hosts	IP
dnsmasq server	192.51.100.10
server01.example	192.51.100.16
server02.example	192.51.100.17

On your `dnsmasq` server:

```

1 > apt-get install dnsmasq
2 > vi /etc/dnsmasq.conf
3
4 cache-size=1000
5 listen-address=198.51.100.10
6 local-ttl=60
7 no-dhcp-interface=eth0
8 no-dhcp-interface=eth1
9 addn-hosts=/etc/dnsmasq.hosts
10
11 > vi /etc/dnsmasq.hosts
12
13 198.51.100.16 server01.example
14 198.51.100.17 server02.example
15
16 > service dnsmasq restart

```

local-ttl sets the time-to-live for any hosts you define in /etc/hosts or /etc/dnsmasq.hosts
cache-size defines the size of the DNS cache.

no-dhcp-interface disables all services provided by dnsmasq except for dns. Without this, dnsmasq will provide dhcp and tftp as well, which you do not want in most scenarios.

On EC2, after restarting dnsmasq you may need to add the following line to /etc/hosts:

```
1 127.0.0.1 ip-10-x-x-x
```

And then on your other hosts:

```

1 > vi /etc/resolv.conf
2
3 nameserver 198.51.100.10
4 options rotate timeout:1

```

The rotate option tells linux to rotate through the nameservers instead of always using the first one. This is useful if you have more than one nameserver and want to distribute the load.

The timeout:1 option tells linux that it should try the next nameserver if it takes longer than 1-second to respond. You can set it to any integer between 1 and 30. The default is 5-seconds and it's capped at 30-seconds. Unfortunately, the minimum value is 1-second, it would be beneficial to set the timeout in milliseconds.

When do I need this?

There is virtually no negative impact to implementing a caching nameserver early; however, it does add another service to monitor—it's not a completely free optimization. Essentially, if any of your pages require a DNS resolution, you should consider implementing a DNS cache early.

Want to know how many uncached DNS requests your server is currently sending? With tcpdump you can see all of the DNS requests going out over the network.

```
1 > apt-get install tcpdump
2 > tcpdump -nnp dst port 53
3
4 09:52 IP 198.51.100.16 > 198.51.100.10.53: A? graph.facebook.com.
5 09:52 IP 198.51.100.16 > 198.51.100.10.53: AAAA? graph.facebook.com.
6 09:52 IP 198.51.100.16 > 198.51.100.10.53: A? api.twitter.com.
7 09:52 IP 198.51.100.16 > 198.51.100.10.53: AAAA? api.twitter.com.
```

The `-nn` option ensures `tcpdump` itself does not resolve IP Addresses or protocols to names.

The `-p` option disables promiscuous mode, to minimize adverse impact to running services.

The `dst port 53` only shows DNS requests sent (for brevity), to see how long the application may have blocked waiting for a response (i.e. to see the request along with the response), exclude the 'dst' portion.

The above example shows how a single HTTP GET could cause four DNS requests to be sent to 3rd party APIs that you may be using. Using a caching DNS resolver, such as `dnsmasq` or `nsqd`, helps reduce the blocking period and possible cascading failures.

Load Balancing with HAProxy

The load balancer is the nervous system of your application, it takes all of the incoming requests and fairly distributes them to your application servers where your PHP code is executed. The configuration we're going to talk about is a called a **reverse proxy**. It's fairly common, but I'm going to show you a few new tricks that you may not have seen before.

One note before we get started—remember that all *incoming* AND *outgoing* data goes through the load balancer. The app servers never speak directly to your users, they only respond back to the load balancer, which then responds back to your users.

There are many load balancer options to choose from, both software and hardware. I don't recommend hardware load balancers for a few reasons: 1) they are extremely expensive (easily over \$40,000—not exactly budget friendly for a startup, especially when you could use open-source software and put that money toward the salary for another engineer); and 2) hardware load balancers tend to be “black boxes,” they don't provide you with the same kind of visibility that you'd get with open-source software. Remember—own your stack!

It's smart to have multiple load balancers, because just one would create a single point of failure. You need to have enough spares to handle all of the traffic when a single load balancer fails. For instance, if you have enough traffic to require two load balancers, you'll need to have a third hot-spare ready for action; otherwise, if one of the primary two fail, the remaining load balancer will not be able to handle all of the traffic on its own. And don't sweat, we will also cover how to setup the necessary DNS records to split traffic between multiple load balancers in this chapter.

HAProxy

HAProxy¹⁵ is a great solution for software load balancing. It's written in C, event-based and built for the sole purpose of running a reverse proxy, so it's extremely fast and lightweight. HAProxy has a small memory and CPU footprint, meaning it's able to squeeze a lot of traffic through low-powered hardware. HAProxy's website shows some benchmarks from 2009 where they were able to hit 108,000 HTTP requests per second and fully saturate a 10GbE network, which is pretty impressive if you ask me.

Additionally, HAProxy boasts an impressive record of being “show-stopper” bug-free since 2002. The quote below is from their website.

In single-process programs, you have no right to fail: the smallest bug will either crash your program, make it spin like mad, or freeze. There has not been any such bug found in the code nor in production for the last 10 years.

¹⁵<http://haproxy.1wt.eu/>

Advantages over Nginx

Nginx¹⁶ is no doubt an awesome piece of software and even includes its own reverse proxy module, making it able to run as a load balancer. Even though nginx makes a decent load balancer in its own right, HAProxy is an even better choice for a couple of reasons noted below.

HAProxy is a very bare-metal piece of software. It does one thing and it does it extremely well, which can be a significant benefit or a huge shortcoming depending on your load balancing needs. For most stacks, HAProxy will be a great fit, but if you need more flexibility, you shouldn't hesitate to use nginx. While as a load balancer it's not quite as good as HAProxy, it can still come close in exchange for a slightly higher CPU and Memory footprint.

Layer 4 vs Layer 7

HAProxy can run in two modes, TCP (Layer 4) and HTTP (Layer 7). In TCP mode, HAProxy simply forwards along the raw TCP packets from the clients to the application servers. This uses about 50% less CPU than HTTP (Layer 7) mode, which involves parsing the HTTP headers before forwarding them to the application servers. Since nginx is primarily a webserver, it only supports HTTP (Layer 7) load balancing.

You'll surely read plenty of opinions saying "it doesn't matter"—Layer 7, double-HTTP parsing, is a small overhead to pay unless you're massive. There's some truth to that, but like I said, let's keep things simple.

One gotcha with TCP load balancing— it's possible to lose the IP Address of the client. Consider this scenario: Client 192.168.25.19 hits your load balancer (which has the IP 10.0.9.1). The load balancer forwards this request onto your nginx servers, but when your PHP code looks at `$_SERVER['REMOTE_ADDR']` it sees 10.0.9.1.. NOT 192.168.25.19.

No problem you think, a lot of load balancers put the "real ip" of the client into a different header like `$_SERVER['X-Forwarded-For']`, so you check that. No dice. Nothing. What's going on? Well, since the TCP packet gets re-sent from the load balancer it obviously takes on the IP Address of the load balancer. Since we're talking about Layer 4 Load Balancing, it's never parsing and rebuilding the HTTP Headers, which means it can never inject a header like `X-Forwarded-For` into the HTTP Request.

Bummer! Are we out of options!? No! Luckily there is a very elegant solution that was created by the guys at HAProxy. They've created their own [PROXY Protocol] that let's you attach some proxy information (like the `REMOTE_ADDR`) at the TCP layer.

The PROXY protocol -only- works with software that specifically includes support for it. Luckily, nginx does just that. In fact, it's incredibly simple.. just add `real_ip_header proxy_protocol` to your nginx config (discussed in the next chapter) and you're good to go. Bam.

SSL Termination

Another benefit that comes along for the ride with TCP (Layer 4) load balancing is delegating SSL termination to the application server. What this means is that the SSL negotiation/decryption

¹⁶<http://nginx.org/>

process of an HTTPS request happens on your application servers, inside of your webserver, since the raw TCP packet is just forwarded along.

With a HTTP (Layer 7) load balancer such as nginx, it wouldn't be able to parse the HTTP headers without first handling the SSL, forcing all SSL negotiation to happen on the load balancer. The downside is that SSL is expensive on the CPU—it's better to spread the usage across your entire application cluster instead of centralizing it on your one or two load balancers.

Better health checks and distribution algorithms

When nginx is setup as a load balancer, the only method of performing health-checks on backend servers is by using a timeout. If an application server times out, it's removed from the pool—otherwise, it's considered healthy, even if it's spitting out jibberish/php code/500 errors. HAProxy provides several better methods of performing health checks on your cluster, which we'll cover below.

Additionally, nginx is limited to a single naive round-robin algorithm for choosing backend servers. HAProxy has several balancing methods, including one that sends new connections to the server with the least amount of existing connections. We'll discuss the different algorithms available in the section below as well.

Installation

```
1 > apt-get install haproxy
2 > vi /etc/haproxy/haproxy.cfg
3
4     global
5         maxconn 50000
6         user haproxy
7         group haproxy
8         stats socket /tmp/haproxy
9         node lb1
10        nbproc 1
11        daemon
12
13     defaults
14     log     global
15     retries 3
16     option  dontlog-normal
17     option  splice-auto
18     timeout connect 5000ms
19     timeout client  5000ms
20     timeout server  5000ms
21     maxconn 50000
22
23     backend application_servers
24     mode      tcp
```

```

25    balance roundrobin
26    option httpchk HEAD / HTTP/1.1\r\nHost: \ example.com
27    option ssl-hello-chk
28    server web02 198.51.100.17:80 check
29    server web03 198.51.100.18:80 check
30
31    frontend example.com
32    bind *:80
33    bind *:443
34    mode tcp
35    default_backend application_servers
36
37    listen stats *:1936
38    mode http
39    stats enable
40    stats uri /
41    stats hide-version
42    stats auth Username:Password

```

HAProxy won't run until you enable the init.d script, so let's do that now.

```

1 > vi /etc/default/haproxy
2   # Set ENABLED to 1 if you want the init script to start haproxy.
3   ENABLED=1
4
5 > service haproxy start

```

Configuration options

Most of the configuration options in `haproxy.cfg` are pretty self-explanatory, but let's go over the most important ones.

`maxconn` - The maximum number of connections that HAProxy will accept at a given time. This should be set as high as possible in most cases. Each connection held by HAProxy uses 33KB of memory, so it's important to make sure that you have enough memory to support whatever value you use (i.e, on a 4GB system, about 127,000 connections).

`timeout connect/client/server` - Timeouts for different connections that are made during the lifecycle of a request, set in milliseconds.

`options dontlognormal` - Only log errors, timeouts and other issues. Used to separate the noise out, especially when dealing with high-volume web apps. Remember, you can still log normal requests in the nginx access log on the application servers.

`options httpchk` - Replace `example.com` with your domain. This checks that the application server returns a 200 status code, used to determine that your application servers are not only up, but *healthy* too.

option splice-auto - Enables TCP splicing if it thinks that it'll improve performance. TCP splicing is a performance enhancement in the Linux Kernel which speeds up packet handling when proxying TCP connections. **Important** You must be using a kernel $\geq 2.6.29$ for this feature.

Balancing Algorithms

HAProxy has several balancing algorithms that you can choose from. These algorithms determine the behavior that's used to pick the backend application server to handle the incoming request.

roundrobin - The simplest. For each new connection, the next backend in the list is used. When the end of the list is reached, it will cycle back to the top. It distributes fairly, but does not take into consideration load or amount of connections.

leastconn - New connections go to servers with the least amount of existing connections. Useful for situations where your average request time/load varies significantly, or if you have long-polling connections.

source - Also known as **sticky sessions**. The IP of the client is hashed and used to determine the backend server. For example, a request from 203.0.113.100 will always be routed to appserver01 and one from 203.0.113.200 will always go to appserver02. Can be useful for A/B testing (by running different versions of the application on each appserver) or sharding users; however, the source algorithm is usually a bad long-term strategy and indicates a broken scaling strategy.

uri - Similar to the source algorithm except instead of hashing based on the client's IP address, it uses the URL requested. This is great for running HTTP proxies because example.com/picture01.jpg will always be routed to backend01 and example.com/picture02.jpg will always be routed to backend02, allowing you to maximize each backend server's filesystem and in-memory cache. Typically, this should be used for serving static assets (images, css, js).



Sticky sessions (via the source algorithm) seem like a good idea at first, but it's almost always a bad choice long-term. Having your users always routed to the same application servers can create hotspots and coldspots, depending on which users are currently active. In my experience, sticky sessions are used because of another area that hasn't been scaled—for example, sessions being stored on the filesystem, tying each user to a particular application server. In other words, it gets the job done but it's sloppy! Avoid the temptation and scale out the right way, it'll pay off in the long run.

Web-UI

HAProxy comes with a helpful web-ui that provides some insights into your load balancer, as well as making general stats available. It's a quick way to see essential information such as number of connections, which application servers are up or down, number of errors and more.

Using the `haproxy.cfg` configuration provided above, the web-ui listens on port 1936 and is password protected with the username "Username" and the password "Password". No doubt, you'll probably want to change that (it's defined by this line: `stats auth Username:Password`).

If you load up `your-load-balancer.com:1936`, you'll be presented with a web-ui similar to the image below. Also worth noting, if you append `;csv` to the URL, it'll return the data in a comma-separated format, making it easy to programmatically tie the data into your monitoring system.

HAProxy version 1.4.18, released 2011/09/16

Statistics Report for pid 6338

> General process information

pid = 6338 (process #1, nbproc = 1)
 uptime = 1d 17h05m45s
 system limits: memmax = unlimited; ulimit-n = 8207
 maxsock = 8207; maxconn = 4096; maxpipes = 0
 current conn = 1; current pipes = 0/0
 current tasks: 1/4

Running tasks: 1/4

Note: UP with load-balancing disabled is reported as "NOLB".

Display option:

- Hide "DOWN" servers
- Refresh now
- CSV export

External resources:

- Primary site
- Updates (v1.4)
- Online manual

Legend:

- active UP
- active UP, going down
- active DOWN, going up
- active or backup DOWN
- not checked
- active or backup DOWN for maintenance (MAINT)

webcluster

	Queue		Session rate		Sessions			Bytes		Denied		Errors		Warnings		Status	Server									
	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req	Conn	Resp		Retr	Redis	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime	Thrtile
Frontend	0	4	-	0	6	2 000	13	14 353	359 975	0	0	0	0	0	0	0	OPEN									
web01	0	0	-	0	2	0	3	-	7	7	8 248	197 102	0	0	0	0	0	1d17h UP	L4OK in 36ms	1	Y	-	0	0	0s	-
web02	0	0	-	0	2	0	3	-	6	6	8 105	162 873	0	0	0	0	0	1d17h UP	L4OK in 36ms	1	Y	-	4	0	0s	-
web03	0	0	-	0	0	0	0	-	0	0	0	0	0	0	0	0	1d17h DOWN	L4TOOUT in 2001ms	1	Y	-	0	1	1d17h	-	
Backend	0	0	-	0	4	0	6	2 000	13	13	14 353	359 975	0	0	0	0	0	1d17h UP		2	2	0	0	0	0s	

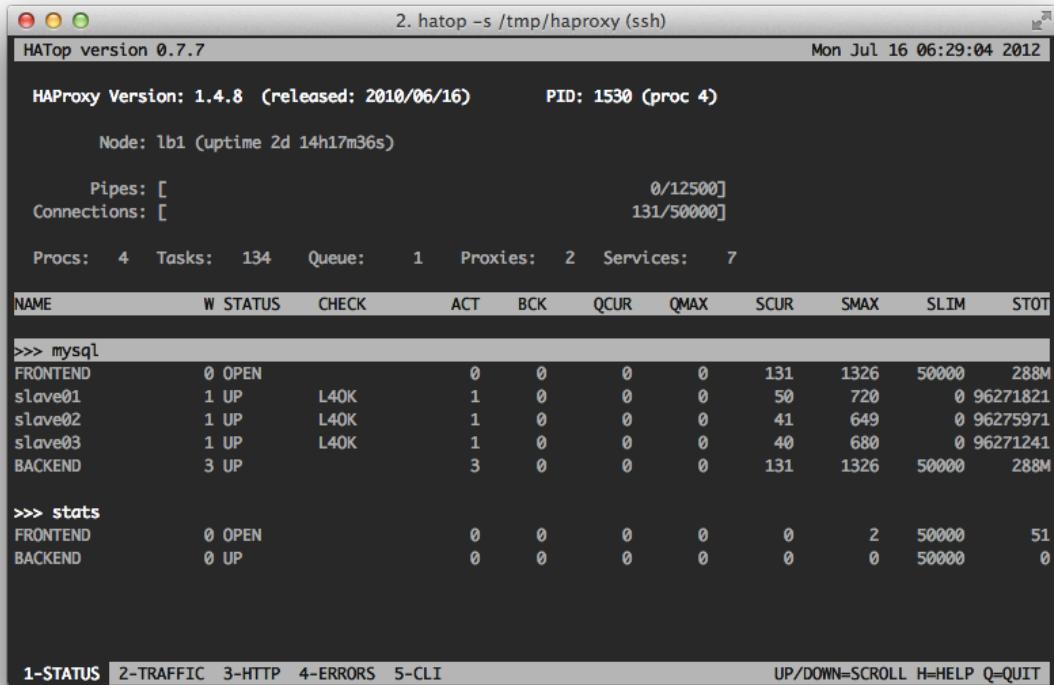
status

	Queue		Session rate		Sessions			Bytes		Denied		Errors		Warnings		Status	Server								
	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	In	Out	Req	Resp	Req	Conn	Resp		Retr	Redis	LastChk	Wght	Act	Bck	Chk	Dwn	Downtime
Frontend	0	2	-	1	2	2 000	22	13 284	220 934	0	0	0	0	0	0	0	OPEN								
Backend	0	0	-	0	0	0	2 000	0	0	13 284	220 934	0	0	0	0	0	1d17h UP		0	0	0	0	0	0	

CLI access with hatop

There's also a CLI monitor called `hatop` that's available. It provides the same data in the web interface as well as the ability to remove and add servers from the pool.

```
1 > apt-get install hatop
2 > hatop -s /tmp/haproxy
```



Choosing the best hardware

You'll see the most gains with a high frequency, modern CPU and a moderate amount of memory. It's tempting to choose an older CPU with more GHz, but the newer, lower-clocked Intel architectures are truly more powerful. For example, our standard build uses a single quad-core SandyBridge Xeon CPU, which we found to perform 2-3x better than a Dual Processor, Quad-Core build with an older architecture.

When in doubt, always consult [Passmark CPU Benchmark](http://www.cpubenchmark.net/)¹⁷, GHz and number of cores mean nothing without the context of CPU architecture.

HAProxy is usually run in single-process mode (configuration setting: `nbproc 1`), which means that it can only run on a single CPU core. What this means is that a faster CPU, not more cores, will directly correlate to the speed of your load balancer.



Don't let the "single-process" thing scare you, though. Since HAProxy is event-based, it's extremely well suited for an I/O bound (network, in this case) workload, and can easily handle 20-30,000 connections per second. It is possible to increase throughput by running HAProxy on multiple cores (simply increase the `nbproc` setting), but it's discouraged unless you truly need the capacity.

Our standard load balancer build is below for reference:

¹⁷<http://www.cpubenchmark.net/>

Intel Xeon 1270 (SandyBridge) 3.4GHz, Quad-Core 8GB ECC DDR3 Memory 500GB Hard Drive
2x 1gbps ethernet

Automatic failover with keepalived

keepalived is a daemon used in high-availability setups which monitors another server and steals its IP address if it stops responding to network requests. When you have it configured, it feels almost magical because it works so seemlessly.

We use keepalived to provide fault-tolerance in our load balancer setup and prevent against a single point of failure by keeping a hot-spare that can jump in if necessary.

Hosts	IP
virtual ip	192.51.100.40
lb.example	192.51.100.20
lbspare.example	192.51.100.30

In this configuration, we have two load balancers, each with their own IP address, and a virtual IP (vip) that floats between `lb.example` and `lbspare.example`. Both load balancers should have HAProxy configured already.

First off, we need to make a configuration change to the linux kernel on both servers by changing an obscure option, `net.ipv4.ip_nonlocal_bind`, which tells the kernel it's okay if services bind to non-existent IP addresses. It's needed because HAProxy on `lbspare.example` will listen on `192.51.100.40`, even when that IP isn't bound to the server.

```

1 > sysctl -w net.ipv4.ip_nonlocal_bind=1
2 > vi /etc/sysctl.d/keepalive.conf
3     net.ipv4.ip_nonlocal_bind=1

```

Next, we install keepalived on both servers.

```
1 > apt-get install keepalived
```

Setup the `keepalived.conf` file on `lb.example`

```

1 > vi /etc/keepalived/keepalived.conf
2     vrrp_instance VI_1 {
3         interface eth0
4         state MASTER
5         virtual_router_id 51
6         priority 101
7         virtual_ipaddress {
8             192.51.100.40
9         }
10    }
11 > service keepalived start

```

Lastly, setup the `keepalived.conf` file on `1bspare.example`

```

1 > vi /etc/keepalived/keepalived.conf
2     vrrp_instance VI_1 {
3         interface eth0
4         state MASTER
5         virtual_router_id 51
6         priority 100
7         virtual_ipaddress {
8             192.51.100.40
9         }
10    }
11 > service keepalived start

```

Notice the main difference between those two files, `1bspare.example` has a lower priority than `1b.example`. With the priority setting, you're not limited to only a single hot-failover—you could have an entire chain of them!

Next, verify that you can ping the vip (192.51.100.40) and that it routes to `1b.example`.

At this point, you can modify `haproxy.conf` file on both load balancers to listen on the vip. HAProxy can bind to multiple IP addresses by simply giving a comma separated list to the `bind` option, so it's possible to listen on both the server's ip and the vip by using the `bind 192.51.100.20:80, 192.51.100.40:80` configuration setting.

How does it all work?

The `keepalived` daemon on `1bspare.example` will monitor the network and verify that `1b.example` is continuously announcing itself on the network. If `1b.example` stops announcing itself (reboot, crash, etc.), `1bspare.example` will send a gratuitous ARP message to the network, letting everyone know that `1bspare.example` is now the owner of the vip. The hosts on the network will update their routing tables and services continuously, almost undisturbed.

When `1b.example` comes back online, it will announce itself to the network and resume control of the vip. The best part? It all happens automatically without any interaction from you.

Tuning linux for a network heavy load

`net.core.somaxconn`

`somaxconn` defines the size of the kernel queue for accepting new connections. It's usually only set to 128, which is too low and means you can only, at most, serve 128 concurrent users. Bump it up.

```

1 > sysctl -w net.core.somaxconn=100000
2 > vi /etc/sysctl.d/haproxy-tuning.conf
3     net.core.somaxconn=100000

```

net.ipv4.ip_local_port_range

ip_local_port_range defines the range of usable ports on your system. On my stock ubuntu installation, it's set to 32768-61000. Increase the range to allow for more connections. The number of available ports limits the number of simultaneous open connections. Remember, even after a connection is closed it still eats a port in the TIME_WAIT state (though we mitigate this with some settings below).

```

1 > sysctl -w net.ipv4.ip_local_port_range="10000 65535"
2 > vi /etc/sysctl.d/haproxy-tuning.conf
3     net.ipv4.ip_local_port_range=10000 65535

```

net.ipv4.tcp_tw_reuse

Part of the TCP protocol is the TIME_WAIT state, which keeps the socket open for up to 4 minutes after the connection has been closed. On a busy server, this can cause issues with running out of ports/sockets. The net.ipv4.tcp_tw_reuse tells the kernel that it's okay to reuse TCP sockets when it's safe to do so, without waiting for the full timeout to happen. Additionally, net.ipv4.tcp_recycle and net.ipv4.tcp_fin_timeout can all be tweaked, but you should be leary to mess with either of these.

```

1 > sysctl -w net.ipv4.tcp_tw_reuse=1
2 > sysctl -w net.ipv4.netfilter.ip_conntrack_tcp_timeout_time_wait=1
3 > vi /etc/sysctl.d/haproxy-tuning.conf
4     net.ipv4.tcp_tw_reuse=1
5     net.ipv4.netfilter.ip_conntrack_tcp_timeout_time_wait=1

```

ulimit -n 999999

On a stock linux configuration, the maximum number of open files allowed per process is set very low (1024). Since sockets are considered files on a linux system, this limits the number of concurrent connections as well. Remember, each connection through your load balancer requires TWO sockets, one for the inbound connection from the client and another for the outbound connection to the backend, so out the gate you're limited to, at most, 512 concurrent connections. Luckily, it's an easy setting to change.

```
1 > vi /etc/security/limits.conf
2      * soft  nofile  999999
3      * hard  nofile  999999
4 > ulimit -n 999999
```

Note: You may have to restart HAProxy for this setting to take effect.

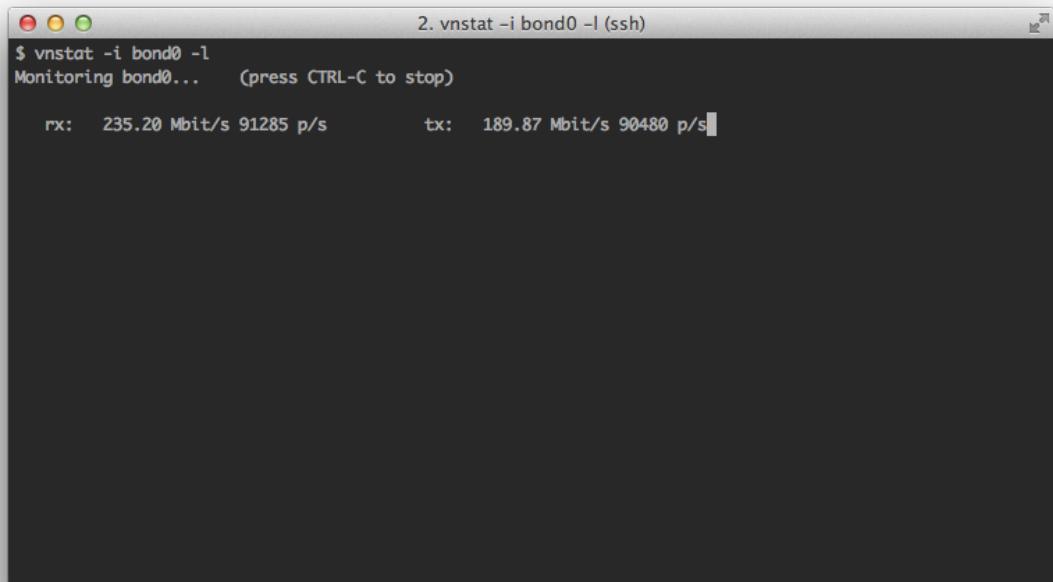
Issues at scale

Debugging load balancer issues can be frustrating, especially if you're dealing with high load. Here are some issues to look for if you're dealing with a slow or unresponsive load balancer.

Saturating the network

This seems like a no-brainer, but it's happened to me before. I remember spending over an hour trying to figure out why one of our Twitpic load balancers was running like molasses and consistently dropping connections. It ended up being that the network card was configurated at 100mbps and we were trying to push 150mbps through it. It's easy to check, though. Using vnstat you can see the current bandwidth going through your network card. Remember to check your *public* AND *private* networks, since your load balancer will most likely be using both.

```
1 > vnstat -i eth1 -l
```



A screenshot of a terminal window titled "2. vnstat -i bond0 -l (ssh)". The window shows the following output:

```
$ vnstat -i bond0 -l
Monitoring bond0... (press CTRL-C to stop)

rx: 235.20 Mbit/s 91285 p/s      tx: 189.87 Mbit/s 90480 p/s
```

Running out of memory

Although unlikely with HAProxy, since the memory footprint is so low, it can happen if you have a high volume of incoming connections and slow/overwhelmed backends. Use `free` to check.

Lots of connections in TIME_WAIT

Like I talked about above, having lots of connections in the `TIME_WAIT` state can overwhelm your server. You can use the configuration options above to mitigate the issue—running `netstat -n | grep TIME_WAIT | wc -l` allows you to see the number of sockets sitting in `TIME_WAIT`.

App Server: Horizontal Scaling with Nginx and PHP-FPM

The application cluster is the brain of your app, responsible for running your PHP code and serving static assets like images, css and javascript. Luckily, app servers are usually the easiest part of the stack to scale, because with the load balancer setup mentioned in Chapter 3, new app servers can be added to the pool and horizontally scaled to increase capacity. And when they fail (because they will fail), HAProxy will automatically remove them from the pool. No alerts, and no waking up in the middle of the night—they scale effortlessly.

Choosing the right version of PHP

The PHP community as a whole is really, really bad about updates. No joke, bad to the point of running releases that are years old. You need to make sure you're running the latest version of PHP, which is PHP 5.4.4 (at the time of writing).

Update for 2014— use PHP 5.5! Everything I say here holds true, except the Dotdeb repositories are deb <http://packages.dotdeb.org> wheezy-php55 all and deb-src <http://packages.dotdeb.org> wheezy-php55 all.

Major releases for PHP keep getting faster and more efficient with memory, so it's important to run the latest release to squeeze the most out of your hardware. 5.4 is substantially better, both in terms of features and speed when compared to 5.3, and likewise when comparing 5.3 to 5.2. PHP is getting better with every major release.

On Debian or Ubuntu, you can use [Dotdeb](#)¹⁸ to get almost immediate, hassle-free updates to PHP, as well as access to some other software that's not kept up-to-date in the Ubuntu Repositories (Redis and nginx, for example). This saves you the hassle of having to wait for your distribution maintainers to update the packages, which often takes weeks, months or even an eternity in some cases.

If you're using Ubuntu Quantal (12.10), PHP 5.4 is already in the apt repository so you don't need to bother adding the php54 lines for Dotdeb.

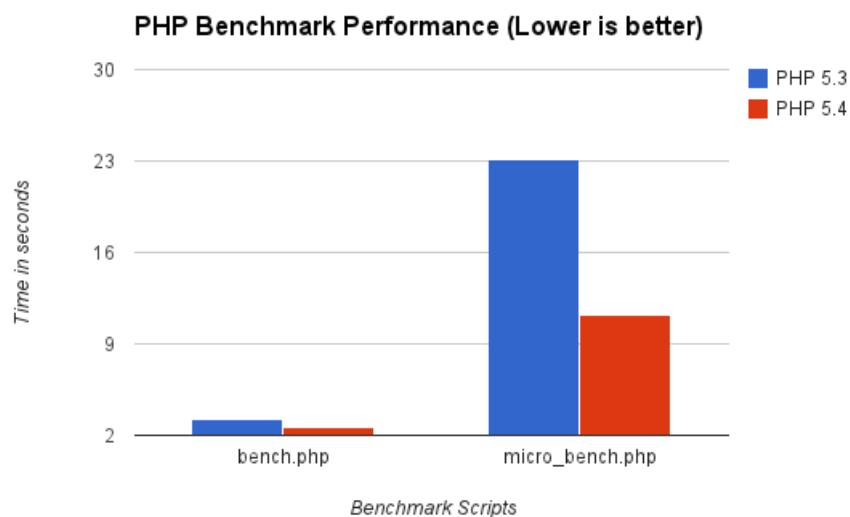
Setting up Dotdeb is as easy as adding another apt source.

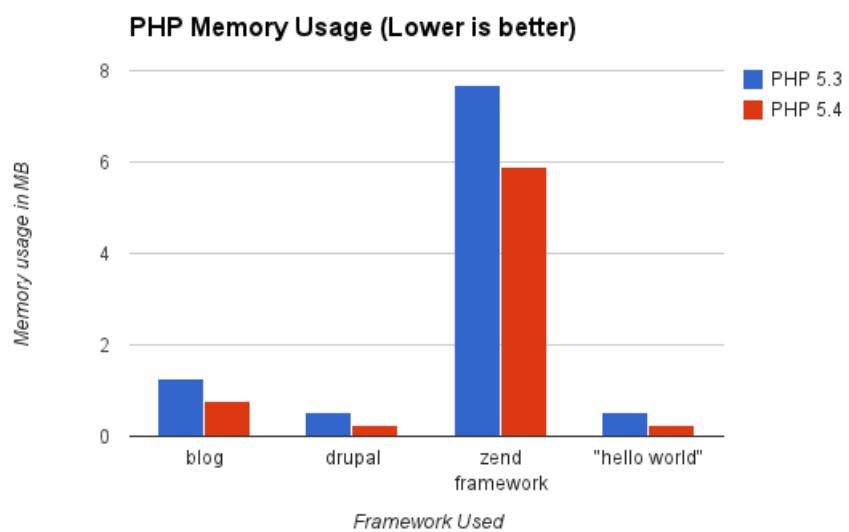
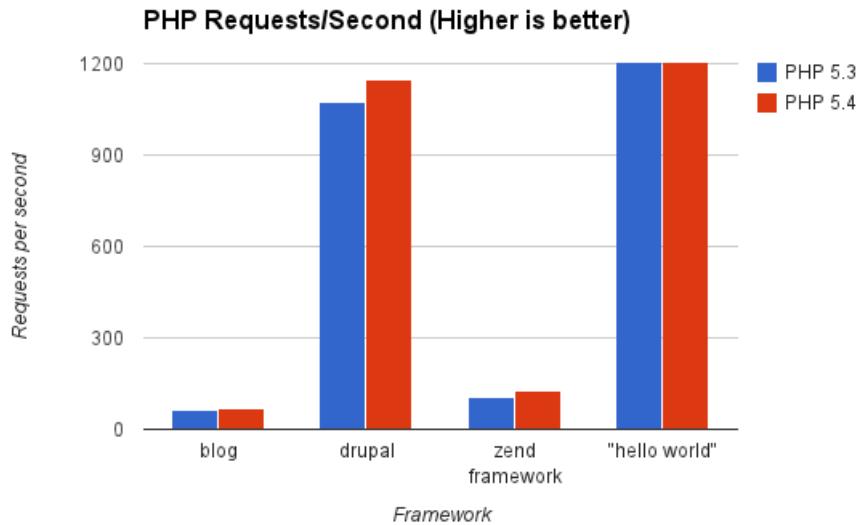
¹⁸<http://www.dotdeb.org/>

```
1 > vi /etc/apt/sources.list.d/dotdeb.list
2
3 deb http://packages.dotdeb.org squeeze all
4 deb-src http://packages.dotdeb.org squeeze all
5
6 # Only include these lines if you are using Ubuntu
7 # 12.04.
8 deb http://packages.dotdeb.org squeeze-php54 all
9 deb-src http://packages.dotdeb.org squeeze-php54 all
10
11 > wget http://www.dotdeb.org/dotdeb.gpg
12 > cat dotdeb.gpg | sudo apt-key add -
13 > apt-get update
```

Benchmarking PHP 5.3 vs PHP 5.4

In every scenario I could find and test, PHP 5.4 outperformed PHP 5.3 in requests per second, as well as memory usage. Here are some graphs with my findings that will hopefully convince you to upgrade to the latest version of PHP!





Nginx and PHP-FPM

As discussed in Chapter 2, we're going to swap out the typical Apache/mod_php for a very elegant nginx/PHP-FPM configuration. Yeah, yeah—you can try to keep Apache and scale it horizontally with HAProxy, but using nginx and PHP-FPM is a much better combination. Trust me, throw out Apache and replace it with nginx.

We moved Twitpic over to using this setup in 2010, and I was admittedly hesitant at first. Apache was all I knew and it was so easy to manage. I was scared that I wouldn't be able to learn nginx or be comfortable with managing it. Boy was I wrong! Nginx is even easier to manage than Apache—the configuration files are very clean and readable. Plus nginx has an [incredible wiki](http://wiki.nginx.org/Main)¹⁹ filled with full documentation on every configuration option.

¹⁹<http://wiki.nginx.org/Main>

Likewise, PHP-FPM is a solid piece of software and just as reliable as using mod_php. Prior to PHP 5.3, it was a patch developed by the same team that created nginx, but now it's bundled with PHP by default. What that means is that you have the full backing and support of the entire PHP community. It's a well-tested piece of software and will definitely get the job done.

Let's assume you have a PHP application in /u/apps/my_app, with the main entry point to the app at /u/apps/my_app/www/index.php. Most frameworks (Zend, Kohana, Cake) should be compatible with this recipe.

Here's how we'd set it up:

```
1  > apt-get install nginx
2  > vi /etc/nginx/nginx.conf
3
4  user www-data;
5  worker_processes 4;
6  pid /var/run/nginx.pid;
7
8  events {
9    worker_connections 768;
10   multi_accept on;
11 }
12
13 http {
14
15   sendfile on;
16   tcp_nopush on;
17   keepalive_timeout 65;
18   server_tokens off;
19
20   include /etc/nginx/mime.types;
21   default_type application/octet-stream;
22
23   access_log /var/log/nginx/access.log;
24   error_log /var/log/nginx/error.log;
25
26   gzip on;
27   gzip_disable      "MSIE [1-6]\.";
28
29   include /etc/nginx/conf.d/*.conf;
30   include /etc/nginx/sites-enabled/*;
31 }
32
33
34 > vi /etc/nginx/sites-available/my_app.conf
35 upstream backend {
36   server unix:/u/apps/my_app/tmp/php.sock;
```

```

37 }
38
39 server {
40
41     listen 80 default;
42     root /u/apps/my_app/www;
43     index index.php;
44
45     access_log /u/apps/my_app/logs/access.log;
46     error_log /u/apps/my_app/logs/error.log;
47
48     location / {
49         try_files $uri $uri/ /index.php;
50     }
51
52     # This location block matches anything ending in .php and sends it to
53     # our PHP-FPM socket, defined in the upstream block above.
54     location ~ \.php$ {
55         try_files $uri =404;
56         fastcgi_pass backend;
57         fastcgi_index index.php;
58         fastcgi_param SCRIPT_FILENAME /u/apps/my_app/www$fastcgi_script_name;
59         include fastcgi_params;
60     }
61
62     # This location block is used to view PHP-FPM stats
63     location ~ ^/(php_status|php_ping)$ {
64         fastcgi_pass backend;
65         fastcgi_param SCRIPT_FILENAME $fastcgi_script_name;
66         include fastcgi_params;
67         allow 127.0.0.1;
68         deny all;
69     }
70
71     # This location block is used to view nginx stats
72     location /nginx_status {
73         stub_status on;
74         access_log off;
75         allow 127.0.0.1;
76         deny all;
77     }
78 }
```

Next, we need to create a symbolic link from the configuration file in /etc/nginx/sites-available to /etc/nginx/sites-enabled. Doing this allows us to quickly disable the application in nginx by removing the symbolic link and reloading nginx.

```

1 > mkdir -p /u/apps/my_app/{logs,tmp,www}
2 > ln -s /etc/nginx/sites-available/my_app.conf \
3       /etc/nginx/sites-enabled/my_app.conf
4 > service nginx start

```

Alright. Nginx is up and running now, but nginx doesn't know how to run PHP. Unlike Apache with mod_php, PHP-FPM runs outside of the webserver in it's own process- nginx just forwards requests to it using FastCGI. We need to install PHP-FPM and configure it.

```

1 > apt-get install php5-fpm
2
3 > vi /etc/php5/fpm/pool.d/my_app.conf
4
5 [my_app]
6 listen = /u/apps/my_app/tmp/php.sock
7 user = www-data
8 group = www-data
9 pm = dynamic
10 pm.max_children = 100
11 pm.start_servers = 10
12 pm.min_spare_servers = 5
13 pm.max_spare_servers = 15
14 pm.max_requests = 1000
15 pm.status_path = /php_status
16
17 request_terminate_timeout = 0
18 request_slowlog_timeout = 0
19 slowlog = /u/apps/my_app/logs/slow.log
20
21 > service php5-fpm start

```

That should be enough configuration to bootstrap your application. Here are a few things worth mentioning:

Use UNIX Sockets for FastCGI/PHP-FPM

You have the option of having nginx talk to PHP-FPM over a TCP socket or a UNIX socket. UNIX sockets are preferred for two reasons:

1. They are marginally faster than TCP sockets because they don't have any network protocol overhead and can avoid hitting the network stack. Realistically, this doesn't make much of a difference on modern hardware running a recent Linux kernel.
2. On extremely high-volume servers, it's possible to run out of TCP sockets while the closed sockets are sitting in the TIME_WAIT state. There are ways to mitigate this, but it's smarter to simply save yourself the trouble by using UNIX sockets, which are only limited by the maximum number of open files.

There are stats for nginx and PHP-FPM baked in

From each application server, you can run `curl 127.0.0.1/php_status` and `curl 127.0.0.1/nginx_status`. You should see some interesting data, similar to this:

```
1 > curl 127.0.0.1/php_status
2 pool: www
3 process manager: dynamic
4 accepted conn: 42244815
5 listen queue len: 0
6 max listen queue len: 128
7 idle processes: 45
8 active processes: 30
9 total processes: 75
10 max children reached: 3
11
12 > $ curl 127.0.0.1/nginx_status
13 Active connections: 57
14 server accepts handled requests
15 334341438 334335834 437990922
16 Reading: 0 Writing: 34 Waiting: 23
```

This data can be hooked up to a monitoring system such as Munin, Monit, Graphite, etc. There are plenty of open-source plugins available on GitHub as well. As you can see, the number of connections, PHP-FPM processes, etc. are available and can be useful when debugging load issues.

Using PHP-FPM's dynamic process management

PHP-FPM works by spinning up some PHP processes to run your application—each process can only handle a single request at a time, so if you need to handle 50 concurrent requests, PHP-FPM must be running at least 50 processes. Without enough available processes, PHP-FPM will have to queue the requests and wait until some become free to handle them.

With this configuration, PHP-FPM dynamically starts and stops processes to handle increased or decreased load, but you need to provide hints and sane limits for it. It's possible to configure a static number of processes that will never change, but it's not worth covering. The dynamic process manager is very robust and handles both high and low load situations extremely well.

The configuration variable `pm.max_children` controls the maximum number of FPM processes that can be run at the same time. Effectively, this setting controls the maximum number of requests that each application server can handle. It should be tuned depending on the memory footprint of your application and the amount of total memory on the server. A good place to start is using this formula:

```
1 pm.max_children = (MAX_MEMORY - 500MB) / 20MB
```

For example, if your application server has 4GB of memory, you'd want to set `pm.max_children` at 180, allowing PHP-FPM to use a maximum of roughly 3.5GB of memory. Remember—you want to leave some memory for nginx to use, as well as any other services you may have running on the system.

The worst thing you can do is set `pm.max_children` too high! Under heavy load, PHP-FPM could very possibly require more memory than is available and begin to swap, likely crashing the server or grinding it to a halt.

`pm.start_servers` is the number of PHP-FPM children that are forked when it first starts up. I recommend 10% of your maximum capacity.

`pm.min_spare_servers` is the minimum number of unused PHP-FPM children that should hang around at any given time. For example, if `pm.min_spare_servers` is set to 5 and you have 40 busy PHP-FPM processes, it will make sure that there are always 45 child processes, with 5 available to handle any new incoming requests.

`pm.max_spare_servers` is the maximum number of unused PHP-FPM children that can hang around before they start getting killed off. This usually happens when your load spikes and drops down after awhile—for example, if PHP-FPM spikes to 55 child processes, but load drops and it is only using 30, there will be 25 processes sitting around doing nothing but waiting for a new web request to come in. With `pm.max_spare_servers` set at 15, we'd kill off 10 of those spare children once they aren't needed anymore.

`pm.max_requests` is the number of requests that a single child process is allowed to serve before it's killed and recycled. This is used to prevent memory leaks, since PHP's garbage collector isn't designed to be long-running. PHP 5.3 includes some improvements to the memory management, so this can be set fairly high. I recommend setting it to atleast 1000, but it depends on your code and which, if any, PHP extensions you are using, as some leak memory more than others.

Choosing the best hardware for your application server

Fast CPU, Lots of memory

Expect each PHP-FPM process to use up to 20MB of memory each, figuring that if `pm.maxchildren` is set at 100 you'll need atleast 2GB of memory for your server.

Nginx, on the other hand, sips on memory—even a loaded application server will only see nginx using 60-80MB of memory total.

You'll want the fastest CPU with the greatest number of cores you can get your hands on. Although PHP is single threaded, as long as you have more PHP-FPM processes than you have cores, you'll be able to take full advantage of each core.

At the time of writing, the Intel Xeon SandyBridge 26xx series seem to be the best bang for your buck. They can run in a dual-cpu configuration, with up to 8-cores for each CPU. With HyperThreading, this effectively gives your application server 32 cores to take advantage of.

Using a PHP Opcode cache

PHP is an interpreted programming language. Every time a PHP application is run, the interpreter needs to read your PHP source code, translate it into an internal opcode program, and then run those opcodes. You can think of the opcode program as an intermediate programming language that the interpreter can understand better—it's really a virtual machine.

Translating your PHP source code to the opcodes has a non-trivial amount of overhead that can be saved by caching the output of the PHP → opcode translation process and re-using for future executions. So how do we accomplish this? Well, we use an opcode cache extension. If you're not already using one, you can easily see a 200-300% reduction in CPU usage. It's an easy win!

As of PHP 5.5, Zend OpCache is bundled with PHP. To enable it, just add `opcache.enable=1` into your `php.ini`. Super easy. The downside is that the user-land memcache-type component that used to be part of APC is not part of Zend OpCache. If you depend on this shared cache, you can install something like [APCu²⁰](#), which brings the user-land cache part of APC back to PHP 5.5. I go into more detail about Zend OpCache and heartaches it caused me while transitioning from APC in the CakePHP Cache Files Case Study.

If you're using PHP 5.4 or below, are a couple of different opcode cache alternatives, and I'll discuss them below.

Are there any downsides to using an opcode cache? For most setups, there are absolutely NO disadvantages. The only *gotcha* that you should know about is that if you're using a PHP encoder/obfuscator, such as [ionCube²¹](#) or [Zend Guard²²](#), you won't be able to use an opcode cache. That being said, this limitation only impacts a small portion of people running closed-source, licensed PHP code.

Tuning Zend Opcache (PHP 5.5 only)

A common pain: “I really want to use Zend Optimizer but I have NO IDEA what settings to use.”

Have you ever noticed how you can spend hours reading docs, but without an example or real world case study, you have NO IDEA what the settings actually do?

Like, you understand what `opcache.memory_consumption` MEANS, but who knows what to set it at? Is 32 too much? Too little? Who the heck knows, nothing comes up on Google, so you just go with your gut.

But what about when your gut is wrong? Do you want to bet your uptime on a guess?

Not me.

I have a bad habit of researching like a maniac. It's awful because I'll start off googling some weird setting and find myself digging through PHP source code three hours later.

Instead of letting this knowledge go to waste, I want to share it with you. I had to spend the time figuring out the best REAL WORLD settings for Zend Optimizer.

²⁰<http://pecl.php.net/package/APCu>

²¹<http://www.ioncube.com/>

²²<http://www zend com/en/products/guard/>

These settings are straight from my `php.ini` file from one of my apps that does 117 million HTTP requests per day. I'll explain what each one does and why it's important so you can tweak it for your setup.

opcache.revalidate_freq - Basically put, how often (in seconds) should the code cache expire and check if your code has changed. 0 means it checks your PHP code every single request (which adds lots of stat syscalls). Set it to 0 in your development environment. Production doesn't matter because of the next setting.

opcache.validate_timestamps - When this is enabled, PHP will check the file timestamp per your `opcache.revalidate_freq` value.

When it's disabled, `opcache.revalidate_freq` is ignored and PHP files are NEVER checked for updated code. So, if you modify your code, the changes won't actually run until you restart or reload PHP (you force a reload with `kill -SIGUSR2`).

Yes, this is a pain in the ass, but you should use it. Why? While you're updating or deploying code, new code files can get mixed with old ones—the results are unknown. It's unsafe as hell.

opcache.max_accelerated_files - Controls how many PHP files, at most, can be held in memory at once. It's important that your project has LESS FILES than whatever you set this at. My codebase has 6000 files, so I use the prime number 7963 for `max_accelerated_files`.

You can run `find . -type f -print | grep php | wc -l` to quickly calculate the number of files in your codebase.

opcache.memory_consumption - The default is 64MB, I set it to 192MB because I have a ton of code. You can use the function `opcache_get_status()` to tell how much memory opcache is consuming and if you need to increase the amount (more on this next week).

opcache.interned_strings_buffer - A pretty neat setting with like 0 documentation. PHP uses a technique called string interning to improve performance—so, for example, if you have the string “foobar” 1000 times in your code, internally PHP will store 1 immutable variable for this string and just use a pointer to it for the other 999 times you use it. Cool. This setting takes it to the next level—instead of having a pool of these immutable string for each SINGLE php-fpm process, this setting shares it across ALL of your php-fpm processes. It saves memory and improves performance, especially in big applications.

The value is set in megabytes, so set it to “16” for 16MB. The default is low, 4MB.

opcache.fast_shutdown - Another interesting setting with no useful documentation. “Allows for faster shutdown”. Oh okay. Like that helps me. What this actually does is provide a faster mechanism for calling the deconstructors in your code at the end of a single request to speed up the response and recycle php workers so they're ready for the next incoming request faster. Set it to 1 and turn it on.

TL;DR-

in `php.ini`

```
1 opcache.revalidate_freq=0
2 opcache.validate_timestamps=0 (comment this out in your dev environment)
3 opcache.max_accelerated_files=7963
4 opcache.memory_consumption=192
5 opcache.interned_strings_buffer=16
6 opcache.fast_shutdown=1
```

APC (PHP 5.4 and below only)

APC is the tried and true opcode cache. It's going to be included with PHP at some point in the future, so it's usually the best choice. APC is distributed as a PHP extension and is easy to install with PECL.

```
1 > pecl install apc
2 > service php5-fpm restart
```

Seriously, it's that easy. On a busy server, you'd see an instantaneous drop in CPU usage. And I'm not talking about a dip, I mean it would seriously fall off a cliff.



There's one gotcha with APC. Make sure to set the configuration value `apc.stat_ctime=1` in `php.ini` if you deploy your code using git, svn, capistrano, or rsync. Otherwise, you may see errors or stale code whenever you push new code to your application servers.

Why? Because in most deployment setups—especially with capistrano—when new code is deployed, only the file metadata gets updated. This causes only the `ctime`, not the `mtime` to get updated as well. Out of the box, APC only checks for code changes by comparing the `mtime`, so even if you deploy new code APC will look at the `mtime` and think it's the same as what it has cached in memory. Using `apc.stat_ctime` tells APC to also check the `ctime`, and fixes the problem entirely.

XCache

An alternative to APC is XCache, which more or less provides the same functionality, though it's written by an independent developer and is not bundled with PHP. That said, it gets frequent releases—for example, a PHP 5.4 compatible version was released as stable long before APC had support for PHP 5.4 in their stable version.

The only reason I'm mentioning it here is because it's what we use at Twitpic, the reasons why are covered in the case study at the end of this chapter.

```
1 > apt-get install php5-xcache
2 > service php5-fpm restart
```

Alternatives

A new trend that we might see more of in the future is using long-running application servers in PHP—that is, the PHP application is started once and sits in a loop waiting for new HTTP requests (this is similar to something like unicorn for Ruby or gunicorn for Python). This type of approach does not need an opcode cache because the PHP process is not restarted for each request, so code only needs to be interpreted once.

I haven't had a chance to play with [Photon](#)²³ yet, but it's one very promising example of this type of PHP application server. It looks very interesting.

Tuning Linux for PHP

Out of the box, Linux can run a fairly high volume of PHP requests without much tuning. Nonetheless, we can tweak it a bit to improve communication between nginx and PHP-FPM, as well as improve I/O performance in situations where PHP might have to hit the disk.

I'm also going to talk about some anti-patterns, or things that might at first seem like a good idea to improve performance, but either have negative side-effects or negligible performance improvements.

²³<http://www.photon-project.com/>

Max open files

Similar to our load balancer, the maximum number of open files allowed per process is set very low (1024). Since nginx communicates with PHP-FPM using Unix sockets, and sockets are considered files on a Linux system, this effectively limits the number of concurrent connections nginx can use to talk to PHP-FPM.

```

1 > vi /etc/security/limits.conf
2      * soft  nofile  100000
3      * hard  nofile  100000
4 > ulimit -n 100000

```

Turning off filesystem access times

By default, in most Linux distributions, the filesystem keeps track of the last time a file was accessed or read. It's rarely useful to have this information and it causes an I/O operation every single time a file is read, such as the PHP interpreter reading your source code. We can disable this behavior by modifying the partition where your PHP files are stored. Open up `/etc/fstab` and look for your main partition. Mine looks like this:

```
1 /dev/sdb1      /      ext4      errors=remount-ro 0 1
```

Change it to

```
1 /dev/sdb1      /      ext4      noatime,nodiratime,errors=remount-ro 0 1
```

`noatime` tells the filesystem to not track when individual files are read, and likewise, `nodiratime` tells the filesystem to not track when directories are read from.

You can remount the filesystem (without a reboot!) with the command `mount -o remount /dev/sdb1` (replace `/dev/sdb1` with whatever filesystem that you had to edit in your `/etc/fstab` file).

Storing temporary files in `tmpfs`

If your application is user-upload heavy, you can reduce some of the I/O overhead by using `tmpfs`, an in-memory filesystem, for the `/tmp` directory, which is where both PHP and nginx buffer uploaded files until they are processed by your application.

Using an in-memory filesystem for temporary files is a great idea because it allows file uploads to be buffered in memory without having to hit disk. This is a great idea for handling small image uploads (like avatars or pictures), but a bad idea if your users are uploading very big files such as videos.

Put this line in your `/etc/fstab` file:

```
1 tmpfs /tmp tmpfs defaults,nosuid,noatime 0 0
```

And mount the new filesystem with `mount /tmp`. If you run `df -h`, you will see the new `tmpfs` filesystem in the output.

```
1 tmpfs 2.0G 4.0K 2.0G 1% /tmp
```

Tweaking your `php.ini` for improved performance

Out of the box, the `php.ini` file is huge and includes every configuration setting known to man. Delete that! You don't need all of those default settings.

```
1 > vi /etc/php5/fpm/php.ini
2 [PHP]
3 engine = On
4 expose_php = Off
5
6 max_execution_time = 5
7 memory_limit = -1
8 error_reporting = E_ALL & ~E_DEPRECATED
9 display_errors = Off
10 display_startup_errors = Off
11 html_errors = Off
12 default_socket_timeout = 5
13
14 file_uploads = On
15 upload_tmp_dir = /tmp/php
16 upload_max_filesize = 50M
17 post_max_size = 50M
18 max_file_uploads = 20
19
20 date.timezone = 'UTC'
21
22 cgi.fix_pathinfo = 0
```

Notice that we set the `max_execution_time` reasonably low, to only 5 seconds. For a fast web application, we don't really want any long-running web requests—in our application, having a web request last longer than 5 seconds usually means that something is wrong. Our goal is for pages to respond in 300-500ms tops!

Anti-Pattern: Storing your code on a Ramdisk

A logical train of thought: Well, if my code is stored on disk, and the PHP interpreter has to read in the source code from the disk every web request, I can speed up the process by storing my

code on a Ramdisk (that is, a portion of memory that is mapped directly to a filesystem). It makes logical sense and I've seen it asked on StackOverflow by well intentioned developers more than a few times.

The reason it's an anti-pattern is pretty simple. **Linux already does this for you!** The Linux kernel will seemlessly use almost all of its free, unused memory to cache files after the first time they're read from the filesystem. It's totally transparent and accomplishes exactly what storing your code in a Ramdisk would do.

Better yet, it doesn't come with any of the extra complexity, and the kernel will free up it's cache under heavy memory pressure, which wouldn't be possible using a Ramdisk.

Anti-Pattern: Turning off access and error logs

The nginx access logs and error logs are very valuable. Not only are they helpful for debugging problems in production, but they're also useful for aggregating stats on your traffic, tracking abusers (think DDoS), and making sure that everything is running as it should.

Still, on a busy server it's tempting to disable these logs in the name of performance. Writing to the disk for every single web request can't be good, right? Wrong! The benefits of having live logs FAR outweigh the marginal performance gain you'd see by disabling them.

1. Hard drives, even rotational non-ssd drives, are very good at sequential writes. Well over 100MB/s.
2. On the busiest servers (think 20,000+ connections per second), nginx only generates 1-2MB of log data/second, which ends up being less than 1% of I/O utilization.
3. Nginx is evented, so it'll never block on writing the log files. Also worth mentioning is that nginx buffers log writes in order to make them more efficient.

Remember, in the Load Balancer chapter we disabled the log files completely because the signal-to-noise ratio is far too high. But, on the application servers each server is only handling a portion of your traffic and the logs become far more valuable and helpful in real-time.

Scaling session handling

Sessions are an important part of every web application—they allow you to keep state between requests and remember your users. The default PHP configuration for sessions is to store the session data in files on your application server. Each user session is assigned a unique token, stored in a cookie within the browser, and this token is subsequently used as the filename to store the session data on the filesystem.

Filesystem based sessions don't scale for two reasons.

The first is because it creates a bunch of unnecessary I/O overhead. For every request, PHP needs to hit the filesystem to read the session data, and at the end of the web request, re-write the file with any new session data that was added or changed during the web request.

The second reason is that it locks you into a single server. Since the session data files are stored individually on a single application server, the session won't be visible to any other servers in your application cluster. For example, if a user logged into his account on `app01`, when he refreshes the page and HAProxy routes him to `app02`, he'll be taken back to the login page because he won't have a valid session data file on `app02`.

There are ways around this, such as storing sessions on a network file system or using `sticky sessions` on your load balancer, but these are ultimately bad choices—think of them as a crutch more than a long term solution. Network file systems are very slow and `sticky sessions` will cause hot-spots and cold-spots throughout your application cluster.

Another horrible piece of advice that's often recommended on forums and sites like StackOverflow is to store session data in a relational database, like MySQL. Database-backed sessions will let you scale a little bit further, but will eventually create an enormous amount of I/O pressure for something you shouldn't be storing in a database to begin with (again, because you're reading from and writing to the database for every single web request). **Sessions are snippets of time-limited throwaway data—not relational data.**

Lucky for us, PHP has a plug-and-play session interface that we can set with the ini setting `session.save_handler`. This interface allows PHP extensions to provide plugins that look and act like native PHP sessions, letting us use the familiar `$_SESSION` variable in a scalable fashion.

There are three pluggable backends worth mentioning, and I'll talk about them below.

Memcached

By far the easiest way to scale your sessions is to setup a memcache server or two and use the session handler provided by the [memcached²⁴](#) extension.

Important: I said [memcached²⁵](#) extension and not memcache. There are two different extensions, the memcached extension being FAR superior for a variety of reasons, mostly because it uses `libmemcached`, which I'll cover more extensively in Chapter 7.

Recently, there has been some criticism for storing sessions in memcache because it's an in-memory, non-filesystem backed storage system. That means if memcache crashes or is restarted, you'll lose all of your session data, effectively logging all of your users out and deleting anything stored in their session. This could be a big downside if you store valuable data in the session, such as shopping cart data or anything else that's important.

The real tradeoff here is speed. Memcache is extremely fast and easy to setup—we use it at Twitpic to store our sessions, and a single server can provide a session backend for tens-of-thousands of simultaneous users on our site. Since it's a central resource, all of our application servers have access to the same session data, and we can scale horizontally without being tied to the filesystem again!

Before we can use memcache to store sessions though, we'll first need to install the [PECL Memcached²⁶](#) extension. Fortunately, it's pretty easy.

²⁴<http://pecl.php.net/package/memcached>

²⁵<http://pecl.php.net/package/memcached>

²⁶<http://pecl.php.net/package/memcached>

```
1 > apt-get install php5-memcached
```

Session save handlers can be registered in `php.ini`, but it's better to do it within your application code. If you set it in `php.ini`, it'll be globally shared with any PHP code running on your application server, even other apps, which can cause undesired behavior.

```
1 ini_set('session.save_handler', 'memcached');
2 ini_set('session.save_path', '192.0.2.25:11211,192.0.2.26:11211');
3
4 # A session token will get stored in the users web-browser as normal,
5 # but instead of the session data getting stored on the filesystem, it'll
6 # be stored for us automagically inside of memcache.
7 session_start();
8 $_SESSION['foo'] = 'bar';
```

Redis

If your session data is more valuable to you, and the off-chance of losing it all due to a server crash is a deal breaker, you should consider using Redis as a backend for your session data. Redis is a data-structure server (covered extensively in Chapter 7) that's similar to memcache because the working data is kept in memory, though it persists its data to disk with the `save` and `appendfsync` configuration options.

You will need the [phpredis](#)²⁷ extension.

```
1 ini_set('session.save_handler', 'redis');
2 ini_set('session.save_path',
3     'tcp://192.0.2.5:6379?timeout=0.5,tcp://192.0.2.6:6379?timeout=0.5');
4
5 # A session token will get stored in the users web-browser as normal, but
6 # instead of the session data getting stored on the filesystem, it'll be
7 # stored for us automagically inside of redis.
8 session_start();
9 $_SESSION['foo'] = 'bar';
```

Cookies

Another method of storing session data is to use a custom session handler that stores all of the session data in a cookie within the user's browser. It's a scalable solution as long as you aren't storing a ton of data in the session—remember, cookies have to be sent back to the server on *every single HTTP request*, so it's best to only use the cookie-session storage for storing less than 1kb of session data. Additionally, most browsers limit the size of a single cookie to around 4kb, so there's a hard limit on the amount of data you can store in a session.

²⁷<https://github.com/nicolasff/phpredis>

One concern with a cookies is private data—the session data (as opposed to just the token) will be stored in the user’s browser and will be visible to the user with a tool such as [Firebug²⁸](#), so it’s not a good idea to store anything you don’t want your users to see inside of cookie-backed sessions. Likewise, unless you’re using SSL, the cookie data will be visible to all going across the wire.

But can’t the users modify the cookie data? No, not if you do it right. We use a technique called HMAC (Hash-based message authentication code) to sign the cookie data, so that when we get the session data back from the user, we can verify with 100% certainty that it has not been tampered with.

Cookie-backed sessions aren’t built into PHP, but using the [SessionHandlerInterface²⁹](#) in PHP 5.4, it’s easy to integrate. SessionHandlerInterface’s behavior can be ported to older versions of PHP with `session_set_save_handler()`, but that’s an exercise for the reader.

I’ve provided a working example class below to show how we’d implement our own `SessionHandlerInterface`. Additionally, I’ve open-sourced this code on [GitHub³⁰](#) (because I love sharing and giving back) and released it on [Packagist³¹](#) for easy installation.

*Note: In order for this technique to be secure, you absolutely **must** change the `HASH_SECRET` to a 128+ character random string before deploying the code*

```

1 <?php
2
3 class SessionHandlerCookie implements SessionHandlerInterface {
4
5     private $data      = array();
6     private $save_path = null;
7
8     const HASH_LEN      = 128;
9     const HASH_ALGO     = 'sha512';
10    const HASH_SECRET   = "YOUR_SECRET_STRING";
11
12    public function open($save_path, $name) {
13        $this->save_path = $save_path;
14        return true;
15    }
16
17    public function read($id) {
18
19        // Check for the existance of a cookie with the name of the session id
20        // Return an empty string if it's invalid.
21        if (! isset($_COOKIE[$id])) return '';
22

```

²⁸<http://getfirebug.com/>

²⁹<http://www.php.net/manual/en/class.sessionhandlerinterface.php>

³⁰<https://github.com/stevecorona/SessionHandlerCookie>

³¹<http://packagist.org/packages/session-handler-cookie/session-handler-cookie>

```

23  // We expect our cookie to be base64 format, so decode it.
24  $raw = base64_decode($_COOKIE[$id]);
25  if (strlen($raw) < self::HASH_LEN) return '';
26
27  // The cookie data contains the actual data w/ the hash concatenated
28  // to the end, since the hash is a fixed length, we can extract the
29  // last HMAC_LENGTH chars to get the hash.
30  $hash = substr($raw, strlen($raw)-self::HASH_LEN, self::HASH_LEN);
31  $data = substr($raw, 0, -(self::HASH_LEN));
32
33  // Calculate what the hash should be, based on the data. If the data
34  // has not been tampered with, $hash and $calculated
35  // will be the same
36  $calculated = hash_hmac(self::HASH_ALGO, $data, self::HASH_SECRET);
37
38  // If we calculate a different hash, we can't trust the data.
39  // Return an empty string.
40  if ($calculated !== $hash) return '';
41
42  // Return the data, now that it's been verified.
43  return (string)$data;
44
45 }
46
47 public function write($id, $data) {
48
49  // Calculate a hash for the data and append it to the end of the
50  // data string
51  $hash = hash_hmac(self::HASH_ALGO, $data, self::HASH_SECRET);
52  $data .= $hash;
53
54  // Set a cookie with the data
55  setcookie($id, base64_encode($data), time()+3600);
56 }
57
58 public function destroy($id) {
59  setcookie($id, '', time());
60 }
61
62 public function gc($maxlifetime) {}
63 public function close() {}
64
65 }
66
67 $handler = new SessionHandlerCookie;
68 session_set_save_handler($handler, true);

```

```
69 session_start();
```

Database Server: Ultimate MySQL Tuning Guide

The database is the heart and soul of your application. It keeps your data safe and pumps it out to all of the application servers. After talking with and interviewing hundreds of my readers, most of them have told me that, hands down, their database has been the most difficult part of their stack to scale.

Even though MySQL is arguably one of the most popular databases in existence, there isn't much helpful information available to help you grow your database from the initial "basic" configuration. Searches online often come up empty-handed and if you do find something, it's either a copy-and-paste of someone's 100 line `my.cnf` file (obviously with no explanation why it works), or some "holier-than-thou" armchair expert telling you to hire a DBA. Hire a DBA, right! That would be great if you were a 100-person enterprise, but a small shop with 5-10 programmers? Or even just two dudes working from their apartments? Forget about it- in what world could you justify having a DBA when you're that small?

Let me tell you a secret. Database optimization and configuration isn't some arcane craft that only those who've completed the 10 sacred quests are allowed the privilege to learn. In fact, it breaks down into a couple of pretty simple rules.

In this chapter, we're going to focus on MySQL, since that's what most people are already locked into and it's not exactly easy to switch to something else at the drop of a hat. That being said, most databases (even NoSQL ones) follow the same scaling principles. Though the settings may have different names, the information in this chapter is still relevant, even if you're not using MySQL.

Getting NoSQL Performance out of MySQL

MySQL gets tons of bad press for not being the fastest kid in town, but here's a secret. With the right tweaking, MySQL can hold its own. NoSQL really shines when it comes to cluster management and scaling out horizontally to multiple servers. For example, Cassandra can bootstrap hundreds of brand new nodes with data, automatically, without any manual configuration. Compare that to having to manually configure and transfer data to a new MySQL slave or shard (which can take hours), and you'll see where the benefits really shine.

Here's the thing, though. You can get pretty far with one server and some read-only slaves. Twitter and Facebook still use MySQL, and I think that speaks for something. 37Signals often talks about how they've been able to take advantage of Moore's Law to scale out without having to deal with complex sharding setups. Scaling your database horizontally and moving to a NoSQL database is great, but comes at a huge cost. Data needs to be reorganized, imported, and code needs to be rewritten. I'm a huge fan of NoSQL but I also believe in using the right tool for the right job, and for most jobs MySQL is the right tool.

Getting the most out of MySQL involves some configuration tweaking, and I'm going to touch on all of the most important settings that need to be tuned. However, there are thousands of settings available, and I'm not going to be able to cover all of them here. Instead, I'm going to give you the "too long, didn't read" version and show you how to get the best bang for your buck.

If you want to know what every single configuration option (of the thousands available) does, as well as understand the deep inner workings of MySQL, you should look into the "bible" of MySQL books: [High Performance MySQL: Optimization, Backups, Replication, and More](#)³². It's the best book (700 pages) available on Advanced MySQL tuning.



What is a slave?

In MySQL, we can scale out using a "Master/Slave" setup. That is to say, we designate a single server as the authoritative owner. All data from that server, or *master*, is replicated to *slave* servers, which provides fault-tolerance and more database capacity. Since the data flows in **one direction** only (from the master to the slaves), the slaves are read-only replicas and **all writes must go through the master**. If you write data to a slave server directly, it will not be replicated and will cause data fragmentation and inconsistency.

Choosing the right MySQL version

You should always be running the latest major version of MySQL available to you. Oracle releases new versions of MySQL every couple of months and it's important to keep up to date because there are often significant performance enhancements made in every version. At the time of publishing, 5.5 is the latest major version, but 5.6 is due out later this year. The speed difference between the same exact queries running on 5.1 and 5.5 can be staggering, so upgrading to the latest version is always a good idea.

Furthermore, benchmarks show that I/O performance on SSD drives is about 20% better on 5.5 than 5.1.

All right, so choose the latest version of MySQL - that's simple. But wait, not so fast! Another option that I recommend exploring is using the [Percona fork of MySQL](#)³³.

Why use the Percona fork of MySQL?

Percona is a consulting company for MySQL that's very well known for their [MySQL Performance Blog](#)³⁴. On top of that, they wrote the [High Performance MySQL: Optimization, Backups, Replication, and More](#)³⁵ book that I mentioned previously. So when I say that they're experts in MySQL, know that I truly mean it.

Anyways, "Percona Server" is a 100% free drop-in replacement for MySQL server and is constantly rebased with the latest MySQL code, so it's always up-to-date with the latest

³²<http://www.amazon.com/dp/0596101716?tag=stevecorona-20>

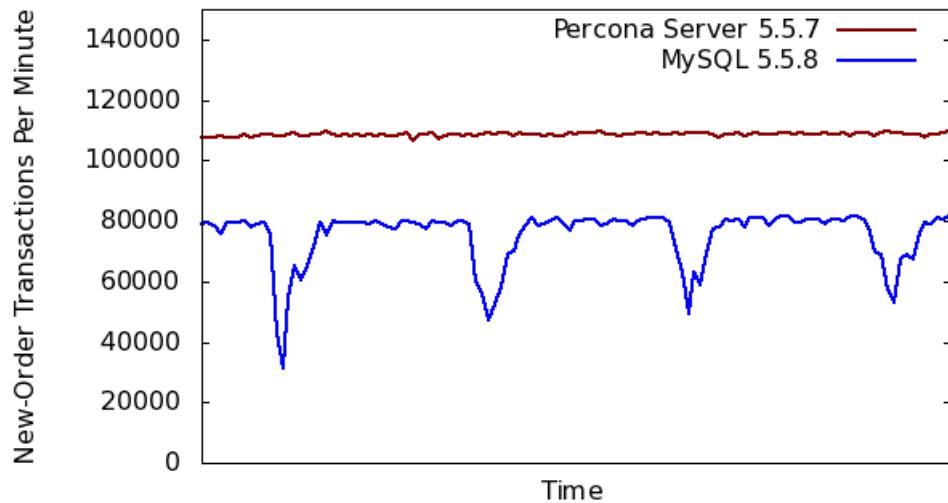
³³<http://www.percona.com/>

³⁴<http://www.mysqlperformanceblog.com>

³⁵<http://www.amazon.com/dp/0596101716?tag=stevecorona-20>

MySQL releases. Your existing code and databases will work with Percona Server without any modifications, just faster while using fewer resources.

- It's 40% faster than the stock MySQL Server.
- Provides more consistent performance over MySQL Server.
- XtraDB, a fork of InnoDB, is included, which offers more tunability and better memory usage.
- More tuning options, giving you greater control and much more. You can [view the feature comparison here³⁶](#).



It's also worth mentioning that Percona Server is rock solid. We've used it at Twitpic in production for over a year without any crashes, data loss, or issues. On top of that, other big players like Flickr, Tumblr, Etsy, Engineyard, and 37Signals are using Percona Server in production, too.

Installing Percona Server

```

1 > gpg --keyserver hkp://keys.gnupg.net --recv-keys 1C4CBDCDCD2EFD2A
2 > gpg -a --export CD2EFD2A | sudo apt-key add -
3 > vi /etc/apt/sources.list.d/percona.repo.list
4     deb http://repo.percona.com/apt precise main
5     deb-src http://repo.percona.com/apt precise main
6 > apt-get update
7 > apt-get install percona-server-server-5.5 percona-server-client-5.5

```

Should I shard my data?

Probably not! Of course, it's hard to say without knowing your data intimately, but in most cases sharding with MySQL is using the wrong tool for the wrong job. When you shard your

³⁶http://www.percona.com/doc/percona-server/5.5/feature_comparison.html

data, you lose everything that MySQL is good at and gain everything that MySQL is very bad at (particularly cluster management). Since you have to rewrite most of your existing code to shard your data, in most cases you're better off switching to a database that supports sharing natively like [MongoDB³⁷](#) or [Apache Cassandra³⁸](#).



What is Sharding?

Sharding is when you split your data across multiple servers, so that no single server holds all of your data. Sharding allows you to scale more effectively when your database usage is too high for a single server. While many NoSQL databases include built-in support for sharding, MySQL does not. When you shard using MySQL, you lose the ability to run queries on your entire dataset and often need to write custom code for interacting with your database.

Dealing with libmysql's inability to set timeouts

Let's talk for a second about how PHP communicates with MySQL. Whether you use `mysql`, `mysqli` or `PDO`, internally PHP either uses `libmysql` or `mysqlnd` to connect and communicate with your MySQL server.

When PHP 5.3 came out, one new feature that it added was `mysqlnd`, which stands for *MySQL Native Driver*. The MySQL Native Driver is a custom C library that PHP uses internally for handling MySQL connections. Prior to `mysqlnd`, PHP was using `libmysql`, a standard C library that ships with MySQL to interact with MySQL.

Since `mysqlnd` was written by the PHP team for the exclusive use of PHP, it's a better fit and is able to tie in nicely with PHP. For example, you can now subclass `mysqlnd` with pure PHP code and change or inject functionality into the way that PHP interacts with MySQL. People have used this ability to add in query logging, [load balancing³⁹](#), and even a PHP-based [query cache⁴⁰](#).

One major flaw with both `libmysql` and `mysqlnd` is that neither allow you to set a connection timeout with a resolution of less than 1 second and neither allow you to set a query timeout at all. That means that if one of your MySQL servers becomes unreachable, the lowest possible timeout (for you to error or retry the connection) is 1 second, which is far too long in a high-volume environment.

Setting a connection timeout with PDO

³⁷<http://www.mongodb.org/>

³⁸<http://cassandra.apache.org/>

³⁹<http://www.php.net/manual/en/book.mysqlnd-ms.php>

⁴⁰<http://www.php.net/manual/en/book.mysqlnd-uh.php>

```

1 <?php
2 $pdo = new PDO("mysql:host=localhost;dbname=test", "username", "password");
3 $pdo->setAttribute(PDO::ATTR_TIMEOUT, 1);

```

Setting a connection timeout with mysqli

And the way we do it with `mysqli` is very similar. Make sure to note the comment that's in line with the code. You can't use the `mysqli` constructor like you're used to doing (i.e, `$mysqli = new mysqli('localhost')` won't work).

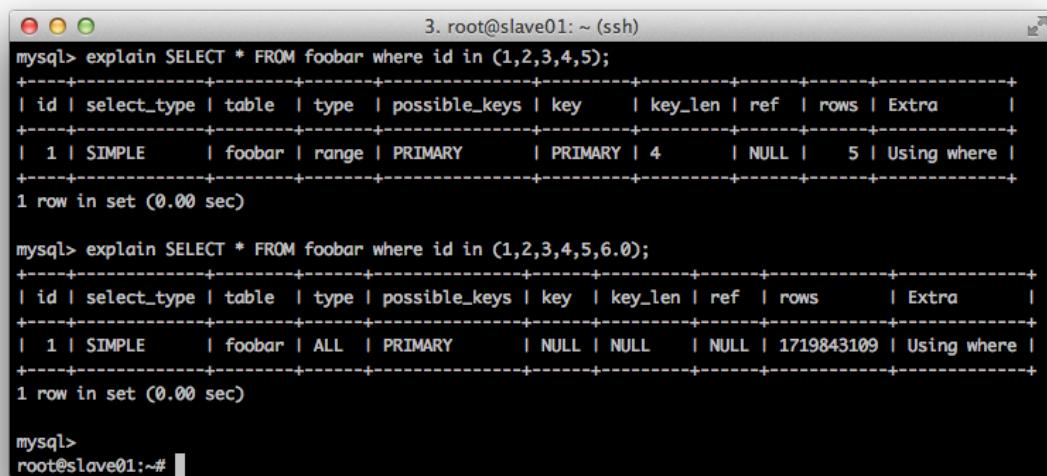
```

1 <?php
2 // Note: When setting a connection timeout with mysqli,
3 // you HAVE to use mysql_init() (which returns a mysqli
4 // object) to create the object and cannot use the constructor.
5 $mysqli = mysqli_init();
6 $mysqli->options(MYSQLI_OPT_CONNECT_TIMEOUT, 1);
7 $mysqli->real_connect("localhost", "username", "password", "test");

```

Query Timeouts

Without a query timeout, a rogue query could block your PHP-FPM process for up to `max_execution_time` (set in `php.ini`, in Chapter 5 we set it as 5 seconds). Still this is too long! In an ideal world, all of our queries have predictable performance characteristics but this isn't an ideal world. The same query with different values in the `WHERE` clause can generate vastly different query plans. For example, look at the following two queries:



```

3. root@slave01: ~ (ssh)
mysql> explain SELECT * FROM foobar where id in (1,2,3,4,5);
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | foobar | range | PRIMARY | PRIMARY | 4 | NULL | 5 |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain SELECT * FROM foobar where id in (1,2,3,4,5,6,0);
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | foobar | ALL | PRIMARY | NULL | NULL | NULL | 1719843109 |
+----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql>
root@slave01:~# 

```

Running the same query on a table with 1.7 billion rows (yeah, that's billion with a B). MySQL can easily scale into the billions with nothing more than the tuning in this chapter; the only difference between the two is that the first query has all integers in the `IN` statement and the

second query has an extra non-integer parameter. So what's the difference? The first generates a query plan to use the PRIMARY index and will respond almost immediately. The second query has to do a full-table scan, scanning every single one of the 1,719,843,109 rows and will likely take hours to run.

Even when PHP's `max_execution_time` kills the PHP-FPM process, the query will continue to run on your MySQL servers - hammering the CPU and killing performance. There has to be better way!

Well, the sad news is that there really isn't. There's not a simple way to do query timeouts in PHP nor is there a server-setting that you can set in MySQL. It's a pretty big oversight, in my opinion. There are two options that I'm going to talk about.

Piggybacking timeouts from HAProxy

In the “Load Balancing MySQL with HAProxy” section later in this Chapter, I talk about how you can load balance MySQL slaves using HAProxy. One small but important feature is that HAProxy allows you to set up TCP timeouts, which effectively gives you the ability to set MySQL connection and query timeouts down to the millisecond. Jump ahead to that section to learn more, but the relevant settings in `/etc/haproxy/haproxy.cfg` are below.

```
1 timeout connect 500ms
2 timeout client 1000ms
3 timeout server 1000ms
```

Using `pt-kill` from the Percona Toolkit

The [Percona Toolkit](#)⁴¹ includes a tool called `pt-kill` which can monitor MySQL for long-running queries and kill them when they pop-up. Of course, sometimes you have legitimate long-running queries from background jobs or OLAP systems, so what I typically do is set up a MySQL user just for my application and use the `--match-user` flag to only kill long running queries for that user. That way, `pt-kill` doesn't interfere with legitimate long-running queries.

```
1 > pt-kill --daemonize --kill --busy-time 60s --match-user webuser
```

Before you use `pt-kill`, you need to install the `percona-toolkit` package. If you follow the instructions under the “Installing Percona Server” above, you can simply install `percona-toolkit` with `apt-get`.

```
1 > apt-get install percona-toolkit
```

The full documentation for `pt-kill` is available [here](#)⁴².

⁴¹<http://www.percona.com/doc/percona-toolkit/2.1/index.html>

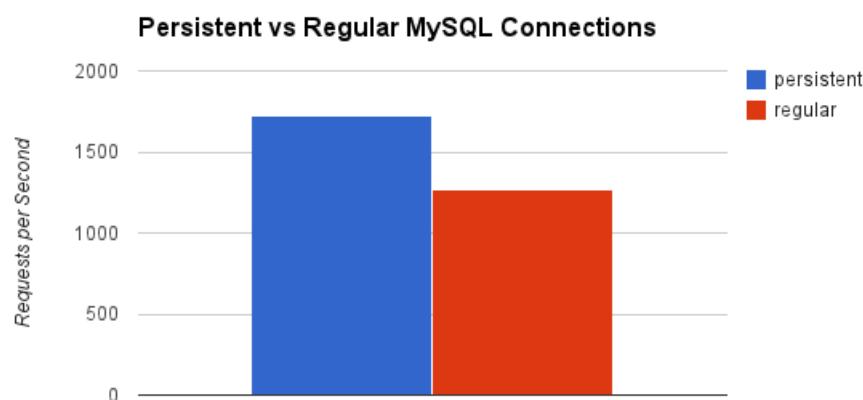
⁴²<http://www.percona.com/doc/percona-toolkit/2.1/pt-kill.html>

Should you use Persistent MySQL Connections?

There is so much bad information about MySQL Persistent connections in PHP, it's hard to get a solid understanding of them and their pitfalls. Most people say, no, you should absolutely not use persistent connections but I think part of the reason is just because it's such a misunderstood topic.

A quick Google search shows lots of threads that read like this- "Well, you probably don't **need** persistent connections, so don't use them!" Or, "Well, they are really complex so don't use them!" Premature optimization and all that. But, sometimes you DO have thousands of MySQL connections, and persistent connections are an option worth exploring.

Before we talk about the pros and the cons, I ran some benchmarks with PHP 5.4 and discovered that persistent connections are about 20-25% faster than normal non-persistent connections. This ended up being around a 20ms difference per request, which obviously isn't much, but when you're talking about a single request or when you are dealing with thousands of requests per second, it adds up.



Benefits

- You remove the overhead of creating a new MySQL connection for each new HTTP request.

Negatives

- There is no mechanism to "reset" a connection, so if you leave a persistent connection in an inconstant state (like an uncommitted transaction), it can still get committed by another PHP-FPM worker that reuses the connection. `mysqli` mitigates this by calling `mysql_change_user()` every time that you re-use a persistent connection, which resets *most* of the connection state.
- Any MySQL variables you set on the connection (with the SET command) will still be there when the connection is reused.

- Your MySQL `max_connections` setting needs to be high enough to handle a connection from every PHP-FPM child in your application cluster. So, if you set `pm.max_children` to 100 in your PHP-FPM pool, and you have 10 application servers, MySQL's `max_connections` needs to be at least 1000.

Persistent connections *do make sense* in some scenarios, for example; if you're creating a new MySQL connection for most of your HTTP requests, not doing anything fancy with SET, and are diligent with your transactions.

Tuning your MySQL Configuration

MySQL offers a vast amount of tuning options through its `my.cnf` file, which (on an Ubuntu system) is located in `/etc/mysql/my.cnf`. Below are the most important settings, with "no-bullshit" comments telling you exactly how they work and impact your system. I'm not covering any MyISAM settings because it's horrible and should almost never be used in a high-volume situation.

```
1 [mysqld]
2
3 # Without considering MyISAM buffers, each connection uses about
4 # 192KB of memory. You need to make sure that-
5 # (max_connections * 192KB) + innodb_buffer_pool_size is less
6 # than your total amount of system memory, otherwise MySQL could
7 # start swapping. Default value is 100 connections.
8 # 2000 connections will use ~400MB of memory.
9 max_connections=2000
10
11 # This should be set to average number of connections * number
12 # of your most commonly accessed tables. So, if, on average
13 # you have 50 connections and have 5 frequently accessed tables,
14 # set this as 250. The default is low (64)
15 table_cache=250
16
17 # Disable the query cache. Both of these must be set as 0 due
18 # to a bug in MySQL. The query cache adds a global lock and
19 # performs poorly with a non-trivial write-load.
20 query_cache_size=0
21 query_cache_type=0
22
23 # Instead of having one big InnoDB file, split it up per table.
24 # This has no performance implications, gives you better data
25 # management features over your individual tables, and more
26 # insight into the system. It only impacts new tables and will
27 # not affect old "monolithic file" style InnoDB tables.
28 innodb_file_per_table
```

```
29
30 # InnoDB has two different versions: Antelope (the older version)
31 # and Barracuda (the newest). Tell InnoDB that we always want to use
32 # the Barracuda.
33 innodb_file_format=barracuda
34
35 # Tells the operating system that MySQL will be doing its own
36 # caching and that it should skip using the file system cache.
37 # Prevents double caching of the data (once inside of MySQL and
38 # again by the operating system.)
39 innodb_flush_method=O_DIRECT
40
41 # Set this to 1 on your master server for safest, ACID compliant
42 # operation (sync after every transaction, high I/O)
43
44 # Set this to 2 on your slave, which can cause up to a second of
45 # data loss after an operating system crash, but frees up I/O
46 # because it only fsyncs data to disk once per second.
47 innodb_flush_log_at_trx_commit=1
48
49 # Set this to 1 on your master server
50 sync_binlog=1
51
52 # Bigger log file size = less I/O used for writes, longer
53 # recovery time during a failure
54 innodb_log_file_size=128M
55
56 # Set this to ~90% of your memory. This is probably the most
57 # important MySQL value that you need to tune.
58 # 64GB -> 57GB
59 # 32GB -> 28GB
60 # 16GB -> 14GB
61 # 8GB -> 7GB
62 innodb_buffer_pool_size=57000M
63
64 # InnoDB uses background threads to prefetch
65 # and store data. The default is 4 threads, but
66 # should really be 4 * # of CPU cores
67 innodb_read_io_threads=32
68 innodb_write_io_threads=32
69
70 # This should be set as the maximum amount of
71 # IOPS that your system has. It sets a max cap
72 # on how much I/O that InnoDB can use.
73 innodb_io_capacity = 5000
74
```

```

75  # This limits the number of threads that InnoDB can perform
76  # at a given time. Setting it to 0 means that it's infinite and
77  # is a good value for Percona 5.5. Non-Percona setups should
78  # set it as 1x the number of CPU cores.
79  innodb_thread_concurrency=0

```

innodb_buffer_pool_size

Arguably, this is the most important configuration setting when tuning MySQL/InnoDB. MySQL caches table data from the disk in memory in order to speed up access times, and it uses this setting to determine exactly how much memory it should use. On a read-heavy workload (i.e., MySQL Slave), you'll want to set this as high as possible while avoiding the possibility of swapping.

A good value is 90% of your server's memory. On a read-heavy workload, if you use `iostat` and see that you have a very high utilization or service time, you can usually add more memory (and increase `innodb_buffer_pool_size`) to improve performance.

Server Memory	Value
64GB	57GB
32GB	28GB
16GB	14GB
8GB	7GB

On a write-heavy workload (i.e., MySQL Master), it's far less important. I've had masters with 128MB buffer pools replicating to slaves with 64GB buffer pools. Also, if you setup a MySQL slave for making backups, you can usually set this low there, too, in order to save memory.

innodb_flush_log_at_trx_commit

This setting can be tweaked to improve performance at the slight cost of data durability. I use it on **SLAVE servers only** because it can cause up to 1 second of possible data loss in the event of a crash or ungraceful shutdown (i.e., tripping over the power cord).

On your master database, you should always set `innodb_flush_log_at_trx_commit=1`. Doing otherwise is irresponsible and can cause data loss.

I like to live on the edge with my slave databases, though. They are repairable, so I consider them throwaway and live a little more on the edge with performance improvements at the cost of durability.

When `innodb_flush_log_at_trx_commit` is set to 1, MySQL will write database changes to the InnoDB Log File and `fsync` them to the disk immediately. The system call `fsync` tells Linux to write the data to the disk *right now*, instead of caching it and writing it sometime in the near future. Obviously, this gives you the most durability because once the `fsync` call returns, you can be 100% positive that the data is stored safely on the hard drive and will still be there if someone rips out the power cord. Obviously, this durability comes at the cost of speed. Writing to the disk for every transaction is slow and uses valuable IOPS, but it's necessary on your master server.



What are IOPS?

IOPS is short for I/O operations per second, I/O referring to your disk drive. Reading and writing to your drive consumes an I/O operation and drives can only perform a limited number of I/O operations per second. On the low side of the IOPS scale are traditional rotational disks (around 200 IOPS). You can improve the number of IOPS by putting them in a RAID, but they are still slow. Likewise, the faster the disk (7200RPM vs 15000RPM), the more IOPS they can provide. On the high end of the spectrum are SSD drives. Depending on the manufacturer, type of flash, and internal components, they can provide anywhere from around 5000 IOPS to over 1 million IOPS.

However, on your slave, you can change `innodb_flush_log_at_trx_commit` to 2. When it's set to 2, MySQL will write out the database changes to the InnoDB log file immediately, but won't call `fsync` just yet. Instead, it will let the operating system cache the write into memory and a background thread will call `fsync` about every second. What this means is that if the server crashes, you could possibly lose up to 1-second worth of data. The trade-off is that your slave servers spend much less of their disk IOPS on handling writes from MySQL replication and can better use the IOPS for scaling your database reads.

The problem that `innodb_flush_log_at_trx_commit` solves is that calling `fsync` is usually *very* expensive in terms of performance. However, if you're using a battery-backed RAID controller with a cache, they can often significantly improve `fsync` speed and may make this setting unnecessary.

`innodb_flush_method`

The InnoDB flush method supports a bunch of different values, none of which are worth going over because the only one that matters is `O_DIRECT`. Setting this value as `O_DIRECT` tells the operating system not to buffer any of the data from the database inside of the file system buffers, because MySQL will be doing that itself (with `innodb_buffer_pool_size`). Not setting this value to `O_DIRECT` will cause double buffering- that is, MySQL and Linux will be buffering the same data, which is a waste of memory.

`sync_binlog`

The binary log is used to log all non-SELECT SQL statements run on the database. It's an important component of MySQL replication. Anyways, on your master server, similar to `innodb_flush_log_at_trx_commit`, the value of `sync_binlog` should be 1. Setting it as 1 tells MySQL to `fsync` the SQL statements added to the binary log in real-time, so you never lose any data and a crash doesn't cause replication to break.

Since slaves don't keep a binary log unless they are told to with `log_slave_updates`, you can remove `sync_binlog` or set it as 0 on your slave servers.

`innodb_log_file_size`

Let's talk about the `innodb_log_file_size` variable. To understand what values to use, instead of just blindly plugging in numbers, let me first explain how it works from a high-level view.

Imagine you were designing a database. You designed it to store the data in a complex file format that was very efficient but required a bit of computation time and random I/O. Since random I/O is slow, and you want to make your database as fast as possible, you decide that you can make your database more efficient by calculating and storing the data after answering the client instead of making them wait for you.

This is great! Things are fast. But uh-oh, Houston we have a problem. If the server crashes before you can store the data into the complex file format, that data will be lost forever since it hadn't been persisted to the disk and was only stored in memory.

One solution you might come up with is to write the data, as it comes in, into a log file. Since the log file is being written sequentially, and there is no random I/O, it ends up being very fast because even the cheapest hard drives are good at doing sequential I/O. Since you know the data is stored in the log file, you can take your time storing it in the slower, but more efficient, file format. Maybe you can even re-order and group different pieces of data together since you're in no rush to persist it to the complex format. If the server crashes, you can just replay the log file and verify that each change has already been made.

That's more or less how the InnoDB log file works. There are two of them, and they're capped at a fixed size. They need to be big enough that they can handle your incoming writes while still allowing InnoDB to leisurely push the data into its more complex data files. Set too small, I/O usage will increase because InnoDB won't be able to efficiently group and re-order writes. Set too big and, in the event of a crash, it will take longer to recover your database because there is more log data that has to be replayed.

So how do we determine the best `innodb_log_file_size`? The best way, instead of setting it to some random value that you read in a book, is to calculate how many megabytes of data per minute are getting written to the file and setting the value to be large enough to hold one hour's worth of writes. Make sure to do this during peak usage, too!

```
1  > show global status where variable_name like "%Innodb_os_log_written%";  
2  select sleep(60);  
3  show global status where variable_name like "%Innodb_os_log_written%";  
4  
5  ***** 1. row *****  
6  Variable_name: Innodb_os_log_written  
7  Value: 234825649664  
8  1 row in set (0.01 sec)  
9  
10 +-----+  
11 | sleep(60) |  
12 +-----+  
13 | 0 |  
14 +-----+  
15 1 row in set (1 min 0.00 sec)  
16  
17 ***** 1. row *****  
18 Variable_name: Innodb_os_log_written
```

```

19      Value: 234828058624
20 1 row in set (0.00 sec)

```

Ok, now take those two values - 234825649664 and 234828058624 and subtract them from each other.

```
1 234828058624-234825649664 = 2408960 bytes
```

2408960, that's the number of bytes per minute that are getting written to my InnoDB log file. [If you Google 2408960 bytes to megabytes⁴³](#), it tells me that MySQL is writing 2.29MB per minute to my log file.

Multiply the megabytes per minute times 60 minutes to determine how much data would be written out in one hour.

```
1 2.29MB * 60 minutes = 137.4MB
```

Since InnoDB keeps two individual log files, divide 137.4MB by 2 and we get about 69MB, so `innodb_log_file_size` should be around 70MB in this scenario. We can go higher, even up to 100MB or 200MB, but as you can see, for most cases, setting it very high is unnecessary.

(Calculation method adopted from the [MySQL Performance Blog⁴⁴](#))

In order for the change to take effect on an existing MySQL database, you need to cleanly shut down MySQL, delete the InnoDB log file, and restart MySQL. If you don't do this, MySQL will complain that the existing log file sizes do not match the value of `innodb_log_file_size`.

The files you need to delete are in your MySQL data directory, named `ib_logfile0` and `ib_logfile1`. Remember, these files are only necessary in the event of a crash, so you may delete them freely as long as you've cleanly shut down MySQL using `mysqladmin shutdown`. MySQL will recreate the files, using the new size specified, when you start it again.

Caution: Do not delete the `ibdata1` file, as this contains all of your InnoDB data if you're not using `innodb_file_per_table`. It's easy to accidentally do this if you use a command like `rm -rf ib*..`

max_connections

You need to set `max_connections` as high as the theoretical maximum amount of connections that your application can ever have. Since each PHP-FPM process will typically only use one connection, you'll want to use the `pm.max_children` setting from your PHP-FPM configuration file for this formula.

Formula for setting `max_connections`:

⁴³https://www.google.com/?q=2408960+bytes+to+megabytes&hl=en&safe=off&output=search&scion=psy-ab&q=2408960+bytes+to+megabytes&oq=2408960+bytes+to+megabytes&gs_l=hp.12...0.0.0.1163.0.0.0.0.0.0.0.0.0.les%3B..0.0...1c.ONtw7scylt0&pbx=1&bav=on.2,or.r_gc.r_pw.r_cp.r_qf.&fp=19fc9cab2f77850f&bpcl=35440803&biw=1129&bih=999

⁴⁴<http://www.mysqlperformanceblog.com/2008/11/21/how-to-calculate-a-good-innodb-log-file-size/>

```
1 max_connections = pm.max_children * number of application servers
```

The default value of 100 connections is far too low, so make sure you bump this up. Another consideration is making sure you have enough memory to handle your theoretical maximum number of connections. MySQL is a threaded server and creates a new thread for each connection, each which requires about 192KB of memory. 192KB * 2000 connections is around 400MB, for example, so you'll want to make sure that you have enough free memory to accommodate your connections after subtracting `innodb_buffer_pool_size` from the total amount of system memory.

innodb_io_capacity

The `innodb_io_capacity` setting tells InnoDB how many IOPS that your system can provide, giving it an upper limit to prevent it from saturating your disks. Therefore, you need to set this value, depending on the performance of the drives on your MySQL system. The default value is set to 100, way too low today's hardware, so you *absolutely* must tune this value. Here are some good starting points-

Drive Type	Value
Enterprise SSD	50,000
Single Consumer SSD	10,000
4 Drives in a RAID-10	5000
Single 7200RPM Drive	200

On our slaves, we use Enterprise-Level Micron SSD drives in a RAID-0, and we're able to set it as high as 150,000.

Don't know how your system stacks up? You can run a quick test with `fio` to determine the number of random IOPS your system can push. Let's pretend you want to test the disk `/dev/sdb1`, which is mounted as `/mysql` for this test.

First we need to install `fio` and setup the job file. The job we're defining will test random read+write I/O performance on a 128MB file. If you're on a system with faster drives, you'll want to increase the size to at least 5GB because the test will finish almost instantly. On a single 7200RPM drive system, the test takes about 30 seconds to run.

```
1 > apt-get instal fio
2 > cd /mysql
3 > mkdir /mysql/fio
4 > vi random-rw.fio
5     [random_rw]
6     rw=randrw
7     size=128M
8     directory=/mysql/fio
```

Now, run the job file that we just created.

```
1 > fio random-rw.fio
```

```

random_rw: (g=0): rw=randrw, bs=4K-4K/4K-4K, ioengine=sync, iodepth=1
Starting 1 process
Jobs: 1 (f=1): [m] [95.7% done] [978K/892K/s] [239/218 iops] [eta 00m:02s]
random_rw: (groupid=0, jobs=1): err= 0: pid=27015
  read : io=51,444KB, bw=1,142KB/s, iosps=285, runt= 45042msec
    clat (usec): min=75, max=260K, avg=3486.45, stdev=5546.46
    bw (KB/s) : min= 408, max= 2467, per=98.65%, avg=1126.60, stdev=322.94
  write: io=50,956KB, bw=1,131KB/s, iosps=282, runt= 45042msec
    clat (usec): min=1, max=10,261, avg= 8.84, stdev=91.79
    bw (KB/s) : min= 344, max= 2680, per=99.06%, avg=1120.37, stdev=382.68
  cpu : usr=0.16%, sys=1.07%, ctx=12868, mif=0, mif=24
  IO depths : 1=100.0%, 2=0.0%, 4=0.0%, 8=0.0%, 16=0.0%, 32=0.0%, >64=0.0%
    submit : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >64=0.0%
    complete : 0=0.0%, 4=100.0%, 8=0.0%, 16=0.0%, 32=0.0%, 64=0.0%, >64=0.0%
  issued r/w: total=12861/12739, short=0/0
  lat (usec): 2=0.02%, 4=7.30%, 10=27.85%, 20=13.02%, 50=1.51%
  lat (usec): 100=9.00%, 250=11.81%, 500=0.36%, 750=0.11%, 1000=0.23%
  lat (msec): 2=1.93%, 4=6.45%, 10=19.76%, 20=0.46%, 50=0.15%
  lat (msec): 100=0.02%, 250=0.02%, 500=0.01%

Run status group 0 (all jobs):
  READ: io=51,444KB, aggrb=1,142KB/s, minb=1,169KB/s, maxb=1,169KB/s, mint=45042msec, maxt=45042msec
  WRITE: io=50,956KB, aggrb=1,131KB/s, minb=1,158KB/s, maxb=1,158KB/s, mint=45042msec, maxt=45042msec

Disk stats (read/write):
  sda: ios=12755/2463, merge=13/1776, ticks=46190/2216420, in_queue=2422030, util=99.33%

```

Once the job finishes running, you should see something similar to the above output. The value underlined in yellow is your average random read IOPS (285 in this case), and the red underline shows the average write IOPS (282). For reference, this is from a single 7200RPM drive system.

It's worth noting that `fio` supports a bunch of different configuration options. There are a few different types of tests you can run, too. By changing the `rw=randrw` variable, you modify the type of benchmark that is performed. The different ones available to you are-

Name	Type of test
read	Only benchmarks sequential reads
write	Only benchmarks sequential writes
randread	Only benchmarks random reads
randwrite	Only benchmarks random writes
rw	Benchmarks sequential reads+writes
randrw	Benchmarks random reads+writes

Random I/O is the worst case scenario and is very slow, so using the value from the `randrw` test is a pretty conservative starting point for `innodb_io_capacity`, but it gives you an idea of how your system can perform. You can try running the job with `rw=rw` to see how your system performs in the best case, 100% sequential, scenario.



What different types of I/O does MySQL use?

- Writing to the InnoDB log file uses Sequential I/O.
- Writing to the InnoDB data files uses mostly Random I/O.
- Reading from the InnoDB data files uses mostly Random I/O.

innodb_read_io_threads & innodb_write_io_threads

InnoDB uses background threads to manage some reading and writing tasks. By default, it's set to 4, which is too low for modern systems. These should be set to $4 \times$ the number of CPU cores on your system.

query_cache_size

The MySQL query cache, at first glance, seems like a great idea. It stores the result of each query into memory, so if you issue the same query again, it can respond immediately- it doesn't even have to parse the query. If you have a heavily read table, it can really speed up the response of your queries, effectively replacing the need for a separate cache layer, such as Memcached.

The problem happens on writes, though. Since the query cache is designed to never serve you stale, out-of-date data, it needs to be able to intelligently update cached data. Unfortunately, that's not the way it handles the problem. Instead, every time there is a write to the database, the query cache completely wipes out and resets the cache for that entire table. **EVERY SINGLE TIME**. Think about that- if you have 100 INSERTS, UPDATES, or DELETES per second, the query cache needs to be flushed 100 times per second. It becomes all, but useless.

Not only is it useless, but it also hurts performance. The larger the cache, the longer it takes for it to be wiped, slowing down each INSERT query. Additionally, the query cache uses a single global lock- that is, only one thread can access the query cache at a given moment. This causes all of your other threads to block while waiting to wipe out the useless query cache.

Here's the skinny. **If your workload has any significant amount of writes, disable the cache altogether.** Remember- InnoDB has its own internal data cache without such constraints, so your queries will still be fast. **If you have tables that are heavily read from and are very rarely written to, the query cache makes sense.** In situations like this, it can actually save you from having to implement an entire cache layer, greatly reducing the complexity of your application.

Due to a bug in MySQL, you need to set both `query_cache_size` and `query_cache_type` as 0.

But, but! What if my application has *BOTH* types of tables? Since the query cache is set globally, and not per table, what do you do? You should probably disable the query cache, because it usually causes more harm than good. You *can* turn on the on-demand which, if used correctly, can provide some benefits.

First, you want to set your configuration values to turn on the on-demand cache.

```

1 query_cache_type=2
2 query_cache_size=16M

```

16MB is a good starting point for the size of your query cache. Remember, you want to keep it very small because it needs to be wiped out on every single write. And it's only storing the queries + results, which are often only a few KB each.

Now, when you issue a query, by default, it will skip the query cache entirely. If you want a specific query to use the cache, you need to change `SELECT` to `SELECT SQL_CACHE`. For example-

```

1 mysql> SELECT SQL_CACHE FROM foobar WHERE id = 123456

```

However, the on-demand query-cache still suffers from the global write that's described above, so it's far from the perfect solution.

XtraDB/InnoDB vs MyISAM

You'll notice that I've not touched on any MyISAM-related performance settings in this Chapter. That's because you shouldn't be using MyISAM in production ever. Why shouldn't you use MyISAM?

It doesn't support row-level locking

So what? That means that every insert into MyISAM locks the entire table, making it unavailable for other concurrent connections to read or write from the table until the query is complete. That's a pretty huge limitation. InnoDB supports row-level locking, so only the rows involved in an insert or update get locked while they are being modified.

If you're in a low-volume environment, it, of course, doesn't make a big difference, but MyISAM is absolutely not suited for a high-volume workload and has no place when scaling. Switch everything to InnoDB and don't look back.

How can you tell if a table is MyISAM or InnoDB?

```

1 mysql> SHOW TABLE STATUS WHERE 'Name' = 'foobar' \G
2 *************************** 1. row ****
3           Name: foobar
4      Engine: InnoDB
5     Version: 10
6   Row_format: Compact

```

How can you switch the table from MyISAM to InnoDB?

```

1 mysql> ALTER TABLE foobar ENGINE = INNODB;

```

Warning: The alter table will probably take a long time and put a write-lock on the table, preventing data from being written to it.

What about searching? It's true that InnoDB doesn't support full text searching like MyISAM. Truthfully, there are far better alternatives for searching, anyways. Check out:

- [ElasticSearch⁴⁵](http://www.elasticsearch.org/)
- [Sphinx⁴⁶](http://sphinxsearch.com/)
- [Solr⁴⁷](http://lucene.apache.org/solr/)

Rumor has it that MySQL 5.6 will add full text searching to InnoDB. I haven't tested it yet, so I can't comment on the performance, but it'll likely replace or at the very least reduce the need for a separate search service.

Tuning Linux for an intensive database

Linux and MySQL are a match made in heaven. Out of the box, MySQL will run pretty decently on a Linux system but we can squeeze out 30-40% more performance (mostly on the I/O side of things, our weakest link!), just by tuning some settings and making the right decisions from the start.

Changing the I/O Scheduler

The Linux kernel ships with three I/O schedulers- they are `cfq`, `deadline`, and `noop`. You can see which one you're using with the following command, where `sda` is the device id of your disk.

```
1 cat /sys/block/sda/queue/scheduler
2 noop deadline [cfq]
```

On most distros, it's set as `cfq` by default, which is supposed to provide fair I/O sharing across the entire system, great for desktops where you have a bunch of different programs playing grabby hands with I/O. But you don't have lots of different programs; you're running a monolithic database server and you want your database to have all of the I/O resource available.

It's not so much which I/O scheduler you choose, `deadline` or `noop` both work swell, it matters which one you don't choose. `cfq` is extremely slow and you'll see massive gains by switching. `noop` is supposed to be a better choice for SSDs and Hardware RAID, but in my testing, both `deadline` and `noop` performed similarly.

⁴⁵<http://www.elasticsearch.org/>

⁴⁶<http://sphinxsearch.com/>

⁴⁷<http://lucene.apache.org/solr/>

```

2. root@slave01: ~ (ssh)
16.77 0.00 5.84 6.35 0.00 71.03

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrrq-sz avgqu-sz await r_await w_await svctm %util
sda 0.00 81.00 1755.00 120.00 28068.00 820.00 30.81 6.97 3.83 3.98 1.70 0.53 100.00
avg-cpu: %user %nice %system %iowait %steal %idle
17.34 0.00 6.03 6.53 0.00 70.10

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrrq-sz avgqu-sz await r_await w_await svctm %util
sda 0.00 78.00 1754.00 127.00 28052.00 868.00 30.75 7.63 3.96 4.13 1.64 0.51 96.80
avg-cpu: %user %nice %system %iowait %steal %idle
21.25 0.00 6.50 8.00 0.00 64.25 (above) CFQ (below) noop

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrrq-sz avgqu-sz await r_await w_await svctm %util
sda 0.00 78.00 2562.00 124.00 40992.00 820.00 31.13 0.74 0.28 0.29 0.06 0.21 56.80
avg-cpu: %user %nice %system %iowait %steal %idle
8.42 0.00 4.46 2.42 0.00 84.69

Device: rrqm/s wrqm/s r/s w/s rkB/s wkB/s avgrrq-sz avgqu-sz await r_await w_await svctm %util
sda 0.00 84.00 1084.00 136.00 27344.00 900.00 29.91 0.37 0.30 0.33 0.09 0.19 23.60

```

3. root@slave01: ~ (ssh)

```

root@slave01:~# echo "noop" > /sys/block/sda/queue/scheduler
root@Slave01:~#

```

Looking at the benchmark above, the top window is running the command `iostat -x 1`, which displays information on the I/O usage every second. Above the white line drawn through the middle, we're running with the `cfq` scheduler. You can see, highlighted in the red column that I/O usage is hovering at around 100%. After 2 seconds, I switch over to the `noop` scheduler. I/O utilization **immediately** cuts in half and hovers around 20%.

The section highlighted in the yellow square just highlights that the read per second and write per second workload stays about the same during the entire test, so the drop in I/O utilization is because of the change in scheduler.

Switching the I/O scheduler is extremely easy and it's extremely low hanging fruit for increasing your database write performance in seconds.

The main disk that my database uses is `/dev/sda`, which is the disk that I'm changing the I/O scheduler for in the example below. If your disk has a different identifier, change `sda` to that identifier in the examples below. `sda` is the right choice for most people.

```

1 > echo noop > /sys/block/sda/queue/scheduler
2 > vi /etc/rc.local
3 # Add this line to your rc.local file because the
4 # I/O scheduler will revert to the default after a
5 # system reboot.
6 echo "noop" > /sys/block/sda/queue/scheduler

```

Tuning swapiness

Swapping usually means the death of a MySQL server. Since MySQL does its own internal caching, it expects the internal buffer pool to always be fast. When Linux swaps out some of

this buffer pool to the disk, it pulls the rug out from under MySQL and will often crash the server when it runs out of memory. It's extremely important to configure MySQL so that you maximize your memory usage without going so high that you can eventually cause the server to swap.

We can configure the `vm.swappiness` setting to tell the kernel that it shouldn't swap inactive areas of memory to the disk. However, keep in mind that if you disable swap completely the server will most definitely crash in an out-of-memory situation, since it doesn't have the luxury of using a disk as a swap space.

```
1 > sysctl -w vm.swappiness=0
2 > vi /etc/sysctl.conf
3 vm.swappiness=0
```

Increasing the number of open files

As we discussed in the Load Balancer Chapter, on a stock-Linux configuration, the maximum number of open files allowed per process is set very low (1024). For MySQL, we need to increase the number of open files for two reasons. The first is to make room for the `max_connections` setting, since each connection eats up one file descriptor. The second reason is that each open table requires a file descriptor! If you have 10 tables and 100 connections, you'd need *at least* 1000 open files since each connection thread maintains its own file handles.

Luckily, it's an easy to setting to change.

```
1 > vi /etc/security/limits.conf
2      * soft  nofile  999999
3      * hard  nofile  999999
4 > ulimit -n 999999
```

Turning off file system access times

As we discussed in the application server chapter, by default, in most Linux distributions, the file system keeps track of the last time a file was accessed or read. It's rarely useful to have this information and it causes an I/O operation every single time a file is read, such as MySQL reads from a table. We can disable this behavior by modifying the partition where your MySQL database files are stored. Open up `/etc/fstab` and look for your main partition. Mine looks like this:

```
1 /dev/sdb1      /      ext4      errors=remount-ro 0 1
```

Change it to

```
1 /dev/sdb1      /      ext4      noatime,nodiratime,errors=remount-ro 0 1
```

`noatime` tells the file system to not track when individual files are read, and likewise, `nodiratime` tells the file system to not track when directories are read from.

You can remount the file system (without a reboot!) with the command `mount -o remount /dev/sdb1` (replace `/dev/sdb1` with whatever file system that you had to edit in your `/etc/fstab` file).

Picking the best file system

Ahh... the database file system wars. There are plenty of options and a thousand opinions available on the internet. Let's talk about hard numbers- I've compared and benchmarked the two most recommended MySQL file systems, `ext4` and `XFS`. We use `ext4` on Twitpic, and from the research I've done, I expected it to perform equally as well as `XFS` - not to mention, `ext4` is the default file system choice for most Linux distros nowadays.

Anyways, using `sysbench` (covered at the end of the chapter), I performed a read/write MySQL benchmark comparing `XFS` to `ext4` and the results surprised me. I ran the test on a 16-core machine to make sure that benchmark wouldn't become CPU bound and skew the results. It created a table that was a bit larger than the `innodb_buffer_pool_size`, so not all of the data could be fit into memory, forcing it to hit the disk for some reads.

Comparing `ext4` to `XFS`, I found that `XFS` was extremely more performant on a benchmark of 100,000 read **and** write queries.

XFS	ext4
21,755 qps	16,560 qps

By the way, to install `XFS` on Ubuntu, you just need to install `xfsprogs` and run `mkfs.xfs` on the device you want to format.

```

1 > apt-get install xfsprogs
2 > mkfs.xfs /dev/sdf1

```

The tests were done using a dedicated disk for the MySQL data files, so there was no I/O competition from other system processes.

Raw Device Mode

You can even use InnoDB in something called **Raw Device Mode**, where it writes data directly to an unformatted disk or partition without using any file system at all. I highly recommend not using it, even though it's slightly faster, because it makes managing your data a complete nightmare. You lose all visibility in the database files that you would otherwise have when using a normal file system. Additionally, benchmarks show that `ext4` and `XFS` are only a tiny bit slower than using a raw device.

For the curious, you can use raw device mode by changing the `innodb_data_file_path` setting-

```

1 innodb_data_home_dir=
2 innodb_data_file_path=/dev/sda1:500Gnewraw

```

Restart MySQL, it will create the data file, then change `innodb_data_file_path` again.

```

1 innodb_data_file_path=/dev/sda1:500Graw

```

The future of file systems

It's worth quickly mentioning [ZFS](#)⁴⁸ and [btrfs](#)⁴⁹. Both are pretty cool and likely the future of file systems, because they support modern features like copy-on-write, SSD caching, and deduplication, but ZFS doesn't have great support on Linux yet and Btrfs is still too experimental to use in production.

Load balancing MySQL Slaves

Having multiple read-only MySQL slaves are great, but you need a way to load balance them in your code. By default, PHP doesn't handle this for you at all, and it's something you need to code in yourself. I'm going to show you a few ways to do it below.

Database Class

The easiest way is to do it within code using a simple database class. Select a slave at random and return a database connection depending on the one you chose.

```

1 <?php
2 class Database {
3     static function connect() {
4         // Assuming your slave hosts come from a configuration file, but
5         // for brevity, we just define them here.
6         $slaves = array('192.51.100.1', '192.51.100.2', '192.51.100.3');
7         $slave = $slaves[array_rand($slaves)];
8
9         return new mysqli($slave, 'user', 'password');
10    }
11 }
12
13 $slave = Database::connect();

```

The downside of handling slave selection this way, and doing it in code in general, is that if one of your slaves crash, it will not be handled gracefully while it waits for the connection to timeout. Of course, you can remove the slave from your configuration manually when it goes down, but that's a manual process and I'm not a huge fan of waking up in the middle of the night to update configuration files.

⁴⁸<http://en.wikipedia.org/wiki/ZFS>

⁴⁹<http://en.wikipedia.org/wiki/Btrfs>

Extending PDO

Similarly to the above, if you're using PDO, you can handle load balancing a bit more elegantly, but with the same downsides as the Database class used in the previous section (ungraceful handling of down slaves).

```

1  <?php
2  class SlavePDO extends PDO {
3
4      public function __construct($dsn, $user=null, $pw=null, $options) {
5          // Assuming your slave hosts come from a configuration file, but
6          // for brevity, we just define them here.
7          $slaves = array('192.51.100.1', '192.51.100.2', '192.51.100.3');
8          $slave = $slaves[array_rand($slaves)];
9
10         $dsn .= ";host={$slave}";
11         parent::__construct($dsn, $user, $pw);
12
13     }
14
15 }
16
17 $slave = new SlavePDO('mysql:dbname=test', 'user', 'password');

```

Using the mysqlnd_ms plugin

The *MySQL Native Driver* is an alternative MySQL driver written by the PHP team that was first included in PHP 5.3. It was added to remove PHP's dependency on `libmysql` and tie MySQL further into the core of PHP. The MySQL Native Driver also added in the ability for developers to create both PHP and C plugins to add functionality PHP's MySQL Driver. Since MySQLND is used internally, MySQLND plugins work with any interface that you use to interact with MySQL- PDO, MySQLi, and even the old-school, deprecated, unrecommended [MySQL extension](#)⁵⁰.

First, you need to install the `mysqlnd_ms` PECL plugin.

```
1  > pecl install mysqlnd_ms
```

Next, add a configuration file to your app. For the example, let's say it's stored in `./config/db.conf`

⁵⁰<http://us1.php.net/manual/en/book.mysql.php>

```

1  > vi ./config/db.conf
2  {
3      "app" : {
4          "master": [ { "host": "192.51.100.10" } ],
5          "slave": [
6              { "host": "192.51.100.1" },
7              { "host": "192.51.100.2" },
8              { "host": "192.51.100.3" }]
9      },
10     "failover": { "strategy": "loop_before_master" }
11 }
```

And then enable `mysqlnd_ms` in your application.

```

1 <?php
2 ini_set("mysqlnd_ms.enable", 1);
3 ini_set("mysqlnd_ms.config_file", "./config/db.conf");
```

Now, whenever you want to make a MySQL connection (this works with both PDO and mysqli), you just pass in `app` as the host.

```

1 <?php
2 $mysqli = new mysqli("app", "username", "password");
```

The cool thing here is that not only will it round robin between your slaves, it will also send any write queries to your Master database. The failure handling is still sub-optimal, though.

The plugin lets us set a failure strategy in the configuration. In the example, I've set it as `loop_before_master`, which means that it will try the other slaves and then try the master when a slave fails. This is great, but it doesn't remember failed slaves between requests, so you'll still pay the cost of waiting for the slave to timeout, every single time. This is, however, better than just outright failure.

Using HAProxy

The best way to handle slave load balancing is to use a software load balancer like HAProxy. As discussed in Chapter 4, HAProxy can do Layer 4 load balancing, i.e., raw TCP sockets. While most people use this functionality load balancing HTTP, it can be used equally as well for internally load balancing your MySQL connections as well.

Why is HAProxy the best solution? For two reasons-

1. When it detects a database slave as unresponsive (crashed, down, whatever), it removes it from the load balancer pool and stops using it. It does this detection quickly and is completely transparent. No configuration changes, no downtime, no waiting for connections to timeout, and, **best of all, no waking up in the middle of the night**.
2. HAProxy lets you set very granular timeouts down to the millisecond. This lets you side step the poor MySQL timeout management in PHP (which is limited to 1 second, minimum) and set more reasonable timeouts.

The installation and setup is pretty similar to how we setup our HTTP Load Balancer.

```
1 > apt-get install haproxy
2 > vi /etc/haproxy/haproxy.cfg
3
4 global
5 maxconn 50000
6 user haproxy
7 group haproxy
8 stats socket /tmp/haproxy
9 node lb1
10 nbproc 1
11 daemon
12
13 defaults
14 log global
15 retries 3
16 option dontlog-normal
17 option splice-auto
18 timeout connect 200ms
19 timeout client 2s
20 timeout server 2s
21 maxconn 50000
22
23
24 listen mysql 192.51.100.100:3306
25 mode tcp
26 balance leastconn
27
28 server slave01 192.51.100.1:3306 check
29 server slave02 192.51.100.2:3306 check
30 server slave03 192.51.100.3:3306 check
31
32 listen stats *:1936
33 mode http
34 stats enable
35 stats uri /
36 stats hide-version
37 stats auth Username:Password
```

HAProxy won't run until you enable the init.d script, so let's do that now.

```
1 > vi /etc/default/haproxy
2     # Set ENABLED to 1 if you want the init script to start haproxy.
3     ENABLED=1
4
5 > service haproxy start
```

With this configuration, we set up three database slaves, each defined with a `server` line. Now, all we have to do is modify our code to connect to HAProxy's IP Address instead of the slave IP Address. (The HAProxy IP is defined in the configuration file in the `listen` setting. In this example, it's `192.51.100.100`.)

```
1 <?php
2 $mysqli = new mysqli("192.51.100.100", "username", "password");
```

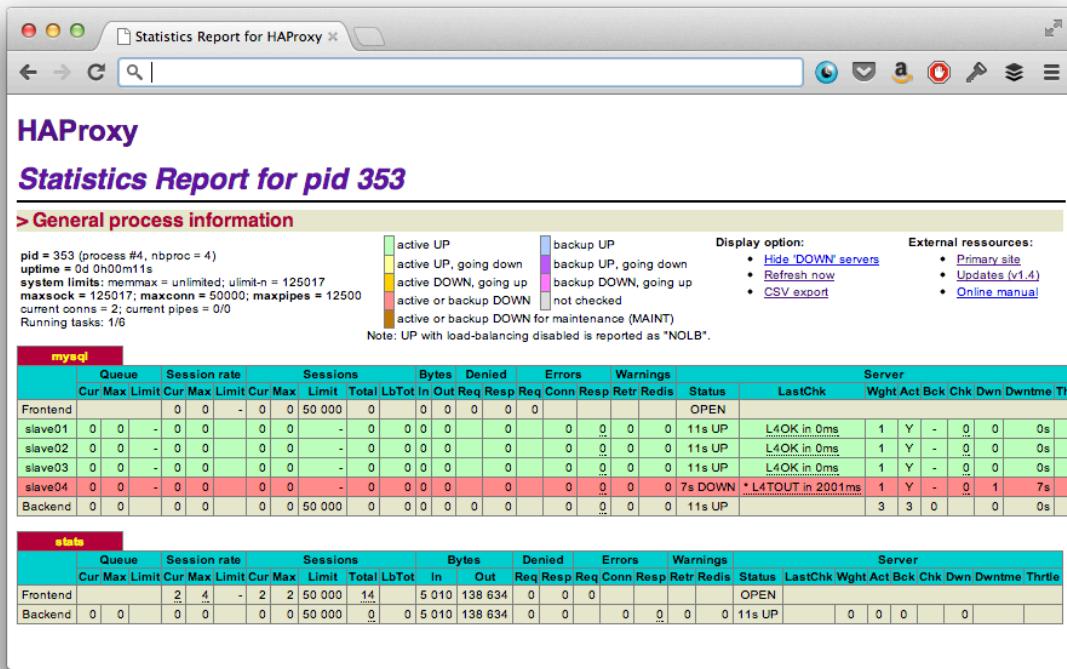
Now, every time you make a MySQL connection, you will connect to HAProxy instead of directly to the MySQL slave. HAProxy will choose the slave with the least amount of connections and connect you to that slave. You can also use `balance roundrobin` as your balancing strategy if you want HAProxy to use less intelligence when picking slaves.

Since we set `timeout connect` as `200ms`, if a connection to a slave fails to happen in less than 200 milliseconds, HAProxy will try another slave instead. Failed or overloaded MySQL slaves are now a non-issue.

With `timeout client` and `timeout server` both set at `2s`, it means that a long-running client or hung query will get disconnected after 2 seconds. Having a very low query timeout threshold makes sense for a web application, but obviously not for any kind of analytics or background processes, so you'll want to connect directly to the slaves or set up a secondary HAProxy `listen` block with different timeouts for those scenarios.

The best feature, in my opinion, is that failed slaves are automatically removed from the pool and re-added once they come back up. Until we ran this setup in production on Twitpic, I'd have to wake up in the middle of the night to manually remove broken slaves from our configuration files. It was a real pain-point for us as we grew to more and more slaves.

You also get a really nice web interface for viewing the status of your MySQL slaves.



Accepting the Master as a SPOF

When you choose to use MySQL, you need to learn to accept and understand that your master server is always going to be a single point of failure. Of course, there are solutions and workarounds like [DRBd⁵¹](#), [Percona XtraDB Cluster⁵²](#), and MySQL Master-Master Replication. The downside, though, is that they are often slow and extremely complex.

The alternative is to accept and deal with the fact that your master database server is going to be a single point of failure. You can work around the problem by scripting the failover process and dealing with the situation with as little downtime as possible. Check out [JetPants⁵³](#) by Tumblr as a starting point for a MySQL master failover/slave promotion toolkit.

What really happens when your master fails?

I used to be worried that if our master database crashed, we would lose data and be faced with an enormous amount of downtime. When I thought about it, though I realized a few things-

The site won't go down

Since we only write data to the master, and never read from it, it's really just a glorified coordinator of data. If it crashes, our site just goes into a read-only state, but it's still up and functioning.

⁵¹<http://dev.mysql.com/doc/refman/5.0/en/ha-drbd.html>

⁵²<http://www.percona.com/software/percona-xtradb-cluster>

⁵³<http://tumblr.com>

We won't lose data

Because we make daily, off-site backups, even in the event of a catastrophic failure, losing just the master is even less of a big deal because data is replicated and stored safely on all of the slaves.

Using [semi-synchronous replication](#)⁵⁴ in MySQL 5.5 is needed to guarantee that we don't lose *anything* in the event of a master crash. With the default asynchronous replication, it's possible to lose data that has been written to the master but not yet replicated to the slaves during a crash. Semi-synchronous replication, instead, guarantees that new data is replicated to at least one slave.

Promoting a slave as the master

Okay, so your master database has failed. What is next? Well, we need to promote a slave to be the new master to get it up and running as quickly as possible.

Let's quickly go through a high level overview of the slave to master promotion process.

Assumptions

- Your master server is named `master01`, and you have five slaves, `slave01-05`.
- You want to promote `slave01` to be the new master.
- `master01` is still up-and-running.
- Your code writes to the MySQL using the database user `app_user`.

If `master01` is still up-and-running, it's important to disable writes to it before failing over to the new master so your data doesn't get out of sync. On `master01`, first delete the user that your application uses (`app_user`), so it won't be able to establish any new connections. Next, put the database in read-only mode:

```
1 > DROP USER `app_user`;
2 > SET GLOBAL read_only=1;
3 > FLUSH TABLES WITH READ LOCK;
```

At this point, your database cluster is in a total read-only mode. If `master01` has crashed or become unavailable, this step is unnecessary because your cluster is already effectively in a read-only state.

Next, we need to pick a slave to be the new master. In this scenario, we've decided to promote `slave01`. On `slave01`, you'll want to copy over your `my.cnf` from `master01`, particularly the `log-bin` setting, and restart MySQL. It's helpful to have a backup master server already configured for this purpose.

On `slave01`, you'll want to run `SHOW MASTER STATUS` to determine the binlog position.

⁵⁴<http://dev.mysql.com/doc/refman/5.5/en/replication-semisync-installation.html>

```

1 mysql> show master status \G
2 *************************** 1. row ***************************
3 File: master-bin.000001
4 Position: 98

```

Now that you know the position of the binlog, you can connect to the rest of your slaves and reconfigure them to use `slave01` as the new master server with the `CHANGE MASTER TO` command.

```

1 mysql@slave02> STOP SLAVE;
2 mysql@slave02> CHANGE MASTER TO           MASTER_HOST="slave01",
3                   MASTER_LOG_FILE="master-bin.000001", MASTER_LOG_POS=98;
4 mysql@slave02> START SLAVE;

```

Lastly, update your application configuration to write to `slave01` instead of `master01`.

Doing it manually isn't the best scenario, and a good portion of this can be scripted. Instead of scripting it yourself, though, it's worth checking out [mysql-master-ha](#)⁵⁵ which can handle automatic master failover by scripting the solution described above.

Further research and reading

The topic of MySQL Master Failover is huge and could easily take up its own book, so it's impossible for me to cover all of the possible solutions here. For most people, taking it into a read-only state and manually promoting a new master is the simplest and best approach. Unfortunately, it's not a solution that works for everyone, so here are some further places to look for more information on some more robust (and more complex) solutions for handling master failure.

- [Automated Master Failover \(Slides\)](#)⁵⁶
- [How to evaluate which MySQL HA solution best suits you](#)⁵⁷
- [MySQL ZFS Replication](#)⁵⁸
- [MySQL DRBd Replication](#)⁵⁹
- [Percona XtraDB Cluster](#)⁶⁰
- [Percona Replication Manager](#)⁶¹
- [Tungsten Replicator](#)⁶²

⁵⁵<http://code.google.com/p/mysql-master-ha/>

⁵⁶<http://www.slideshare.net/matsunobu/automated-master-failover>

⁵⁷<http://www.percona.com/live/mysql-conference-2012/sessions/how-evaluate-which-mysql-high-availability-solution-best-suits-you>

⁵⁸<http://dev.mysql.com/doc/refman/5.0/en/ha-zfs-replication.html>

⁵⁹<http://dev.mysql.com/doc/refman/5.0/en/ha-drbd.html>

⁶⁰<http://www.percona.com/software/percona-xtradb-cluster>

⁶¹<https://github.com/jayjanssen/Percona-Pacemaker-Resource-Agents/blob/master/doc/PRM-setup-guide.rst>

⁶²<http://code.google.com/p/tungsten-replicator/>

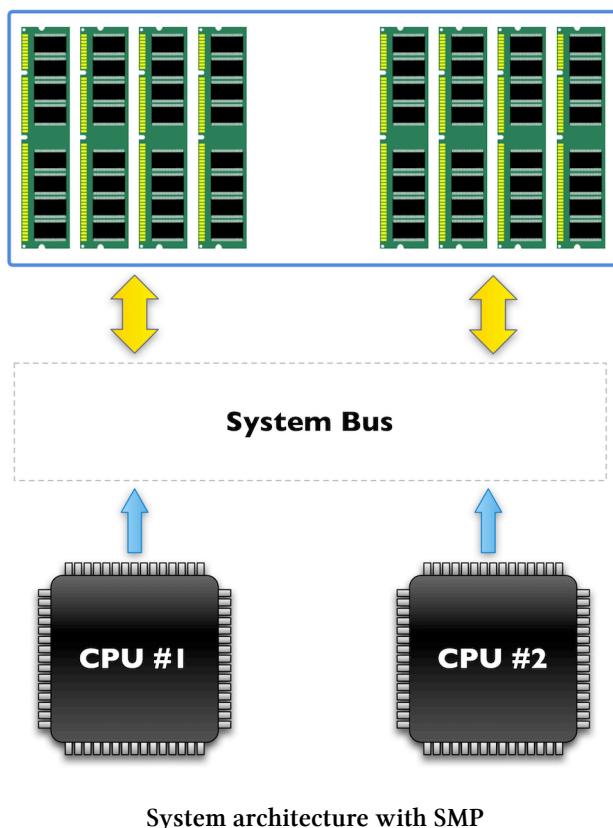
Understanding issues with NUMA

TL;DR- NUMA (Non-Uniform Memory Access⁶³) is pretty much a complex topic and only involves a subset of MySQL implementations: Servers running *multiple* Intel Nehalem or newer CPUs, with large amounts of Memory (i.e., over 64GB) and a very large `innodb_buffer_pool_size`. What happens is MySQL swaps out large amounts of memory even though there may be plenty of free memory on the system, causing MySQL performance to drop dramatically.

While the problems around NUMA architecture haven't been completely solved, you can fix most of the issues by patching `/usr/bin/mysqld_safe`, running Percona Server 5.5, and setting `zone_reclaim_mode` to 0.

What exactly is NUMA?

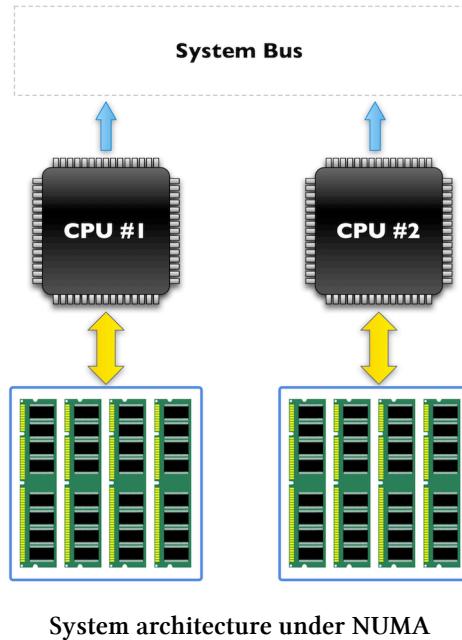
Before Nehalem CPUs, most multi-CPU systems used a SMP architecture which gave both CPUs equal access to all of the system memory.



Starting with Intel's Nehalem architecture, Nehalem processors and newer processors (Sandy Bridge and Ivy Bridge included), use a new method of accessing memory in multiple CPU systems. It's worth noting that this only matters on systems with *multiple physical CPUs*, not single CPU systems with multiple cores.

⁶³http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access

Anyways, described simply- with NUMA, instead of having a central access path to memory, each CPU now has direct access to some local memory banks. This ties specific sticks of memory to each CPU and makes accessing non-local memory slower and more complex.

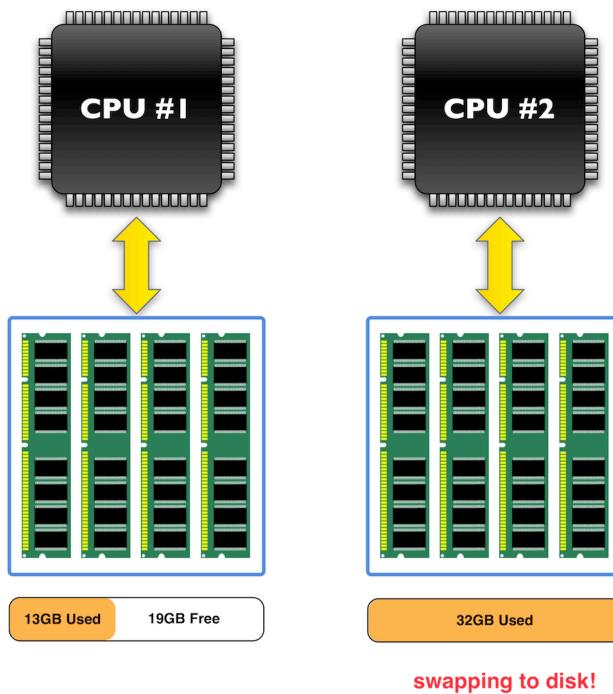


System architecture under NUMA

How does NUMA impact MySQL?

Let's imagine a scenario- You have a huge MySQL system with 2 Physical CPUs and 64GB of memory. Obviously, to take advantage of your 64GB, you set your `innodb_buffer_pool_size` to be 57GB. You run your system under heavy load and notice that even though there is plenty of free system memory, MySQL is performing very poorly because it's swapping memory to the disk. At first glance, this doesn't make any sense.

What's happening is that Linux doesn't let each CPU address the full 64GB- instead, each CPU can only work with its "local" pool of memory- 32GB in this case. As soon as a single CPU needs to access more memory than is free in its local pool, Linux starts swapping to the disk even if there is free memory in the other CPU's local pool.



What happens when a CPU needs to address more than 32GB

How can we overcome this problem?

There are a couple of ways that we can overcome this serious limitation.

1. Set `innodb_buffer_pool_size` to 32GB.
2. Only use MySQL servers with a single physical CPU.

Obviously, these are both horrible solutions. Luckily, there is a better solution. The first step is to use Percona Server 5.5 instead of the stock MySQL Server from Oracle. The version from Percona includes numerous enhancements and patches that make MySQL a bit saner on NUMA systems.

The second part of the solution is to tune Linux to allow memory interleaving across the CPUs, effectively removing the 32GB limit from each CPU and allowing it to work with the full 64GB.

To do this, we need to install `numactl`.

```
1 > apt-get install numactl
```

Next, we have to patch the MySQL startup script `mysqld_safe` to use `numactl`.

```
1 > vi /usr/bin/mysqld_safe
```

Once the file is open, we need to replace this line (line 735 on my system):

```
1 cmd="`mysqld_ld_preload_text`$NOHUP_NICENESS"
```

with a line that uses numactl:

```
1 cmd="/usr/bin/numactl --interleave all $NOHUP_NICENESS"
```

Further Reading

A great article on NUMA and MySQL is [The MySQL “Swap Insanity” Problem and the Effects of the NUMA Architecture](#)⁶⁴ by Jeremy Cole. It's pretty long, but an interesting read, if you want to really understand the problem and the solution.

Choosing the best hardware

If you're going to spend money on hardware, putting it into your MySQL cluster is almost always going to give you the biggest performance increase.

When choosing a CPU, there are a couple of things to keep in mind.

CPU Speed (impacts a single query)

MySQL executes each query in a single thread. It does not parallelize your query, so a single query can only take advantage of one CPU core. So, for single query performance in a purely CPU-bound environment, more GHz is always going to win over more cores. Keep in mind though that unless you can fit all of your data into memory, MySQL is rarely CPU-bound.

CPU Cores (impacts concurrency)

MySQL can't split up a single query across multiple cores, but it can run multiple queries concurrently and having more cores gives you more concurrency. In a high-volume environment, more cores are better.

Right now, the best CPU on the market is the Intel Xeon SandyBridge 26xx Series. You can put up to two CPUs per box, each of which can have up to 8-cores each. That gives you 32 concurrently executing threads with Intel's Hyper Threading. On top of that, they make them as fast as 3.5GHz per core.

Memory

This one is easy. Get the most amount of memory that you can possibly cram into your motherboard. As long as you have more data than memory, there is no reason (other than cost) why you should avoid maximizing the memory in your database servers. The more memory you have, the bigger you can set `innodb_buffer_pool_size`.

⁶⁴<http://blog.jcole.us/2010/09/28/mysql-swap-insanity-and-the-numa-architecture/>

We run ours with 64GBs per slave, but 128GB, 256GB, or even 512GB in a box isn't unheard of nowadays. At the time of writing, a 16GB stick of ECC/Registered DDR3 runs around \$175, so *even 512GB only costs around \$5500*.

If you go with the Intel Xeon 26xx CPUs, SuperMicro even makes a motherboard with enough sockets to fit 768GB. Crazy times we live in.

Storage

Making the right storage decisions is very important to the performance of a MySQL cluster. Not all storage is built the same and a master/slave setup means that you have vastly different storage requirements. What works on your master database server won't necessarily work well for your slave server, and vice-versa.

Hard Drives

I highly recommend using high-end server hard drives for your master database. Grab a couple of 10,000 or 15,000 RPM drives and put them into a RAID (discussed below).

Why not go for SSDs on your master? Remember what we talked about when sizing out `innodb_log_file_size`? For writes, MySQL can use mainly sequential I/O by taking advantage of a correctly sized log file. Sequential I/O is fast, even on rotational disks, so for a write-only master database server, it's not really necessary to pay the price for SSDs.

Put your Master Server's disks into a RAID! Don't play with fire. Your drives will fail, usually at the worst time, and will cause downtime.

Don't use RAID-5! The cost/GB is tempting in a RAID-5 but it's too slow to use on a high-volume database server. If you're working on the cheap, use a RAID-1.

Do use a RAID-10 RAID-10 is combination of RAID-1 and RAID-0. RAID-0, by itself, offers the best performance but is dangerous because losing a single disk will cause downtime. In RAID-10, two RAID-0's are mirrored with RAID-1. Losing a disk won't cause any downtime.

RAID-10's downfall is that it has a bad cost/GB ratio. For example, four 600GB drives in a RAID-10 (2.4TB of raw space) will only yield a 1.2TB RAID array.

Use hardware RAID Controller Don't mess around with Software RAID. It's not worth it. Investing in hardware RAID for your master server is the right move, from both a speed and durability point of view. Why hardware RAID?

- RAID cards have internal memory (often around 512MB) that is used to cache data before writing it to a disk. This makes `fsync` (and subsequently, any write to a disk) very fast.
- Good RAID cards offer a battery or capacitor backup, which guarantees that the data inside of the internal memory will get written to the disk after loss of power.

I really like the [Adaptec 5405Z RAID card](#)⁶⁵. It's PCI-Express, has a 512MB write cache + 4GB flash cache, can connect up to 256 drives, and works great with Linux.

Solid-State Drives

For your slave MySQL servers, SSDs are where it's at. Your slaves are going to be handling an unrelenting workload of random reads- which rotational disks are very bad at. On the other hand, SSDs are several orders of magnitude faster than rotational disks are at random I/O. It's not uncommon for an SSD to push 30,000 random IOPS while your typical hard drive can barely do 200 IOPS with the same workload.

The downside of SSDs is that they're expensive. The prices are coming down, especially in the consumer market, but price per GB is still high.

One consideration to make when looking at SSDs is that the performance between brands and products can be **VASTLY** different. You can't compare just GBs anymore, you have to look at the raw IOPS performance. Some companies make 256GB SSDs with 5,000 IOPS and others with 500,000 IOPS.

Flash Wear Due to the underlying technology that's used to make SSD drives, it's possible to wear them out if you do heavy writing to them. The topic of wear endurance is pretty controversial. Depending on the manufacturer and type of flash-memory used, it's hard to wrap your head around it and get a definitive answer.

My conclusion: *I don't worry about it.* Our slaves are replaceable and with our HAProxy load-balancing setup, a crashed slave doesn't create any downtime.

If you're worried about it, one way to mitigate the problem is to write your InnoDB Log Files and Binary Logs on a separate, non-SSD drive. You can change the location of the log file location with the `innodb_log_group_home_dir` setting and similarly, the location of the Binary Log can be changed with the `log_bin` setting. Because the binary log files store temporary data and only depend on sequential writes, non-SSD drives can be fast enough, without wearing down your flash drives.

If you're really worried about flash wear, the easiest solution is to look for the most durable types of flash drives. The biggest differentiator between flash storage is the type of storage cells used- **SLC** (Single Layer Cell), **MLC** (Multi Layer Cell), and (less commonly) **TLC** (Three Layer Cell).

SLC Flash Storage can only hold one bit per cell, while MLC holds two bits per cell and TLC can hold three. MLC and TLC have a capacity advantage because they can store more data in a single cell, increasing storage density and reducing costs.

The downsides to MLC and TLC, though, is that with more "states" represented in a single cell, it needs to be more exact. Everyone loves the glass-of-water analogy here, so let's use that.

Imagine that a cell of flash storage was represented by a glass of water. If it was an SLC cell, and only had to store one bit of data, you could do that pretty easy. An empty glass would be `0` and a full glass would be `1`. Even if the glass got inaccurate over time, and a little water was leftover

⁶⁵<http://www.amazon.com/dp/B002DYAUTC?tag=stevecorona-20>

when emptying it, you could still assume that it's empty (0), if there was a little bit of water left in the bottom.

Now imagine if the glass of water was an MLC cell. Now you need to represent 2 bits in a single cup. How would you do that? Well, maybe an empty glass would be 00, 1/4 full would be 01, 1/2 full would be 10, and completely full would be 11. It's doable, but your measurements need to be more precise—if there was a little water leftover in the empty cup, you might get confused whether or not it's empty (00) or 1/4 full (01). Add in TLC, needing to have 3 bits in a single cell, and the need for precision gets even more demanding.

So, if flash wear is a big deal for you, SLC is the way to go. You'll pay for it though, it can be up to 5x more expensive and typically tops out at a lower drive capacity.

Cell Type	Lifespan per Cell (number of erase cycles)
SLC	100,000
MLC	10,000
TLC	5,000

RAID It's not necessary to use RAID on your slave servers (since we don't need the drives to sustain failure—we just replace the drive and rebuild the slave), but you can use it to increase your disk space or improve performance.

It's often cheaper to buy 2x smaller high-end SSDs than 1x larger high-end SSDs. For example, if you need 400GB of SSD storage, you could either buy a single 400GB drive or 2x 200GB drives and put them in a RAID-0.

You can also use this technique to squeeze more IOPS out of your SSDs. Two SSDs in a RAID-0 will offer almost 2x the number IOPS when compared to a single drive.

Network Card

Wait, really? Having a fast network card is important on a MySQL server? Yeah it is! Our MySQL slaves can easily push out over 100mbps of traffic during peak hours, so it's important to get the fastest ones your network can support (1000mbps in our case). Additionally, I recommend bonding the cards together to handle network failure and increase throughput.

What if you don't have as much MySQL traffic? I still recommend getting the fastest network cards that your network can support. You'll thank me when you have to restore a slave with your 500GB MySQL backup. On gigabit, that's only about an hour but over 12 hours on your standard 100mbps LAN.

Online Schema Changes

A big weakness of MySQL is that it's very difficult to make schema changes (add/remove columns or indexes) on tables when they get very large.

The reason for the slowness is twofold—MySQL has to scan and modify every row in the database to add a new column. It does this by copying the data into a temporary table, which causes high

I/O usage and takes a significant amount of time (hours to days on hundred-million row tables that use hundreds of GBs of disk space). On top of that, while the data is being copied, MySQL holds an exclusive write-lock on the table, meaning that no new data can be added or changed in the table during this (extremely) long process.

This limitation is one of the reasons why companies are moving towards schema-less databases like MongoDB and Cassandra - you can add columns at will without having to painstakingly alter the database.

Reddit has talked about how they work around this issue by structuring their database differently⁶⁶. This is a solution I really dislike, but what they essentially do is turn MySQL into a key-value store.

Imagine you had a normalized database that looked like this:

id	username	name	location
1	steve	Steve Corona	Charleston, SC
2	rob	Robert Johnson	New York, NY

Using Reddit's structure, they would store it as denormalized:

id	key	value
1	username	steve
1	name	Steve Corona
1	location	Charleston, SC
2	username	rob
2	name	Robert Johnson
2	location	New York, NY

The advantage is simple: You can create a new column by simply using it. You don't have to modify your MySQL table at all.

There are a bunch of things *wrong* with this design- The data is denormalized, so it's going to use way more disk space because you have to store the key text for every "row", and you can't enforce database constraints like foreign keys, default values, or types. I'm sure if you spent more time thinking about it you could probably find 10 other reasons why this is *wrong*.

But none of that matters. The real reason why storing your data this way is a bad design is because you're **using the wrong tool for the job**. If you want to use a key-value store or a schema-less database, *USE A KEY-VALUE STORE OR SCHEMA-LESS DATABASE*. We don't use hammers to put screws into the wall, and likewise, we shouldn't use MySQL for something it's not designed to be. Use the right tool.

The least frustrating way to make schema changes

If you're stuck with MySQL, what's the easiest way to make schema changes without locking up your entire database cluster and subsequently taking down your entire website? I'm going to outline the best approach below.

⁶⁶<http://highscalability.com/blog/2010/5/17/7-lessons-learned-while-building-reddit-to-270-million-page.html>

Pretend you have a table, `foobar`, and you want to add a new column to it, `username`. Our database cluster has five MySQL slaves and a single master.

The SQL for making the changes to `foobar` is `ALTER TABLE foobar ADD COLUMN username VARCHAR(25)`.

We want to add the `username` column to `foobar` completely seemlessly, without causing any downtime for our users. If we were to run the `ALTER TABLE` statement on the master server, it would get replicated to the slaves immediately, completely locking the `foobar` table. To make matters worse, MySQL Replication is single-threaded, so the slaves would start to lag behind while they were waiting for the `ALTER TABLE` to finish. Obviously, this is not acceptable given our constraints.

The best way to handle the schema change is to take each slave, one-by-one, and cut off all traffic to it by removing it from our HAProxy pool. Once a slave is removed from service, we turn off MySQL replication, run the `ALTER TABLE` on it individually, wait for it to finish, restart replication, and put it back into the pool. It's a time-consuming process, but prevents any website downtime.

But there's still the little matter of making the change to the master server. The process here is similar, though. We manually promote one of the database slaves to be the master, demote the master to a slave, and run the `ALTER TABLE` command. When it's done, the old master is promoted again and your entire database cluster now has the new schema without any downtime.

This process is extremely painstaking and time consuming, but it's really the only way to make the schema changes without locking the table in production. It also gives you good practice for getting comfortable with master/slave promotion and demotion outside of a crisis.

If you're locked into MySQL but absolutely need instant schema changes, there's a commercial storage engine for MySQL called [TokuDB](#)⁶⁷ that can add new columns and indexes without locking your table and degrading performance. TokuDB can also compress your data up to 25x and reduce I/O usage- in my tests a billion row 160GB InnoDB table only weighed in at 30GB in TokuDB, with better performance. Downsides? It costs \$2500/server/year (but is free to try) and doesn't work with Percona.

Further Topics

Writing this chapter was like going down a rabbit hole. As I was writing and researching topics, I'd constantly run into other sections that I wanted to talk about. In order to prevent myself from going on a thousand tangents and actually *finish* the damn thing, I had to limit the amount of topics that I covered.

Here's a list of the other topics that I wanted to cover. I might add a section for some in future edits or revisions of the book, so if you're particularly interested in one, send me an email to cast your vote.

mysqlproxy

⁶⁷<http://tokutek.com>

Proxy server for MySQL that lets you write scripts in Lua to inject behavior. It can be used for MySQL R/W splitting, slave load balancing, etc. [More information here](#)⁶⁸.

Optimizing Queries with `explain`

MySQL Connection Pooling w/ `mysqlnd_mux`

Asynchronous MySQL Queries

How to quickly bootstrap a MySQL Slave

Tumblr JetPants

Benchmarking MySQL with `sysbench`

⁶⁸<http://dev.mysql.com/doc/refman/5.1/en/mysql-proxy.html>

Cache Server: Using Redis

A common way to improve the performance of your PHP application is to use an in-memory object cache to speed-up expensive to calculate by calculating it once and persisting the result between HTTP requests. The data can come from any slow source- 3rd Party APIs and Filesystems just to name some, but more often than not, the data is coming from your database.

In an ideal world, the database's query cache would handle this sort of thing for us. However, if you recall from Chapter 6, we talked about how MySQL's query cache is all but useless. The reason is that it completely erases the query cache after every `INSERT` or `UPDATE`, making it all but useless for tables with even a moderate amount of database writes.

Another advantage of using a separate object cache layer is that with simple key hashing, you can distribute the cached objects across multiple servers, allowing you to use memory from every machine in your Memcached or Redis cluster. MySQL's cache, on the other hand, can only take advantage of the resources of a single machine. That's not to say that tuning MySQL isn't important, it's just that having an object caching system complements MySQL with improved performance.

Choosing between Redis, Memcached, and APC

We have a couple of options available to us when looking for an in-memory object cache. Each one is useful and excels for different needs, so let's talk about each of them.

APC

In the Application Server chapter, we discussed using the [PECL APC Extension](#)⁶⁹ as a mechanism for speeding up your application by caching the compiled bytecode of your source. On top of that, [APC also has an interface for storing and retrieving](#)⁷⁰ data between PHP requests using shared memory, which can be used as an object cache.

Here's an example of how we can use APC. Imagine that in one script, you have this code, which you run.

```
1 <?php
2 $array = array("foo1", "foo2", "foo3");
3 apc_add("foobar", $array);
```

And in separate script, that runs independently from the code above:

⁶⁹<http://pecl.php.net/package/APC>

⁷⁰<http://php.net/manual/en/book.apc.php>

```

1 <?php
2 // $array will contain array("foo1", "foo2", "foo3") after
3 // fetching it from APC.
4 $array = apc_fetch("foobar");

```

Using `apc_add` and `apc_fetch`, you're able to store and retrieve data lasts longer than the lifecycle of a normal PHP request. Of course, there are no guarantees and data can come and go, so it should only be used for re-calculatable cache data.

APC offers atomic counters and expiration times, too, which gives it a similar feature set as Memcached.

The biggest pro of using APC is that it's extremely fast. In fact, it's faster (at a micro level) than Redis or Memcached. Why? It uses shared memory within the PHP process, so it doesn't need to go out on the network or communicate with a separate server. It's biggest asset, however, is also it's biggest weakness. Because the shared memory is tied to a parent PHP-FPM process, each server has to maintain its own separate cache data, effectively making it non-distributed since all cache data needs to be re-calculated and re-stored for each server.

Furthermore, because APC relies on shared memory, it only works with PHP-FPM or `mod_php`. A PHP script running on the command line or in a cronjob won't be able to see the shared memory because it doesn't share the same parent process as your PHP-FPM workers.

Does this make APC useless? Not at all. It's highly useful for caching small pieces of data that don't take too long to compute. For example, I've used it with some success for caching configuration files. Without APC, the configuration files (stored in [YAML format⁷¹](#)) would have to be re-read and parsed on every request. Instead, since the files don't change that often, I was able to parse the YAML files once and store the result in APC. The code looked something like this:

```

1 <?php
2 function load_config() {
3     $config = apc_fetch("config");
4
5     if ($config === false) {
6         $config = yaml_parse_file("app/config/config.yml");
7         // Store the parsed file in APC for the next access.
8         // The 3rd parameter is an automatic expiration time.
9         apc_add("config", $config, 86400);
10    }
11
12    return $config;
13 }

```

It's worth noting that the APC cache is *never persisted to disk*. That means if PHP-FPM crashes or gets restarted, the cache will be wiped clean.

⁷¹<http://www.yaml.org/>

Memcached

Memcached is the ol' reliable object cache software. It seems like all of the big guys use it- Twitter, Facebook, and 37Signals to just name a few. Memcached really pioneered the idea of a separate, distributed cache layer for your application and was really the only option a few years ago.

Memcached is similar to APC in a lot of ways, except that it runs as a threaded C server, so you need to go over the network to communicate with it. Compared to APC, communicating over the network is obviously slower but still ends up being extremely performant to the tune of 500,000+ operations per second. The major advantage is that with some simple hashing, multiple Memcached servers can effectively be grouped together to distribute your cached data and provide you with the resources of all of the servers combined. That is to say, if you had 10 Memcached servers, each with 4GB of memory, you'd effectively have a 40GB in-memory cache available to you.

Memcached provides all of the same features as APC, including CAS (check-and-set) tokens which allow you to avoid race-conditions by only allowing new values to be written if they key hasn't been modified.

That being said, I'm not going to cover Memcached in this chapter because Redis, which was released a couple of years ago, has far outpaced Memcached in terms of features and reliability while offering similar performance (aka, damn fast).

Stuck on Memcached?

If you like Memcached, or you're already using it and stuck with it, I've included a short bonus chapter on Memcached. But you should seriously consider switching to Redis. It's a drop in replacement.

Redis

Redis is where it's at. If you've never used it before, you're in for a treat- it's got enough coolness to make your inner geek excited. Just like Memcached and APC, it can do key/value caching with GET and SET primitives, but it also adds other data types like lists, hashes, and sorted sets. There are literally hundreds of commands, and I won't be able to cover them all here, so [check out the Redis documentation⁷²](#) to play with the commands. There's even an in-line console that let's you play with each command right on the website.

Remember, the entire dataset stored in Redis MUST fit into memory. While Redis does persist your data to disk, it only reads from disk during startup so you must always have more memory than data.

Why is Redis better?

- Memcached/APC data is never saved to disk, meaning that if a server crashes or gets restarted, you start with a blank slate. Redis, on the other hand, can be configured to sync

⁷²<http://redis.io/commands>

to disk, allowing you to always have a warm cache on-hand. When your site becomes **dependent** on having a fast cache, there are serious performance implications to blowing away your entire cache because of a restart. (While you can configure Redis to sync after *every* write, when you're using it as a cache it's better to allow for some data-loss in order to keep it fast.)

- Memcached/APC only offer key/value primitives, GET and SET, and atomic counters. While you can hack extra data types, they are often unsafe and full of race conditions. Redis, on the other hand, includes lists, sets, sorted sets, hashes, master/slave replication, transactions, and even lua scripting. Since I won't go over every single Redis command, you can check out the [full documentation here](#).⁷³
- Lastly, Depending on which version of libmemcached you're using (mainly older versions that were bundled with Ubuntu 10.04), there are some serious bugs with the way that timeouts are handled. As in, they are completely broken. This bit us *hard*, a single instance of Memcached going down would block all of our PHP-FPM processes and take down the site. The C [hiredis](#)⁷⁴ library that most Redis client use, on the other hand, has been reliability solid because it's written by [antirez](#)⁷⁵, the creator of Redis.

Redis Commands

Redis offers way more than just the typical set and get operations that we've seen with Memcached and APC. I'll briefly cover them here with some examples and common use-cases. The [full command list is available here](#)⁷⁶.

Simple Operations

The most basic features give us exactly what APC and Memcached have- simple GET and SET operations.

```

1 <?php
2
3 $r = new Redis;
4 $r->connect("127.0.0.1");
5
6 $r->set('key', 'foobar');
7 $r->get('key'); // returns foobar
8
9 $r->setex('key', 100, 'foobar'); // set with 100 second expiration
10 $r->psetex('key', 100, 'foobar'); // set with a 100ms expiration
11
12 $r->setnx('key', 'foobar'); // set only if key doesn't exist
13 $r->delete('key');
```

We have atomic counters, too!

⁷³<http://redis.io/commands>

⁷⁴<http://github.com/redis/hiredis>

⁷⁵<http://twitter.com/antirez>

⁷⁶<http://redis.io/commands>

```

1 $r->incr('views');           // 1
2 $r->incr('views');           // 2
3 $r->incrby('views', 10); // 12
4 $r->decr('views');           // 11

```

Other single key commands worth exploring- `ttl`, `expire`, `exists`, `expireat`, `rename`, `type`, and `persist`.

GET and SET commands are great for storing singular data. The increment and decrement commands are atomic (meaning, you don't have to worry about race conditions.)

Hashes

Redis can store hashes, which are pretty much like a PHP associative array, and can be used to group information together by key.

```

1 <?php
2
3 $r->hset('user:1', 'name', 'Steve');
4 $r->hset('user:1', 'username', 'scorona');
5
6 $r->hset('user:2', 'name', 'John');
7 $r->hset('user:2', 'username', 'jdoe123');
8
9 $r->hget('user:2', 'name'); // returns John
10 $r->hlen('user:2'); // returns number of keys (2)

```

Other hash commands worth exploring- `hgetall`, `hexists`, `hvals`, `hkeys`, `hincrby`, `hmset`, `hmget`, `hdel`, and `hsetnx`.

Hashes can be used just like SET and GET, but are great (and more efficient+convenient) for storing data that can be grouped together by a common key. For example, if we were storing user data in Redis, it makes sense to use a hash because the data can be grouped together by the user id.

Lists

A Redis List is like a numerically indexed array in PHP.

```

1 <?php
2
3 $r->lpush('key', 'foo'); // returns 1
4 $r->lpush('key', 'bar'); // returns 2
5 $r->rpush('key', 'test'); // returns 3
6 $r->lsize('key'); // returns 3
7 $r->lindex('key', 0); // returns bar
8 $r->lset('key', 1, 'testiest');

```

Other list commands- `lpushx`, `lpop`, `lsize`, `ltrim`, `lrange`, `lrem`, and more.

Lists are great for storing, well, lists of data. You can add, remove, trim, push, and pop items from the list all atomically. I love to use lists for time ordered data, because they are naturally sorted by insertion time when you use `lpush`.

A great use case is for notification feeds, where you have a list for every user and push new notifications to the front of the list.

Lists are also the basic primitive that our queuing system, Resque, uses as it's underlying data structure.

Sets and Sorted Sets

Sets are similar to lists in that you can store a bunch of data in single key. The main differences are that sets are completely unordered and can not contain any duplicate data. Unlike lists, sets give you more robust features- checking to see if a value is inside of a set, unioning sets together and even calculating intersections and differences. There is a lot of power here.

```

1 <?php
2
3 $r->sadd("names", "steve", "john", "alice");
4 $r->scard("names"); // returns 3
5 $r->sismember("names", "steve"); // returns true

```

On top of sets, we also have sorted sets which are slower but allow you to order your set as they are normally completely unordered.

One use case for sets is storing lists of data that you want to guarantee is unique- usernames, category names, or titles. You can quickly use `sismember` to determine if a value is part of your set.

Transactions

You can also group commands together into a transaction. Transactions are faster for issuing a lot of commands because it only involves one roundtrip to Redis.

```
1 <?php
2
3 $r->multi()
4   ->set("key", "value")
5   ->set("foo", "bar");
6   ->hset("user:2", "name", "Steve")
7   ->exec();
```

Transactions also run as an atomic unit, but you can disable that by passing `Redis::PIPELINE` to `multi()`, which transmits the commands to the server in one block, but doesn't guarantee that the commands will execute atomically.

Pub/Sub

Redis includes a really cool publish and subscribe feature that's great for doing cross-process, asynchronous communication.

Imagine we had two scripts, in the subscriber:

```
1 <?php
2
3 $r->subscribe(array('updates'), function($r, $channel, $data) {
4   echo "Received {$data}\n";
5 });
```

And in the publisher:

```
1 <?php
2
3 $r->publish('updates', "Some new data");
```

The subscriber would receive our message from the publisher, like magic. Multiple subscribers can listen into the same channel and you can have multiple channels. It's pretty powerful and can be leveraged for backend script communication, implementing a chat room, and for streaming services over websockets.

The importance of Atomic Operations

All commands in Redis are atomic. What that means is that each command is isolated and doesn't impact the state of any other commands. Let's look at an example of where this matters (and why atomic counters are important).

The naive way of counting might look like this:

```

1 $v = $r->get("key"); // $v is 1
2 $r->set("key", $v+1); // Now setting "key" as 2

```

That would be wrong, though, because if another process changed the value of `key` to 2 between our `get` and `set` commands running, we'd set it at 2 again and lose the increment.

In Redis (and Memcached and APC), we can solve this by using atomic counters. Effectively, we tell the counter how much we want to increase the counter by, and it does the math on the server.

```

1 $v = $r->incr("key"); // Returns the new value of "key"

```

With Redis, the `watch()` command lets you do check-and-set operations, so while using `incr()` is preferred, you could fix the first example like this:

```

1 $ret = false;
2 while ($ret == false) {
3     $r->watch("key");
4     $v = $r->get("key");
5
6     // Will return false if "key" was modified by another client
7     // between watch() and exec().
8     $ret = $r->multi()->set("key", $v+1)->exec();
9 }

```

Performance Limitations of Redis

Unlike Memcached (which is multi-threaded), Redis is single-threaded and is evented. What that means is that each Redis server can only take advantage of a single CPU core. This isn't usually a problem because most Redis commands are $O(1)$ or $O(n)$, so they don't burn up much CPU. In the case of needing to scale your Redis CPU usage, you can run multiple Redis daemons, using different ports, on the same server.

Redis uses more memory to store the same data on a 64-bit system than on 32-bit system because pointers are 8 bytes on 64-bit systems and 4 bytes on 32 bit systems. Each item that you store in Redis uses atleast one pointer (3 for lists, 8+ for a sorted set), so if you have lots of small keys the memory difference can become significant.

One way around it is to run your system in 32-bit mode with PAE (Physical Address Extension), which allows your kernel to address 36-bits of address space (64GB of memory). Even with PAE, each process can only see 4GB of memory, so you'd have to run 16 separate Redis daemons to use all of the free memory.

In practice, does it matter? Not really. Suck it up, use a 64-bit system, and just add more memory. PAE is a kludgy kernel hack anyways.

Installing Redis from Dotdeb

Ubuntu bundles an old version of Redis and the Redis website only provides the source code, so we need to install it from [Dotdeb](http://dotdeb.com)⁷⁷. Of course, you could compile Redis from scratch, but I prefer to use packages when possible.

Setting up Dotdeb is as easy as adding another apt source (you can skip this if you already used DotDeb for installing PHP 5.4).

```

1 > vi /etc/apt/sources.list.d/dotdeb.list
2
3 deb http://packages.dotdeb.org squeeze all
4 deb-src http://packages.dotdeb.org squeeze all
5
6 > wget http://www.dotdeb.org/dotdeb.gpg
7 > cat dotdeb.gpg | sudo apt-key add -
8 > apt-get update

```

Installing Redis is as easy as running apt-get.

```

1 > apt-get install redis-server
2 > service redis-server start

```

At the time of writing, Redis 2.6 was *just* released. It's preferable to always run the latest release because new features and optimizations are always being added between versions.

Installing the `phpredis` C extension

There are some pure PHP redis libraries, like [Predis](https://github.com/nrk/predis)⁷⁸ and [Redisent](https://github.com/jdp/redisent)⁷⁹ that do a good job of implementing a Redis library without any C code, but it comes at a cost- they're 3-4x slower.

The best C extension for connecting PHP to Redis is [phpredis](https://github.com/nicolasff/phpredis)⁸⁰- it's frequently updated and works very consistently. Unfortunately, it doesn't use [hiredis](https://github.com/redis/hiredis)⁸¹ yet, but it still works very well.

Note: PHPRedis is not included in the PECL Repository. You may skip the installation steps below by just using `pecl install redis`.

Installing it takes a tiny bit more work than usual, because it's not in the official PHP PECL repository.

⁷⁷<http://dotdeb.com>

⁷⁸<https://github.com/nrk/predis>

⁷⁹<https://github.com/jdp/redisent>

⁸⁰<https://github.com/nicolasff/phpredis>

⁸¹<https://github.com/redis/hiredis>

```
1 # You might not need these if you already have them
2 > apt-get install git build-essential
3
4 # First install igbinary
5 > pecl install igbinary
6
7 > git clone git://github.com/nicolasff/phpredis.git
8 > cd phpredis
9 > phpize
10 > ./configure --enable-redis-igbinary
11 > make && make install
12 > vi /etc/php5/conf.d/phpredis.ini
13 extension=redis.so
```

After that, you'll need to restart PHP-FPM.



What's igbinary?

Igbinary is a replacement for the out-of-the-box PHP serializer. Normally, PHP's serializer is string based and serializes objects into an ASCII-based format. Not only is this slow, but it's also not space efficient, which makes a big difference when you're storing PHP objects (such as arrays, database results, and models in your cache). Igbinary, instead, replaces the serializer with one that outputs binary instead.

Tuning Redis for Performance

Luckily, Redis has a very well documented configuration file in `/etc/redis.conf`, so I don't need to go over each and every option here. You can just read the configuration file (the entire version is available [here](#)⁸²). I will cover the most important settings, but luckily there aren't many.

Redis Persistence

Redis has a very unique persistence setup and stores data to disk in a very different way than a traditional database like MySQL. In fact, Redis actually has two methods of persistence that can be used by themselves or together (recommended). Because Redis stores the *entire* dataset in memory at all times, and the unique way it stores data to disk, it's less dependent on disk speed than traditional databases.

RDB

The RDB persistence method is used to snapshot the entire dataset stored in Redis and put it on disk. When a RDB snapshot is taken, it snapshots the data into a new file and deletes the old RDB file. That is to say, RDB files are never updated, so saving a new snapshot only uses very fast

⁸²<https://raw.github.com/antirez/redis/2.6/redis.conf>

sequential I/O. However, because taking a new snapshot requires copying the entire dataset to disk, it needs to happen in the background since the dataset can often be several gigabytes in size. Because of the limiting factor, when using purely RDB snapshots as your persistence method, it's possible to lose data if Redis restarts before the most up-to-date snapshot can be written to disk.

Redis can create RDB snapshots manually with the `save` or `bgsave` command and will also create one when properly shutdown with the `shutdown` command. Likewise, we use the `save` setting in `redis.conf` to turn on automatic snapshotting. The default setting looks like this:

```
1 save 900 1
2 save 300 10
3 save 60 10000
```

The `save` setting takes two parameters, # of seconds and # of changes. With the settings above, it will cause Redis to create a snapshot after 900 seconds if more than 1 key has changed, after 300 seconds if more than 10 keys have changed, and after 60 seconds if more than 10,000 keys have changed.

RDB is convenient because it's fast and space-efficient- great for taking backups. To take a backup, you just make a copy of the `dump.rdb` file. Super easy. The disadvantage, though, is that you can lose data- up to 15 minutes with the default settings.

AOF

AOF persistence, or append-only file, is similar to the InnoDB log file in MySQL. Any write command (`set`, `hset`, `lpush`, etc) sent to Redis is written into a log file which can be replayed in case of crash. Redis allows you configure how it handles calling `fsync` on the AOF via the `appendfsync` setting. It can **always** `fsync`, which is safest but performs the worst because `fsync` is an expensive system call. That being said, when `appendfsync` is set as **always**, you will not lose *any* data in the event of a crash. If you're okay with losing up to second of data, you can set `appendfsync` to `everysec` and Redis will `fsync` the AOF around once every second. You can also set it to `no`, which relies on Linux syncing the AOF to disk whenever it decides to. `everysec` is usually the best option if some dataloss during a crash is okay.

Before Redis 2.4, the AOF would just grow and grow unless you manually told Redis to rewrite it. Now it will handle this for you automatically. Lastly, if you use AOF with RDB, during an AOF rewrite, Redis will truncate the starting point of the AOF to the last RDB snapshot and keep the AOF small.

Tuning Linux

We can tune Linux for Redis similarly to how we tuned it for MySQL. I won't repeat them here since they're exactly the same.

1. Turn off Filesystem Access Times
2. Use XFS

3. Increase Open Files
4. Tune Swapiness
5. Change I/O Scheduler to anything but CFQ

Notable `redis.conf` Settings

`maxclients`

The maximum amount of concurrently connected clients. Set to 10,000 by default. Usually enough. Increase it if you have a very, very busy Redis server.

`maxmemory`

The maximum amount of memory for Redis to use. It's always a good idea to give your Redis server a hard cap, I recommend 95-97% of your servers memory.

`maxmemory-policy`

The behavior that Redis should use when it hits `maxmemory`. There are a couple of different settings, but `volatile-lru` is almost always what you want. It will remove the less-frequently used keys that have an expiration time, but never remove keys without an expiration time. This is why you should always attach an expiration time to data that can be regenerated, even if the expiration is years in the future.

<code>maxmemory-policy</code>	Behavior
<code>volatile-lru</code>	Removes only keys w/ an expiration, least recently used
<code>allkeys-lru</code>	Removes any key, least recently used
<code>volatile-random</code>	Removes only keys w/ an expiration, randomly
<code>allkeys-random</code>	Removes any key, randomly
<code>volatile-ttl</code>	Removes only keys w/ an expiration, closest to expiring
<code>noeviction</code>	Don't remove any data, just return an error on writes

`appendonly`

By default set as no, it's almost always a win to use both RDB and AOF, so set it to yes.

Choosing the right hardware

Redis will benefit from a few smart hardware choices. Memory is the biggest factor because **all of the data must be able to fit into memory**. As far as CPU, you want to choose a modern, high frequency CPU. Cores make little to no difference (because Redis runs as a single process), so choose more GHz over more cores.

If you're using the persistence feature in Redis (you should), you'll benefit from having your data file on a RAID 1 or RAID 10, but SSDs aren't necessary. Redis syncs the datafile to disk in the background, so slower disks won't really impact performance.

It's helpful to use a second disk for the `appendonly` log file, since it will use a steady stream of sequential write IOPS. Again, not necessary to have an SSD here if you're using `appendfsync everysec`. If you use `appendfsync always`, a SSD or Hardware RAID w/ Battery Backup for the log file will be beneficial due to faster `fsyncing`, which subsequently speeds up writes to Redis.

Our standard setup is a Xeon 1270 (3.4GHz) with 64GB of Memory and a RAID 1.

Scaling Redis to Multiple Servers

There are two common ways of scaling Redis past a single machine, **Master/Slave Replication** and **Sharding**. Both methods allow you to scale to multiple machines, but they scale differently depending on your use of Redis. Both techniques, however, can be used together to provide the best case scenario of fast reads, writes, and redundancy. **Redis Cluster** is going to be released sometime in the future and will make scaling Redis even easier, but don't hold your breath- it's been pushed back 2 or 3 times already.

Replication

The easiest way to scale is by using the built-in Master/Slave replication. It's similar to MySQL- one Redis server is designated as a master and all writes must be sent to it, which get replicated to multiple read-only slaves. Replication solves two problems- it scales reads (but not writes), useful if you have a read heavy workload or heavily use the expensive functions in Redis (like `SORT`, `SINTER`, etc). On top of that, Replication also adds a layer of redundancy and failover, since your dataset is available on multiple machines.

The nice thing about Redis replication is that it's extremely easy to setup and use.

On your slave, you just need to add `slaveof 192.51.100.10 6379` to your `redis.conf` file (of course, replace `192.51.100.10` with the Master's IP address) and the slave will bootstrap itself when it's restarted. Yes, this includes syncing itself with the master, so you don't need to worry about migrating or transferring data manually. Way easier than MySQL!

On your slave, make sure you have `slave-read-only` set to `yes`, so you don't accidentally write any data to the slave directly.

Sharding

Sharding is a common technique for splitting up data across multiple servers by using a consistent hash algorithm to determine where a specific piece of data is stored on a cluster of servers. By sharding your data, you're able to split your data across multiple servers and scale horizontally. Since each shard in your cluster is only handling a subset of your entire dataset, the amount of server resources available to your cluster grows as you add more machines. For example, if you sharded your data across two Redis servers, each with 16GB of memory, you will effectively be able to store up to 32GB of data in Redis.

Sharding is done on a per-key basis. Each key that you use to store a piece of data is hashed and the value of that hash is used to determine which server to send the data to. Likewise, when you

want to retrieve a piece of data using its key, the key is hashed and the value is used to determine which server to get the data from. Because of this, a single key, even if it's a set or a list, will always be stored on a single server. Even if the list has a million values in it. One key, one server. Keys are split up, not data.

How consistent hashing works

A quick sidetrack, in case you aren't familiar with consistent hashing. Pretend you had 5 servers and you wanted to shard your data across them. How can we store data to these 5 servers and later retrieve the data from them without knowing where the data is stored? Of course, we could pick one randomly and later ask all of them for the data, but that's inefficient and doesn't scale to thousands of servers.

What we can do, however, is create a hashing algorithm that uses the key to figure out where to store the data. A simple algorithm might use the first letter of the key to determine which server should store the data. If the key starts with a-e, send the data to `server01`, f-j to `server02`, k-o to `server03`, etc. When you want to retrieve the data, you'd run the same algorithm on the key and it'd tell you which server to pull the data from. Obviously, real-life hashing algorithms are more complicated, but the concept is the same.

Sharding with `phpredis`

With Redis, Sharding is handled purely on the client-side by the library being used. `phpredis` supports sharding using the `RedisArray`⁸³ class. It's pretty easy to use, you just pass in array of hosts to the constructor:

```
1 <?php
2 $r = new RedisArray(array("192.51.100.11",
3                           "192.51.100.12",
4                           "192.51.100.13"));
5
6 $r->set("foobar", "some_value");
7 $r->get("foobar");
8
9 $r->lpush("list", "some_data");
```

You can use all of the normal Redis commands on a `RedisArray`, the key will be hashed and the command will be automatically be sent to the correct server.

Transactions are run on a single host You can no longer use transactions that work on multiple keys if the keys are going to be located on multiple servers. When you use `multi()`, you have to pass in the Redis host to use by using the `_target()` function.

⁸³<https://github.com/nicolasff/phpredis/blob/master/arrays.markdown>

```
1 <?php
2 $r = new RedisArray(array("192.51.100.11",
3                         "192.51.100.12",
4                         "192.51.100.13"));
5 $host = $r->_target("foobar")
6
7 $r->multi($host)
8   ->get("foobar")
9   ->set("foobar", "123")
10  ->set("another_key", "123")
11  ->exec();
```

See how we lookup the host of the foobar key by using `_target()`? Everything in that transaction will get run on whatever value is returned for `$host`, even `set("another_key", "123")`, which may not normally hash to the value of `$host`.

“The Dogpile”



Creative Commons Image, from <http://www.flickr.com/photos/85622685@N08/7934519880/>

When you use Redis to cache data that's database intensive to calculate, it's important to avoid a common “gotcha” called the dogpile. Let's setup a scenario.

Imagine you have a busy site and on the homepage you show the latest photos from your most popular users. Unfortunately, as the amount of users and photos grows, the query to show the

most popular photos becomes slower and slower, because the amount of data the query has to process grows into the millions and tens-of-millions. Now, the query takes over 10 seconds to run and it not only significantly slows down the speed of your homepage (which gets blocked waiting for the query), but also hammers your database servers.

At first, you solve this problem naively! Let's just cache it! Your solution might look something like this:

```

1 <?php
2
3 $r = new Redis;
4 $r->connect("127.0.0.1");
5
6 $popular_photos = $r->get("popular_photos");
7
8 if ($popular_photos === false) {
9     // Calculate the popular photos from MySQL
10    $popular_photos = expensive_database_call();
11
12    // Store the data with a 10 minute expiration
13    $r->setex("popular_photos", 600, $popular_photos);
14 }
```

And this would work great- if you only had exactly one person accessing your site at a given time. We set an expiration, so after the first run, the popular photos get stored in Redis for 10 minutes. Once they expire, the `if` statement gets run and we regenerate the popular photos. But remember- this query takes 10 seconds to run. While `expensive_database_call()` is running, every incoming web request is going to jump through and also call `expensive_database_call()`, since the data isn't in Redis yet. Hence the dogpile.

The easiest way to solve the problem is just to remove the `if` statement entirely and call `expensive_database_call()` from a cronjob. It removes the dogpile problem completely.

It's not always possible to solve this problem with a cronjob, though, so an alternative solution is to create a Redis lock (using `setnx`), which only permits one client to run `expensive_database_call()`.

```

1 <?php
2
3 $r = new Redis;
4 $r->connect("127.0.0.1");
5
6 $expiration      = 600;
7 $recalculate_at = 100;
8 $lock_length     = 20;
9
10 $value = $r->get("popular_photos");
```

```
11 $ttl    = $r->ttl("popular_photos");
12
13 if ($recalculate_at >= $ttl && $r->setnx("lock:popular_photos", true)) {
14     $r->expire("lock:key", $lock_length);
15
16     $value = expensive_database_call();
17
18     $r->setex("popular_photos", $expiration, $value);
19 }
```

Here's how it works:

1. We pull from the popular_photos key as normal, but also retrieve the TTL, which returns the number of seconds left before the key expires.
2. The expiration is set at 600 seconds, but when popular_photos is 500 seconds old and only has 100 seconds before it expires, we kick off the process of refreshing the data.
3. \$r->setnx("lock:popular_photos", true) tries to create a key to use as a lock. **setnx returns false if the key already exists, so only one client will get a true value and enter the if statement.**
4. We use the expire command to set a 20 second expiration on the lock- that way, if PHP crashes or times out while the lock is being held, another process can eventually regenerate the data.
5. We call expensive_database_call() and put the refreshed data back into the popular_photos key as normal.

As you can tell, it requires a little bit more leg work than the naive solution, but it alleviates the dogpile entirely- very necessary when you're going to caching long-running queries.

Russian Doll Caching



Creative Commons Image: <http://www.flickr.com/photos/7604548@N07/2577737283/>

When I first started using a cache in my web stack, Memcached was still very new and the only option available. Redis wasn't even in existence at that point, so I didn't have the luxury of using it. The way I tackled the problem was to cache as many database queries as possible, freeing up MySQL to work on harder queries.

I tackled the problem in a pretty naive way- let's pretend I had a User model through my ORM. The ORM handled all of the querying and just returned a nice object. The code looked something like this:

```
1 <?php
2
3 $id    = $_GET['id'];
4 $user = $r->get("user:{$id}");
5
6 if ($user === false) {
7     $user = User::find($id); // This issues the SQL
8     $r->setex("user:{$user->id}", 86400, $user);
9 }
```

Caching this way is called read-thru caching. It relies on expirations to clear out stale data and writing straight to the database will cause there to be stale (possibly, invalid) data inside of our cache.

We can solve the stale data problem by either deleting the `user:{$user->id}` key when updating the record or updating it.

Deleting the record

```

1 <?php
2
3 $username = $_GET['username'];
4 $user      = User::find_by_username($username);
5
6 $user->city = "New York";
7 $user->save();
8
9 $r->delete("user:{$user->id}");

```

Updating the record

```

1 <?php
2
3 $username = $_GET['username'];
4 $user      = User::find_by_username($username);
5
6 $user->city = "New York";
7 $user->save();
8
9 $r->setex("user:{$user->id}", 86400, $user);

```

This isn't a great solution for all but the simplest cases, though. Reason being- any other cached queries that refer specifically to our User record are going to still be stuck with stale data. Imagine you had a page that showed a list of all the users. You'd use a query like, `SELECT * FROM users`. If you cached that query and changed one of the users individually, your cached `SELECT * FROM users` would give you stale data.

Obviously, you can overcome this by purging all of the related cache keys when saving your User model, but that's gross. You'd need to remember and keep track of all places where the User model can get cached.

Key-based Expiration

Russian Doll Caching, a term made popular by 37signals in this blog post⁸⁴, is a fancy name for **Key-based expiration**. 37signals caches views (aka rendered HTML) instead of database queries, but the technique is the same. Caching views is less prevalent in PHP because it's slightly more complex and rendering views is much less expensive in PHP than Rails.

⁸⁴<http://37signals.com/svn/posts/3112-how-basecamp-next-got-to-be-so-damn-fast-without-using-much-client-side-ui>

Anyways, Key-based expiration is technique for expiring a group of related cache keys by changing a timestamp in the key anytime data in that group changes, forcing it to be re-fetched from the database. You might be asking yourself- but what about all of that extra data? Since we're not manually purging old data, won't it eventually all build up? We rely on Redis' `maxmemory-policy` setting, which will remove the most unused keys when it needs to free up memory.



Configuring the `maxmemory-policy` setting

In the `redis.conf` file (discussed later in the chapter), you'll need to configure the `maxmemory-policy` setting to tell Redis how it should handle an out-of-memory situation. There a few different strategies, buy volatile-lru is the best for key-based expiration. When set as volatile-lru, *only keys that have expirations set* will be eligible for removal, with the least recently used ones getting booted first.

It works by building the timestamp into the key-

```

1 <?php
2
3 $id    = $_GET['id'];
4 $slug = $r->get("slug:user:{id}");
5 $user = $r->get("${slug}:user:{id}");
6
7 if ($user === false) {
8     $user = User::find($id); // This issues the SQL
9     $r->setex("${slug}:user:{id}", 86400, $user);
10 }
```

In the above example, you'll notice that we make an extra cache lookup to find the value of `slug:user:{id}`. That returns the timestamp of the last time the record was modified. Since we use the return value of that lookup to build our main cache key, `${slug}:user:{id}`, every time it changes, it will stop us from using stale data, and force a pull from the database.

This is what our modified code that updates the database would now look like. Notice that it updates `slug:user:{id}` with the latest timestamp, but does not delete any data from the cache.

```

1 <?php
2
3 $username = $_GET['username'];
4 $user = User::find_by_username($username);
5
6 $user->city = "New York";
7 $user->save();
8
9 $r->set("slug:user:{user->id}", time());
```

The power in key-based expiration is that you no longer need to manually manage your keys. You just make sure to update the slugs every time you change data that the group relies on, and it'll force an expiration. This technique works equally well on both Memcached and Redis.

Taking advantage of Redis

You can take advantage of Redis' feature set by using it's hash datatype to use Russian Doll caching without having to deal with the complexities of Key-based expiration.

It's easier to show you how it works, so let me show you an example. Pretend that in our code, we have three related queries.

1. `SELECT * FROM images WHERE id = {$image_id} AND user_id = {$user_id}`
2. `SELECT * FROM images WHERE location_id = {$location_id} AND user_id = {$user_id}`
3. `SELECT * FROM images WHERE user_id = {$user_id}`

These queries are all based around the `$user_id`. Here's how one of the queries' code would look using hash-based expiration instead of key-based.

```

1 <?php
2
3 // For Images @ at a Location
4
5 $loc_id    = $_GET['location_id'];
6 $user_id   = $_GET['user_id']
7
8 $image = $r->hget("user:{$user_id}", "location_id:{$location_id}");
9 $query = "SELECT * FROM images WHERE location_id = {$loc_id} AND user_id = {$user_id}";
10
11 if ($image === false) {
12     $image = Image::where($query);
13     $r->hset("user:{$user_id}", "location_id:{$location_id}");
14     $r->expire("user:{$user_id}", 86400);
15 }
16 }
```

Two things worth noting here- The first is that we're caching the query inside of a hash, `user:{$user_id}`, instead of regular key. The second is that we have to manually call `expire` because `hset` doesn't support adding an expiration to the hash in one command.

Now, when updating the `Image` model, instead of updating a slug, we simply delete the `user:{$user_id}` hash, which will wipe out the cache for that user.

```

1 <?php
2 $id      = $_GET['id'];
3 $image = Image::find($id);
4
5 $image->title = $_GET['title'];
6 $image->save();
7
8 $r->del("user:{$image->user_id}");

```

Clearing out stale data this way gives you all of the advantages of the timestamp/slug method discussed earlier with none of the hassle of having to manage a separate key.

Redis Bitmaps

Redis has a nifty feature that's been talked about quiet a bit frequently, but on the surface doesn't seem very useful or interesting. Redis Bitmaps! So basically, in Redis you can use the `setbit` and `getbit` commands to individually toggle bits inside of a string stored in Redis.

Woohoo, how is this useful? Well it turns out, it's great for collecting stats in your application. The classic example that I like is using Bitmaps for tracking user activity in your application. For example, let's say you wanted to count how many unique users logged in your app each day.

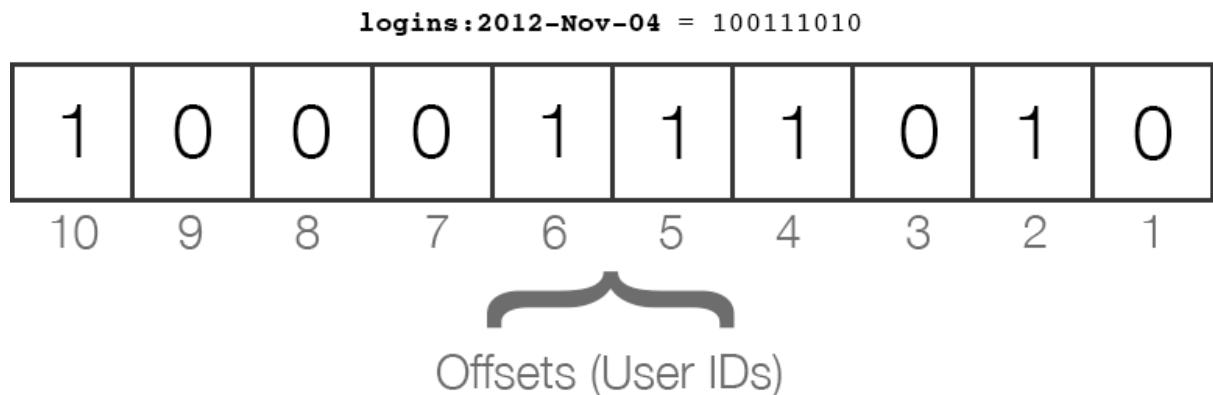
We're going to use a bitmap stored in the key `logins:2012-Nov-04` to track the logins.

```

1 <?php
2
3 function login() {
4     // do your user authentication here
5     $user = User::authenticate($_POST['username'], $_POST['password']);
6
7     if ($user->invalid()) {
8         return false;
9     }
10
11    $key = "logins:" . date("Y-M-d", strtotime("today"));
12
13    $r = new Redis;
14    $r->setbit($key, $user->id, 1);
15 }

```

What this is doing is changing the bit at offset `$user->id` to 1 (from the default of 0) when the user logs in. This ends up being a very memory efficient way of storing true/false values for millions of users- even if you had a million users, `logins:2012-Nov-04` would be less than 1MB in size.



The user IDs are mapped to the offsets, and the operations are very fast. In the image above, we can see that User #10, User #6, User #5, User #4, and User #2 have logged in on November 4th, 2012.

You're probably saying- that's great, but how's that better than just storing a hash or a counter? The power comes in the bit operations that you can perform.

Population Count

Using the `bitcount` we can perform a population count- that is, count the number of bits that we flipped to 1, and very quickly count how many users logged in on a specific day.

```

1 < ?php
2
3 $r->bitcount("logins:2012-Nov-04"); // Returns 5

```

Bitwise Operations

Here is the real power with Bitmaps. You can run any bitwise operations on two different bitmaps and see the results. Imagine you wanted to know which users logged in to both your desktop AND mobile apps on a specific day?

The data would look like:

```

1 web:logins:2012-Nov-04      => 001001001
2 mobile:logins:2012-Nov-04 => 101110001

```

The code would look like this:

```

1 <?php
2
3 $r->bitop("and", "both:logins:2012-Nov-04",
4           "web:logins:2012-Nov-04",
5           "mobile:logins:2012-Nov-04");
6
7 $r->bitcount("both:logins:2012-Nov-04"); // Returns 2

```

On top of that, you have or, xor, and not at your disposal, which you can use to create all sorts of different aggregations very quickly and without using much data.

Redis Notification Feeds

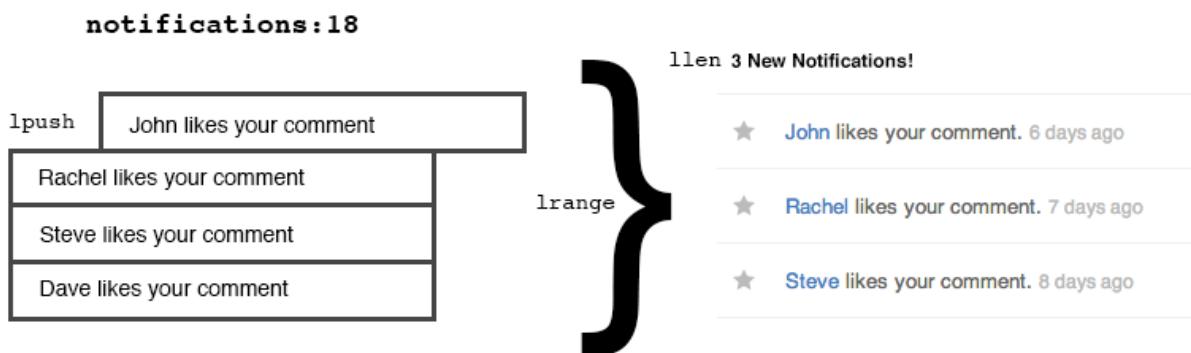
Another popular use of Redis is to create a notification feed for your application. Why use Redis instead of the database? Well notifications tend to be write-heavy throwaway data- once the user views them, there's no point in storing them anymore. Plus they are essentially just lists. A beautiful fit for Redis.

Imagine you wanted to notify a user everytime someone liked a comment that they wrote. You might implement the notifications for it like this:

```

1 <?php
2
3 function notify_of_like($comment, $current_user) {
4     $r = new Redis;
5     $r->lpush("notifications:{$comment->user->id}", "{$current_user->username} 1 \
6 likes your comment.");
7 }

```



Using lpush, we can quickly prepend new notifications into the list of notifications so that they are in a LIFO (last-in, first-out) queue, effectively ordered by creation time.

To pull the latest 20 notifications for a particular user, we just need to use lrange(notifications:user_id, 0, 20). What about a little counter that shows how many notifications they have? That's as simple as llen(notifications:user_id). And once the notifications have been viewed, we can trash them with ltrim(notifications:user_id, 0, 20).

Worker Server: Asynchronous Resque Workers

The overall goal of every website is to render the site as quickly as possible for the end-user. It's been proven time and time again, that faster website equals happier users, more revenue, and more pageviews. The fact is- speed is an important factor and having a faster application *will* factor into your bottom line.

Amazon has [talked about how a 100ms speed difference improved their revenue by 1%⁸⁵](#). No big deal right? In 2011, Amazon had \$50 billion in revenue, so speeding up their site by 100ms improved their bottom line by \$50 million.

That's great. And so far, we've talked about strategies to improve speed through optimization- increasing database performance, caching slow queries, and moving to more scalable pieces of software. But what about when there's a fixed cost, a piece of code that no matter how much you optimize, will always take seconds to run? Maybe a computational or database heavy query that's as optimized as it's going to get. Or a 3rd party API call to Twitter or Facebook. How can this type of workload scale without causing slow pages?

The answer is to do the work in the background, of course! When I say "in the background", I mean in a separate process, outside of the web request. The work gets deferred for later so that the web request can render the page for the user as quickly as possible.

Why use Queues?

There're a bunch of subpar ways to push work into the background. I know what you might be thinking- why would I want to run an entirely separate piece of software to manage my queue when I could just use this find MySQL database server that I have? Create a new table, store the jobs in it, delete them when done?

No! No! Wrong! Don't do it!

Repeat after me. The database is not a queue. The database is not a queue. Why? Learn from my mistakes, because I started out using MySQL as a queue and it was terrible.

Pretend we created a MySQL table called `jobs` to handle our queuing system. If we were inserting jobs to post Facebook statuses in the background for our users, it might look like this:

<code>id</code>	<code>user_id</code>	<code>text</code>	<code>job_state</code>
1	1	Post this FB Status in the Background	finished
2	8	Some status here!	running
3	3	Scaling PHP Book is Awesome!	running
2	8	Steve needs better examples!	pending

⁸⁵<http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it>

In our web request, when we wanted to post a new Facebook status using the Facebook API, we'd `INSERT` a new row into the `jobs` table with the text for the status, `user_id` of the user, and mark the `job_state` as `pending`. So far so good.

Next, we'd have a worker process. It'd be a cronjob, running in the background. It'd search for pending jobs in MySQL to run so it could post the statuses to Facebook. We can imagine it'd look something like this:

```
1 <?php
2
3 $m = new mysqli("localhost", "user", "pw");
4
5 // Loop until we're out of jobs to run
6 while(1) {
7
8     // Find a job with the state set as pending
9     $job = $m->query("SELECT * FROM jobs WHERE job_state='pending' LIMIT 1");
10
11    // Quit the loop if we didn't find a job
12    if ($job->num_rows != 1) break;
13
14    // Update the job state to running so that no one else can nab it
15    $m->query("UPDATE jobs SET job_state='running' WHERE id = {$job['id']}");
16
17    // Post the status to Facebook
18    post_to_facebook($job['user_id'], $job['text']);
19
20    // Mark the job as finished.
21    $m->query("UPDATE jobs SET job_state='finished' WHERE id = {$job['id']}");
22
23 }
```

So far so good, right!? **Wrong**. This solution is broken because it can't scale past more than one worker. Why? We have a major race condition between finding the job and changing the status from `pending` to `running`. When a worker runs the query `SELECT * FROM jobs WHERE job_state='pending' LIMIT 1`, it's possible that another worker can also get the same job in the milliseconds before the following `UPDATE` query changes `job_state` to `running`. With this race condition and more than concurrent worker, Facebooks statuses would get double posted! Catastrophe.

You can solve the race condition by using Locks, but it's messy, slow, and just isn't a very good solution. Use the right tool for the right job.

Getting started with Resque and php-resque

Resque is a queueing *library* open-sourced by [GitHub⁸⁶](#) that uses Redis Lists as primitives to create a reliable, fast, and persistent queueing system. Originally written in Ruby, Resque has been [ported to PHP⁸⁷](#) and is hands down the best way to do queueing in PHP.

Best of all? Since Resque is *just a library*, we don't have to deal with another software service to setup. It just piggybacks off of our Redis installation, with no other software to install! Woohoo.

Of course, there's a beautiful Resque UI that gives you really nice interface into your system so you can see what's going on, monitor failure, and retry failed jobs. I recommend using the UI, but it's not a requirement.

Installing Resque and php-resque

If you don't want the Web UI, there is nothing to install. All you need to do is hook the [php-resque⁸⁸](#) library up to your code. It's on GitHub, so you can pull it into your application anyway that you'd like, but I highly recommend that you use [Composer⁸⁹](#).

Add php-resque to your `composer.json` file

```

1  {
2      "require": {
3          "chrisboulton/php-resque": ">=1.2"
4      }
5  }
```

And install the package

```
1  > php composer.phar install
```

Optionally, you can install the PECL `procitle` extension, which will give you more informative output when you check your Resque process status from the command line.

```

root@server01:~# ps ax | grep resque
9528 pts/0    Sl    0:00 /usr/bin/ruby1.8 /usr/bin/god start resque -c resque.god
9537 ?        Ss    0:00 resque-1.0: Forked 9597 at 2012-11-06 04:23:24
9597 ?        S    0:00 resque-1.0: Processing twitter_replies since 2012-11-06 04:23:24
9601 pts/0    S+    0:00 grep --color=auto resque
root@server01:~#
```

Process 9597 is processing the `twitter_replies` queue

⁸⁶<https://github.com/blog/542-introducing-resque>

⁸⁷<https://github.com/chrisboulton/php-resque>

⁸⁸<https://github.com/chrisboulton/php-resque>

⁸⁹<http://getcomposer.org/>

```

1 > pecl install -f proctitle
2 > vi /etc/php5/conf.d/proctitle.ini
3     extension=proctitle.so

```

If you want the option Web UI, you need to install the Resque RubyGem.

```

1 > apt-get install rubygems
2 > gem install resque

```

Setting up the Workers

With the setup so far, you can queue up background jobs but you don't have anyway to run them. Later in the chapter, we'll talk about the specifics of creating a new job, but we still need a process that runs in the background to handle running the jobs as they come in. Long term, I recommend using `god` or `monit` to handle monitoring the background jobs, but we can start the process manually from the command line. (In fact, the `php-resque` project includes a base `monit` configuration to get you started).

```

1 > VERBOSE QUEUE=* COUNT=2 APP_INCLUDE="..../index.php" php resque.php

```

Run this way, this command would run a maximum of 2 workers, check all queues, and include your applications init/autoload from `..../index.php`. The different values are discussed below.

COUNT The `COUNT` value is used to set the maximum number of worker's that will be run concurrently. Depending on the nature of your jobs (are they CPU bound? I/O bound?) you should set this *atleast* as high as the number of CPU cores on your machine. If your work is database or I/O bound, you can usually set it much higher. **##### QUEUES** With the `QUEUES` variable, we can set the queues that we want this process to work on. Why would we do that? It gives you more control over the queues, allowing you dedicate more resources to certain queues and set priorities. Each queue is checked in alphabetical order, so a queue whose name begins with `a` will be checked before a queue that begins with `z`. By manually specifying the queue names, you can have specific queues checked first. If you set the value to `*`, it will process all of the queues.

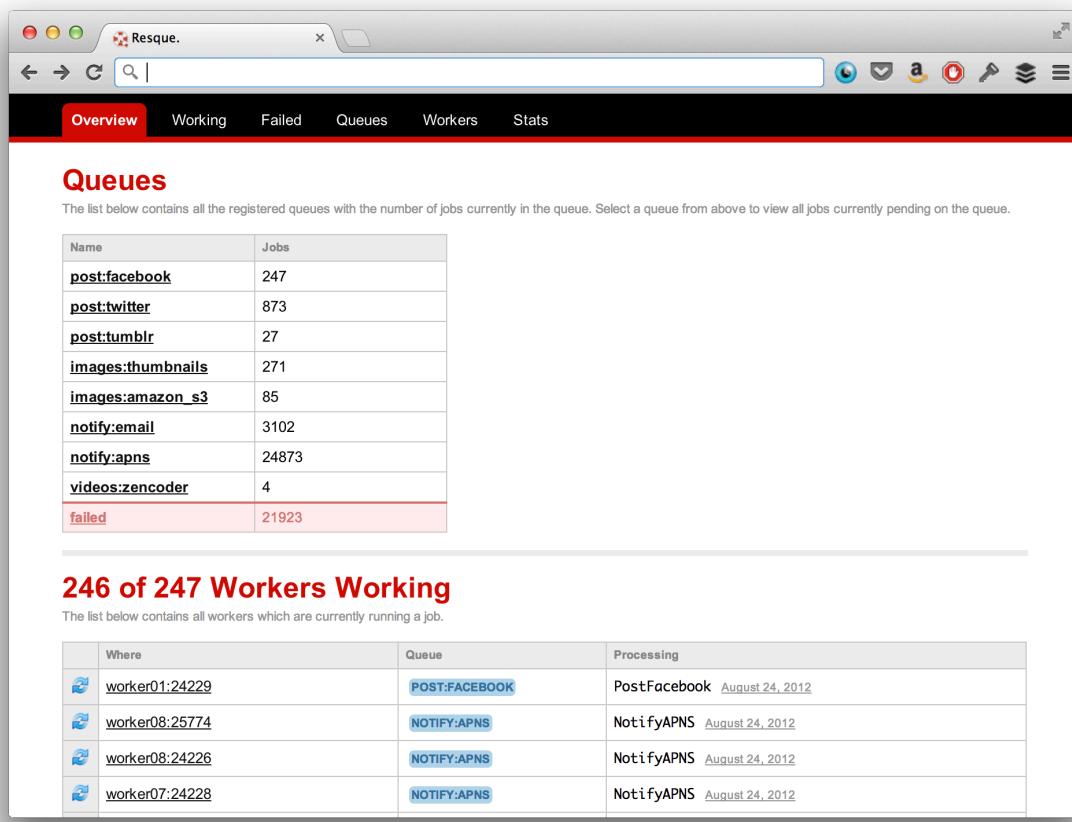
Example value:

`QUEUES=high,medium,low,post:facebook`

APP_INCLUDE Lastly, (and most importantly), you need to tell `php-resque` how it can autoload your code. Out of the box, it has no idea where to find your code that needs to be run, so you need to point it to a file that does your autoloading and initialization. With most frameworks, this can be your `index.php` file.

Resque Web-UI

Resque comes with a lightweight Web-UI that runs using a tiny Sinatra server. You can start it up by just running `resque-web` and it'll start running in the background on port 5678. `resque-web` expects Redis to be running on localhost, port 6379, if you need to point it to a different Redis server, you can pass it in using a CLI argument like so: `resque-web -r 192.150.0.10:6379`.



The screenshot shows a web browser window titled 'Resque.' with the following content:

- Queues:** A table showing registered queues and their job counts:

Name	Jobs
post:facebook	247
post:twitter	873
post:tumblr	27
images:thumbnails	271
images:amazon_s3	85
notify:email	3102
notify:apns	24873
videos:zencoder	4
failed	21923
- 246 of 247 Workers Working:** A table showing currently running workers:

Where	Queue	Processing
worker01:24229	POST:FACEBOOK	PostFacebook August 24, 2012
worker08:25774	NOTIFY:APNS	NotifyAPNS August 24, 2012
worker08:24226	NOTIFY:APNS	NotifyAPNS August 24, 2012
worker07:24228	NOTIFY:APNS	NotifyAPNS August 24, 2012

An example of the Resque UI

One area where Resque really shines is the way that it handles failure. Namely, it was built with failure on mind- Jobs will fail, and it's important to have a complete visibility into the system. One way that it gives you this visibility is by tracking failed jobs and giving you the relevant information- what the job was, why it failed, and the ability to retry the job manually.



The screenshot shows a failed Resque job with the following details:

- Worker: [worker02:22174](#) on [NOTIFY:APNS](#) at [September 25, 2012](#)
- Class: [NotifyAPNS](#)
- Arguments: `{"id"=>"4664803", "notification"=>"New Message!", "badge"=>1, "attempts"=>1}`
- Exception: [DeviceNotFoundException](#)
- Error: [Device UDID Rejected by APNS Service](#)

An example failed Resque Job

The Resque Process Model

Resque has an interesting process model behind it that it uses to handle memory leaks and failure, so it can run reliably for long-periods of time. When you start Resque, a parent (or master) process is created. The parent process polls Redis for new jobs that have been added to the queue from the client. When a new job is found, the parent process forks a new child process and the child process runs your code to handle the job. After the job finishes, the child process exits.

Why this model?

Forking child processes to run each job makes sense from a concurrency standpoint- since PHP doesn't have threads, it's the best way to handle multiple jobs concurrently.

On top of that, this model also handles failure very well. Since each running job is isolated into its own process, if it crashes, throws an exception, or leaks memory, it won't impact other jobs or the parent process. This is important, since PHP isn't really designed for long-running processes and could very well start leaking memory. Since the child process is thrown away at the end of the job, any leaked memory can be reclaimed at the end of the job.

The downside to this model is that there is a fixed overhead for running each new job that comes in. Forking a child process is an expensive system call and puts a hard limit on the amount of new jobs/second that can be run concurrently, and can be a significant problem if you have many small, short-lived jobs. The Ruby world has overcome this with [Sidekiq](#)⁹⁰, a fork of Resque that uses threads for running jobs instead of the Parent/Child model. Unfortunately, PHP doesn't support threads, but I think that we'll eventually see a hybrid approach where child processes are used for multiple jobs and recycled after a certain amount of time- effectively giving you the best of both worlds.

My Patches

Originally, when `php-resque` first came out I was using my own fork of it in production. One design decision that `php-resque` made was bundling a Redis Library with the software. Unfortunately, the library that they bundled, `Redisent` did not use namespaces or take into consideration naming conventions of other Redis Libraries (namely, `phpredis`). Because of this, `Redisent` would clash with `phpredis`, both libraries trying to define a `RedisException` class, causing PHP to fail out with an error.

Originally, I changed the `php-resque` library to use C `phpredis` extension, and eventually moving over to the C extension seems to be one of the long-term goals of the project. Right now, they still bundle `Redisent`, but luckily have merged a patch that only defines `RedisException` if it doesn't exist yet- atleast allowing the two libraries to play nice with each other.

The new code that they merged looks like this- it's amazing how many headaches it solves.

⁹⁰<https://github.com/mperham/sidekiq>

```

1 <?php
2
3 if (! class_exists("Redis")) {
4     class RedisException extends Exception { }
5 }
```

My original patched version of `php-resque` is on [GitHub as `stevecorona/php-resque`](https://github.com/stevecorona/php-resque)⁹¹. I wouldn't recommend using it unless you really need the performance of the C Redis Library.

Writing a Resque Job

Let's step through the process of writing a Resque job. Pretend we have an application, it posts a status to Twitter. The naive, non-Resque version might look something like this:

```

1 <?php
2 class Comments_Controller extends Controller {
3     public function create() {
4         $user = $this->session->user;
5
6         $twitter = new Services_Twitter($user->username, $user->password);
7         $twitter->statuses->update($_GET['comment']);
8
9         return new View("Your comment has been posted to Twitter");
10    }
11 }
12 }
```

This code works great, but it suffers from some problems. First off, it's slow because an API request to Twitter takes atleast 1 second. It's also failure-prone- if Twitter is down or has momentary errors, the status won't get posted. On top of that, it's an easy DoS endpoint that can be used against your site. If I wanted to crash your app, I could just point my bots at this endpoint and eat up all of your PHP-FPM workers, because the API call blocks for so long. This actually happened to Twitpic, before- there's a case study included on the topic.

Anyways, let's refactor it into a Resque job and fix all of these issues.

Enqueuing the Job

First let's handle the frontend code, we need to refactor the code inside of the `create` action. Instead of posting to Twitter directly from the web request, we want to instead setup a job that can be run later.

⁹¹<https://github.com/stevecorona/php-resque>

```
1 <?php
2 class Comments_Controller extends Controller {
3     public function create() {
4
5         $args = array("user" => $this->session->user->id,
6                       "comment" => $_GET['comment']);
7
8         Resque::enqueue("post:twitter", "PostTwitter", $args);
9         return new View("Your comment has been posted to Twitter");
10    }
11 }
```

That was easy. Resque::enqueue does all of the heavy lifting for us. It takes three arguments, by the way- post:twitter is the name of the queue that we want the job to go into, PostTwitter is the name of the class that we need to define with the code to run the job, and \$args is an array of data that will get serialized and passed into the PostTwitter class.

The immediate benefit to switching to this code is that it will run *fast*. The time it takes to enqueue the data in Redis/Resque is orders of magnitude less than the amount of time it takes to make an API call.



A note about \$args

Keep in mind that anything inside of \$args will be serialized inside of Redis when you create the job. It's much better to pass in simple values here instead of full-blown objects. Notice how we pass in \$this->session->user->id instead of the entire user object. Using the id, we can rebuild the user object from the database inside of PostTwitter.

Running the Job

Now that we have the queuing portion of our code refactored, we need to write a new class called PostTwitter that php-resque will instantiate when our job is processed. php-resque expects job classes to implement the perform() method, which is where we put the posting to Twitter code.

Note: The \$args array passed in from the frontend will be available in the PostTwitter class as \$this->args.

```

1 <?php
2 class PostTwitter {
3     public function perform() {
4
5         $user      = User::find($this->args['user']);
6         $comment   = $this->args['comment'];
7
8         $twitter = new Services_Twitter($user->username, $user->password);
9         $twitter->statuses->update($comment);
10
11    }
12 }
13 }
```

Let's talk about the advantages for a second.

1. The application will always offer constant, fast performance because it's never waiting on a third-party service.
2. The easy DoS endpoint is gone
3. If Twitter doesn't respond or times out, the `Services_Twitter` will throw a `Services_Twitter_Exception`, causing the job to fail gracefully (which can later be retried).

Handling Failure

Appropriately handling failure in Resque is important. If an uncaught exception is thrown from your job, `php-resque` will assume that the job has failed and remove it from the queue. As we saw above, the nice thing is that the failed jobs are shown in the Resque Web UI with plenty of information to help you diagnose the failure and retry the job. This is very helpful when you have jobs that fail rarely and only in the case of a bug or edge-case.

However, sometimes we have jobs that predictably fail and should be retried automatically. For example, in the case of the `PostTwitter` class, if Twitter times out or happens to be momentarily unreachable, we don't want to fail the job immediately. We can accomplish automatic retries by adding a little bit more code.

```

1 <?php
2 class PostTwitter {
3
4     const MAX_ATTEMPTS = 5;
5
6     public function setUp() {
7         Resque_Event::listen("onFailure", function($ex, $job) {
8
9             // Check to see if we have an attempts variable in
10            // the $args array. Attempts is used to track the
11            // amount of times we tried (and failed) at this job.
12     }
13 }
```

```

12     if (! isset($job->payload['args']['attempts'])) {
13         $job->payload['args']['attempts'] = 0;
14     }
15
16     // Increase the number of attempts
17     $job->payload['args']['attempts']++;
18
19     // If we haven't hit MAX_ATTEMPTS yet, recreate the job to be
20     // run again by another worker.
21     if (self::MAX_ATTEMPTS >= $job->payload['args']['attempts']) {
22         $job->recreate();
23     }
24
25 });
26 }
27
28
29 public function perform() {
30
31     $user      = User::find($this->args['user']);
32     $comment   = $this->args['comment'];
33
34     $twitter = new Services_Twitter($user->username, $user->password);
35     $twitter->statuses->update($comment);
36
37 }
38 }
39 }
```

You can see above that we didn't modify our `perform` method at all. Instead, we added a `setUp` method, which gets called during the job creation process, and used the `Resque_Event` callback system to add in a hook that only gets run after a job fails. So, when our Twitter Library throws a `Services_Twitter_Exception`, the job will fail and call the anonymous function that we added to `onFailure`. Our `onFailure` callback checks for the existence of an `attempts` variable and recreates the job as long as it hasn't hit `MAX_ATTEMPTS`. The reason we limit the number of attempts is to prevent runaway jobs and limit the amount of failure that can happen.

In practice, using a setup like this gives a good balance between manually handling job failure and dealing with it automatically. Some errors can simply be fixed by trying the job again, especially if they are dependent on an external API.

Scheduling Jobs

In the classic Resque setup, jobs are run FIFO (first-in, first-out) order, with no specific timing. This usually works well if you want your jobs to be run immediately, but doesn't fit all use cases if you want jobs to have a cool down period or be run around a specific time instead of immediately.

You can hack something together with cron, but instead of doing that I recommend checking out the [php-resque-scheduler plugin⁹²](https://github.com/chrisboulton/php-resque-scheduler). It gives you the ability to create jobs for future times instead of right now.

Post to Twitter in 1 hour

```
1 ResqueScheduler::enqueueIn(3600, 'post:twitter', 'PostTwitter', $args);
```

Post to Twitter at a specific time (4PM on 11/19)

```
1 $date = new DateTime('2012-11-09 16:00:00');
2 ResqueScheduler::enqueueIn($date, 'post:twitter', 'PostTwitter', $args);
```

Alternatives to Resque

There's a never ending list of alternatives for queueing solutions. For 99.9% of cases, Resque is simply going to be the best choice.

1. It's built on Redis, so it's fast, reliable, and has constant performance.
2. With Redis RDB and AOF, queues are persistent to disk- you won't lose anything in a crash.
3. The library is well-tested and heavily used, so you don't have to worry about bugs.
4. There is a high amount of visibility into the system.
5. It's unlikely that you'd need to scale your Redis server, but if you had to you can take advantage of the normal scaling methods- sharding and master/slave replication.
6. Overall, it's **simple**. The complexity of the entire system is low.

Kestrel

[Kestrel⁹³](https://github.com/robey/kestrel) is a queuing system written by Twitter in Scala. Being in Scala, it runs on the JVM and is pretty fast. In fact, it was our first choice at Twitpic because it came out way before Resque was even around.

Pros

- Runs on the JVM, so it's reliable and fast.
- Uses Memcached API, so you use Memcached client to interact with it
- *Fan-out* queues. The idea is that you insert a job into one queue and it automatically distributes it to many other queues. Great if you are putting the same job into different queues.

Cons

- Memory hog, JVM GC pauses cause inconstant response time
- Library is lower-level than Resque, have to roll-your-own functionality
- Infrequent updates by Twitter team

⁹²<https://github.com/chrisboulton/php-resque-scheduler>

⁹³<https://github.com/robey/kestrel>

Gearman

[Gearman](#)⁹⁴ is a PHP extension and daemon for queuing, and thus it has very tight PHP integration. It's used by both Flickr and Digg.

There really aren't any advantages to using Gearman over Resque. It uses its own proprietary server, so I ask- why add more layers to your stack when you can reuse existing pieces since you're likely to be using Redis for caching or sessions anyways?

Beanstalkd

[Beanstalkd](#)⁹⁵, is another queuing server written in C. It has a handful of PHP clients. Similar to Resque but with a more complex API, including better priority control, pausing jobs, and job scheduling.

If you *need* these features, it's worth checking out but there's less documentation available than with Resque.

RabbitMQ

RabbitMQ is one of the faster (if not THE fastest) messaging queue available. It's written in Erlang and I've even heard of people that are pushing streaming video through it. Crazy. It can push 100k messages per second, per CPU core.

It's also probably the most robust messaging queue out there, too. It includes every feature under the sun. The downside is that it's *complex*. Setting it up, configuring it, and integrating into your code is a big undertaking. There are *books* on RabbitMQ. That should tell you something.

If you need to process 100,000s of background jobs per second than use RabbitMQ. Otherwise- don't. Most sites- even big ones like Twitter and Twitpic are able to run on Kestrel and Resque. Choose the simpler solution here.

Have a weird use case? Sure, use RabbitMQ if it seems like it fits but just expect to dump some engineering and learning time into it. Meanwhile, you can have Resque up and running in 5 minutes.

Things you should put in a queue

Here are some common things that you should put into your queue.

- Storing user uploads on S3? Do that in a queue.
- Resizing images
- ANY 3rd Party API Call- Twitter, Facebook, Tumblr. Always.
- PDF Generation
- Sending emails- SMTP servers can timeout.
- Heavy database inserts (Twitter-like social networks pre-calculate timelines. That means, if you have 500 followers, when you post a new tweet Twitter does 500 database inserts.)

⁹⁴<http://www.gearman.org>

⁹⁵<http://kr.github.com/beanstalkd/>

Coding and Debugging at Scale

We've gone through every layer of the stack and we've finally made it down to the most uninteresting layer- the PHP code! Everyone wants to know tricks and cheat codes for optimizing PHP. Does this code run faster than that? Well, let's settle some debates.

Scaling your code

Everyone wants to know "scaling" tactics. They think that big sites are doing things like using `for` loops instead of `foreach` because they're marginally faster or using `"` instead of `'` for strings. It's just not the case. Microoptimizations are the pick up lines of scaling— they don't work and they make you look like a fool. Scaling your code is the easiest part of scaling the stack- you just add more application servers. It's completely horizontal. As indulging as it seems, you'd gain far bigger payoffs by optimizing your database or cache instead of trying to squeak out 1 or 2% performance gain in your code. If raw speed is that important- just throw money at the problem. More application servers and more GHz (remember- PHP is single threaded- more GHz, not more cores, will improve response time).

Scale algorithms, not syntax (but write clean syntax anyways)

Here's the bottom line- none of the code stuff really matters. The very *marginal* speedup that you'd gain is totally flushed down the toilet as soon as you hit the database or cause the garbage collector to run. I guarantee you'll find 15 low-hanging infrastructure optimizations for every single code-path optimization you find.

I'm not saying to write bad code- write **sane** code and optimize the blatantly poor code but don't go chasing the 2% stuff. The most obvious offenders (as with any programming language) are-

Nested Loops (Big O and Algorithm Analysis)

If you need more than 3 levels of indentation, you're screwed anyway. - Linus Torvalds

If you can code around it, try to avoid a loop inside of a loop. Simple enough- more loops, more code being run. If you know **Big O/Algorithm Analysis**⁹⁶, you want to avoid $O(n^2)$ and $O(n^3)$

An Example of $O(n^2)$:

⁹⁶http://en.wikipedia.org/wiki/Analysis_of_algorithms

```

1 $foo = range(0, 100000);
2 $bar = range(0, 100000);
3
4 foreach ($foo as $f) {
5     foreach($bar as $b) {
6         // This code will run 100000 * 100000 times
7         // You'd want to focus your attention on optimizing
8         // this code, ideally removing the loop.
9     }
10 }

```

Objects or arrays?

Common sense says that arrays should use less memory than an object. There is just more “stuff” that tags along for the ride when you use an Object versus a list of values in an array.. right?

Not so in PHP 5.4— Objects can actually optimize the way they store data and end up with a smaller memory footprint than arrays. Convenient for both performance AND readability. Gone are the days of using arrays over objects for “performance reasons.” An in depth post on the subject can be found [here⁹⁷](#).

Use the standard library, luke

Just quickly worth talking about- when possible, always prefer using built-in standard library functions over your own functions, if possible- remember, the PHP standard library is written in C so it is much faster than PHP code. This especially applies to the `array_` functions, which can often be 3-10x faster than the same algorithm coded in PHP.

The True Evil of Premature optimization

Premature optimization is a true evil. I hope that if you’ve learnt one thing from this book, it’s to make smart scaling decisions and shoot for scaling out the big pieces that are going to improve performance the most. There’s a lot of bad advice on the internet from archaic experts that give scaling advice like “never use temporary variables! change double quotes to single quotes, they’re faster!”. As a smart reader, I know you’re not falling for traps like that, but it can be hard to wade through all of the BS.

Except for rare circumstances, where you’re doing something really inefficient, your PHP code will likely never be the cause of scaling issues. PHP’s virtual machine tends to be pretty poor performance wise, but because the PHP standard library is written in C, your code still tends to run fast.

Note- I said your *CODE*, not your *architecture*. But swapping out double quotes for single quotes won’t make any difference. You’re a smart reader, though, so I know you won’t be falling for those traps.

⁹⁷<https://gist.github.com/nikic/5015323>

To Framework or not to framework?

Choosing whether or not to use a framework is a no-brainer. I've never heard of a site failing because they chose to use a framework. If you look any other language for the web- Python, Ruby, Java- all of these guys are using MVC frameworks like Django, Rails, and Play.

PHP seems to have a different sentiment in it's community, though. People say frameworks are slow. Or that they're bloated. Or too complex. While some of this may be true, it doesn't represent modern PHP frameworks- I think the hate is just a defense mechanism for guys that don't understand or "get" MVC.

You can use one of the "traditional" PHP frameworks that have been around forever like Zend, Kohana, Symfony, or CakePHP, but alternatively check out some of these newer frameworks that tend to be more modern and "lightweight".

- [Laravel⁹⁸](http://laravel.com/)
- [FuelPHP⁹⁹](http://fuelphp.com/)
- [Fluf PHP¹⁰⁰](http://nijikokun.github.com/fluf/)
- [Slim¹⁰¹](http://www.slimframework.com/)

How about DIY frameworks?

So you want to build a DIY framework for your product? Don't do it- learn from my mistakes!

Managing your own framework is like building two products instead of one. Not only do you have to worry about coding your application, but you now also need to deal with writing all of the underlying framework code for it too. This is likely the difference between failing miserably and succeeding.

Truthfully, I fell into this trap with the second version of Twitpic. We built out our own framework. It was difficult- not because writing a MVC framework is hard, but because handling all of the edge cases, gracefully failing, and covering all of the features we needed takes a lot of code.

It made development painful. Want to add a new feature to the app? Let me just code it up... oh wait, we're missing a function from the framework to do that, so before I can add that new feature to the app, I need to fill in the missing framework code. It slows you down.

If you want to build a framework to get a better grasp on how they work- that's great. But sure as hell don't use it in production. Learn from my mistakes- you'll kick yourself in the butt later.

ORMs aren't evil

Related to frameworks, let's talk about ORMs for a second- usually the "Model" part of your application. Plain and simple, they are very useful- ORMs simplify your code and make it easier because you can just call methods instead of having to write SQL.

⁹⁸<http://laravel.com/>

⁹⁹<http://fuelphp.com/>

¹⁰⁰<http://nijikokun.github.com/fluf/>

¹⁰¹<http://www.slimframework.com/>

But.. but.. if I can't write SQL, then it's going to be inefficient! In my experience, most mature ORMs generate almost the exact same SQL that I would have written. Sure, there are some edge cases around JOINs that might cause problems, but most ORMs let you pass in your own SQL to handle those situations.

At Twitpic, we use an ORM library and I wouldn't ever think about not using one. The benefits outweigh the slight performance overhead- and to be honest, most of the "controversy" surrounding ORMs comes from pre-mature optimizers and is often incorrect. Our ORM allows us to pass in raw SQL for complex-to-generate queries, but if I had to do it again I would leave this feature out- as we've scaled our architecture, we've optimized most production JOINs out, so any off-the-shelf ORM would have done.

I'm not saying never write SQL. For example, ORMs usually don't handle updating or deleting multiple objects very well (also called N+1 queries). Consider something like this-

```

1 < ?php
2
3 // I want to delete all of the "notes" this user has added.
4
5 $notes = Note::find_by_user_id($user_id);
6
7 // Loop over each note and delete it, effectively generating
8 // DELETE FROM notes WHERE id = $note_id
9 foreach($notes as $note) {
10     $note->delete();
11 }
```

The example above will generate and execute a SQL statement for each note that exists. If you were doing your own SQL, you could have just executed `DELETE FROM notes WHERE user_id = $user_id` instead and handle the whole delete in a single query. This isn't usually a big deal, but if you're deleting (or updating), say, 5000 items, it can have a meaningful impact. Luckily, it's not a common design in most web applications, but this is a case where I'd just write my own SQL instead of relying on an ORM.

Although most frameworks package their own ORM, if you aren't using a framework or have your own in-house framework, there are a few open-source standalone ORM packages that you can use. Similar to recommending against DIY frameworks, I also don't recommend DIY ORMs, as they are especially complex and have many edge cases to consider. Here are some modern options-

- [PHP ActiveRecord¹⁰²](#)
- [Propel¹⁰³](#)
- [Doctrine¹⁰⁴](#)

¹⁰²<http://www.phpactiverecord.org/>

¹⁰³<http://propelorm.org/>

¹⁰⁴<http://www.doctrine-project.org/>

Capistrano for code deployment

Hopefully you're not still stuck deploying code by dragging and dropping over FTP, or even worse- editing files on your production machine! If you are, you need to get yourself some [Capistrano](#)¹⁰⁵. Besides being horribly inefficient and prone to failure, manual deployment doesn't scale well past one server. Capistrano fixes that. You run a command on your computer (cap deploy) and it deals with pushing new code to all of your servers.

And it's easy to setup, you can be rolling in the next 10 minutes.

You need to have Ruby installed on your computer. Capistrano is language-agnostic, but itself is written in Ruby.

You're installing Capistrano on the computer you're deploying from, i,e your development machine. Not your application server.

```
1 > gem install capistrano
```

Now, go into your project and make a new file named `Capfile`.

```
1 load 'deploy' if respond_to?(:namespace)
2 load 'app/config/deploy' # The path to your deploy.rb file
```

Your `Capfile` is read by Capistrano and tells it where to find `deploy.rb`, the configuration file for your deployment options. In my case, my `deploy.rb` file is located at `./app/config/deploy.rb`, but you can change the second line of the `Capfile` to wherver you want to put your `deploy.rb` file.

Next, we need to create the `deploy.rb` file. Mine is in `app/config/`, but you can put it anywhere as long as you modify the second line of your `Capfile`.

You can do ALOT with Capistrano. The most basic use case is this-

1. Run `cap deploy` on your computer.
2. Capistrano logs into your application server(s), usually over SSH.
3. Capistrano clones the latest version of your trunk/master branch (both SVN and Git are supported)
4. Reloads PHP-FPM to get the latest code changes

That's the basic use case, and that's all our basic `deploy.rb` file will do (we'll strip out some of the default Rails options). But here are some other ideas that you can use it for-

- Minify/Compile your Javascript, LESS, Coffeescript, SASS
- Run database changes, i.e, adding new columns or new indexes on deployment
- Tail the log files of all your servers
- Deploy to multiple environemnts (production, development, staging)

Example `deploy.rb` file

¹⁰⁵<https://github.com/capistrano/capistrano>

```

1 set :application, "My Application"
2 set :repository, "git@github.com:MyApp/app.git"
3 set :branch, "master"
4
5 set :deploy_to, "/u/apps/myapp/"
6
7 set :deploy_via, :remote_cache
8 set :scm, :git
9
10 # SSH User to Deploy As
11 set :user, 'deployer'
12 set :password, '123456'
13
14 set :use_sudo, false
15 set :keep_releases, 5
16
17 ssh_options[:forward_agent] = true
18
19 # Hostnames of your App Servers
20 role :app, ['app01', 'app02', 'app03']
21
22 namespace :deploy do
23
24   task :finalize_update do
25     # This strips out some of the default rails
26     # tasks that we don't need for PHP.
27   end
28
29   # Optional- Reload PHP-FPM after the code is deployed
30   task :restart, :roles => [:app] do
31     run "#{sudo} service php5-fpm reload"
32   end
33
34 end

```

Atomicity and Rollbacks

If you're not sold on the ease-of-use that Capistrano adds to your development flow- consider this.

Capistrano deployments are atomic. That means that while it's deploying code, it doesn't impact any currently running code.

Think about it: if you have a website that you manually deploy to, when you upload files to it over FTP or manually run `git pull`, while it's transferring the new code to your remote server, there are old files and new code files mixed together. Like, if you changed both `foo.php` and

bar.php, since the files are transferred sequentially, for a split moment the old foo.php and the new bar.php will be running on the server at the same time.

This can cause all sorts of hairy situations- crashes, exceptions, and data loss, because you're running a mix of both old and new code, who knows what the outcome will be. It gets exponentially worse, the more files that you have.

Capistrano works differently, though.

1. PHP-FPM is told to read code from a symbolic link, let's call it /u/apps/my_app/current.
2. When you deploy new code with Capistrano, it creates a new directory in /u/apps/my_app/releases (i.e., /u/apps/my_app/releases/749847589) and downloads your code into that folder.
3. Once *all* of the code has been transferred, it moves the symlink of /u/apps/my_app/current to the new directory it created- /u/apps/my_app/releases/749847589.
4. Jump up and down! Capistrano deployed your code atomically. All of it changed at once, all of the new code works together.

This method also makes it easy to rollback bad deployments. Pretend you deploy some code with a hidden bug that breaks your site (this never happens in real life... right!?). If you were using traditional manual deployment, you'd have to find the old files and transfer them back, adding to the amount of downtime.

Instead, since capistrano keeps each deployment in its own directory, it can just roll back to the previous version by symlinking /u/apps/my_app/current back to the old release. And with just one command!

```
1 > cap deploy:rollback
```

Easy peasy. I could write a whole book on Capistrano and this only touches on the surface- but DO IT! It's a really great tool and a "best practice". Checkout more documentation on the [Capistrano wiki](#)¹⁰⁶.

Live Debugging PHP with strace

I have a confession to make. strace is probably my alltime favorite tool for debugging when shit hits the fan. It's saved my ass more than once and if you learn how to use it, it'll save yours too.

strace is a unix tool that attaches to a currently running process and monitors the system calls that are made by that process- it gives you visibility into how the program is interacting with the OS. strace can be used for a bunch of different things, but it's a life saver for debugging weird issues in production.

Let's pretend that you have some PHP code that uses in Memcached and your Memcached server goes down. Ideally, you'd have some monitoring or logging in place that would tip you off to

¹⁰⁶<https://github.com/capistrano/capistrano/wiki>

this issue, but if you didn't you'd be going in blind- you wouldn't have any visibility into what was broken, your site would just be broke, likely timing out if you don't have proper Memcached timeouts set.

What to do? Well, you could check each Memcached server manually, but this gets let's convenient if you have more than one Memcached server- logging into each one, trying to figure out which one is broken and timing out.

First, we need to find a running `php-fpm` process to attach to.

```

1  > ps ax | grep php-fpm
2
3  11662 ?      S      0:03 php-fpm: pool www
4  11663 ?      S1     0:06 php-fpm: pool www
5  11698 ?      S      0:04 php-fpm: pool www
6  11721 ?      S      0:04 php-fpm: pool www
7  11745 ?      S1     0:11 php-fpm: pool www

```

Ok, we have several `php-fpm` processes running, let's go ahead and trace 11662.

```

1  > strace -r -p 11662
2
3  0.000000 restart_syscall(<... resuming interrupted call ...>) = 0
4  4.720248 socket(PF_INET6, SOCK_DGRAM, IPPROTO_IP) = 3
5  0.000110 close(3)                 = 0
6  0.000145 gettimeofday({1361146314, 442683}, NULL) = 0
7  0.000115 socket(PF_INET, SOCK_STREAM, IPPROTO_IP) = 3
8  0.000086 fcntl(3, F_GETFL)        = 0x2 (flags O_RDWR)
9  0.000097 fcntl(3, F_SETFL, O_RDWR|O_NONBLOCK) = 0
10 30000.000079 connect(3, {sa_family=AF_INET, sin_port=htons(11211), sin_addr=inet
11 net_addr("192.150.0.10")}, 16) = -1 EINPROGRESS (Operation now in progress)

```

Very quickly, we can see on the last line that it's stalling for 30 seconds making a connection to 192.150.0.10 on port 11211, which happens to be one of our 10 Memcached servers. Now we can jump right into that server and fix the problem, since we know it's the culprit.

Another use case for this type of live-debugging is if you're using a 3rd party API that's slowing down your code- you can use `strace` to see how long the connections are taking. This is probably my most often go-to method for jumping into an "oh-shit-the-site-is-down situation".

In the CakePHP Cache Files Case Study, I demonstrate how `strace` was invaluable in figuring out exactly where the bug in my code was.

A real life strace scenario

Last week, I was sitting on my couch, binging on some new episodes on NetFlix.

Ding ... Ding

A text message? This late? Can't be good.

Grudgingly, I pause NetFlix after a few seconds ("How it's made" is super addicting) to check out the messages...

"From PagerDuty: Your API is Down!"

Fuck! An alert telling me that my site is down. EXCUSE ME! I'm over here trying to learn how Nail Clippers are made. Damnit. Off to the computer.

I sit down and try to load up `api.myjob.com` and... nothing. Chrome just sits there, the condescending little blue circle thing spinning into oblivion.

My site is DOWN, and it's down hard. Requests are timing out.

What tactics would you use? Some good answers might be...

- Check out automated alerts, like ones sent from Nagios
- Look at your alerting/graphing systems like Copperegg or Graphite
- Verify the "usual suspects" like Memcache or MySQL are healthy

But that's NOT what I did. Any guesses?

I logged into one of my PHP servers, grabbed one of the PIDs for a PHP-FPM worker, and ran `strace -p`.

If you're not using `strace` on a reoccurring basis, you're doing yourself a MAJOR disservice. It's perfect for quick and dirty debugging. Sure, in an IDEAL world, you'd get a notification telling you exactly which service was down, but I'm a realist and I know that doesn't always happen.

When your site is down, wishing you had better alerting isn't the answer. You need to get your API back up ASAP. Everything else comes second.

So, I log into my webserver over SSH and run `ps` to grab a random PHP-FPM process id, like this:

```
1 $ ps aux | grep php-fpm
2 myapp 128131 2.9 0.0 324132 57396 ? R 10:54 11:50 php-fpm: pool www
```

See 128131? That's a process id I can get a sneak peak into using `strace`.

Next, I attach `strace` to the process and see what's up...

```
1 $ sudo strace -p 128131
2 gettimeofday({1399247077, 563898}, NULL) = 0
3     sendto(17, "\372\0\0\0\3SELECT * FROM users WHERE id =\"..., 254, MSG_DONTWAIT\
4 , NULL, 0) = 254
```

I see it waiting on the last line for several seconds. Just, like that, I can immediately see that it's sending a SQL query to the database server and is waiting for the response.. looks like my database server is fubared again! And just like magic, I'm immediately able to diagnose the

downtime. Strace gives me the power to address the database server faster, fix the problem, and get back to watching “How It’s Made”.

A good friend of mine is a scaling expert...

He’s the kind of guy that can not only program well, but can also talk for hours about hardcore linux internals. I asked him— *“If you could teach everyone just ONE thing about scaling... what would it be?”*

His response? Strace. 1000 times strace. Nothing else can give you so much insight and information about your stack in a single command. You can have all of the monitoring IN THE WORLD, and strace will STILL give you new insights into your application.

Parting Advice

I hope that by the time you've finished this book, you've picked up some solid nuggets of information that you can implement today and use to improve and refine your stack. Scaling and performance optimization is a "life-long" process that you constantly need to work on and refine— what works for you today may not necessarily be the best solution tomorrow as your traffic changes, hardware gets faster and cheaper, and new tools come out.

What I hope that you've learned is that scaling can be complex, but it's not impossible, and that it's more about owning your stack, being intimate with all of the moving pieces, more than writing "faster code". Go forth with your new knowledge and build some massively successful websites ;)

If you purchased the Solo Founder or Startup version of the book, the next 70 or so pages are filled with great case studies of real-life problems that I ran into scaling Twitpic. These case-studies are filled with invaluable information that you can't find anywhere on the internet— far from your average PHP tutorial.

1. Horizontally scaling user uploads
2. How DNS took down Twitpic for 4 hours
3. Scaling MySQL Vertically (10 to 3 MySQL Servers)
4. Hot MySQL Backups w/ Percona
5. Async Posting w/ Kestrel saved my butt
6. The missing guide to Memcached
7. HTTP Caching and the Nginx Fastcgi Cache
8. CakePHP Framework Caching: A lesson in debugging
9. Optimizing image handling in PHP
10. Benchmarking and Load Testing Web Services

Want to upgrade?

If you're regretting only grabbing the PDF, rejoice! You can upgrade and grab the case studies, simply pay the difference. It's completely automated and you'll be able to download and read these 10 awesome case studies immediately.

[Upgrade to PDF + Case Studies¹⁰⁷](#)

The direct link is <https://gumroad.com/l/vmyM>

** Scaling is an art, not a science **

¹⁰⁷<https://gumroad.com/l/vmyM>

Sending Feedback and Testimonials

If you enjoyed this book, I'd *love* to hear from you. Testimonials are welcome and will be featured with a backlink to your website. Send questions, feedbacks and testimonials directly to me at [¹⁰⁸](mailto:hello@scalingphpbook.com) and I'll email you back ASAP.

In too deep? Need help ASAP?

I know what it's like to be up the creek without a paddle— website constantly crashing with huge scaling problems that you needed fixed weeks ago. If you're too busy to implement these changes yourself, or need in-depth evaluation and code-review, I'm available for remote and on-site consultation. Email me at [¹⁰⁹](mailto:hello@scalingphpbook.com) with the subject "HELP!" and I can be digging through your code in less than an hour. I can typically diagnose core issues, provide hot-fixes, and create a long-term growth/scaling plan in about 4 hours. It's powerful and quick.

Updates?

What about updates? I'm dreading the idea of this book ever becoming outdated. While I can't promise to make major revisions (I might, thought, there is a bunch of stuff I wanted to add but had to cut, lest I write a 400-page snorefest), I will try to keep it reasonably up-to-date and address major breaking changes in PHP. For example, I originally wrote this book when PHP 5.4 was released, but have updated it for PHP 5.5.

About the Author

Steve Corona is the CTO of Twitpic, one of the largest sites on the internet, and the biggest photo sharing application for Twitter. Steve dropped out of college in 2008, and grew Twitpic to 60 million visitors and 20 billion HTTP requests with no formal training or knowledge.

¹⁰⁸ <mailto:hello@scalingphpbook.com>

¹⁰⁹ <mailto:hello@scalingphpbook.com>

Case Study: Horizontally scaling user uploads

Similar to cookie handling, another issue that people commonly run into when scaling their application servers is dealing with user uploads. When we talk about user uploads, I have the following requirements in mind:

1. Uploads need to be available immediately

In most cases, users expect their uploaded files to be available immediately. Showing a broken image or unavailable file is typically unacceptable in most circumstances (video being the obvious exception, due to the processing time involved). If we move the uploaded file around the system, it needs to be entirely invisible, and seamless to the user.

2. Uploads get pushed to long-term storage in the background

Storing the uploaded data on your application servers for more than a few minutes is dangerous and irresponsible. We've designed our application servers to scale horizontally, to **fail without losing data**, so using them for long-term storage violates these principles. Hard-drives crash and running a RAID-10 on your application servers isn't cost effective.

There are plenty of *easy* long-term storage solutions that can be used depending on your particular needs. We store petabytes of data in [Amazon S3¹¹⁰](http://aws.amazon.com/s3/), but other options include [Rackspace Cloud Files¹¹¹](http://www.rackspace.com/cloud/public/files/) and [Edgecast Storage¹¹²](http://www.edgecast.com/services/storage/).

If you have enough data to make an IaaS solution too expensive (or just can't/don't want to use one), there are "roll-your-own" solutions available like [MogileFS¹¹³](https://github.com/mogilefs) and [GridFS¹¹⁴](http://www.mongodb.org/display/DOCS/GridFS) that are worth exploring.

Remember, a major principle in scaling is to decouple your systems! Application servers should serve your application, not store data.

P.S. Don't even THINK about storing the uploaded files in your database. Use the right tool for the right job.

3. No single points of failure or centralization (kids, say no to NFS!)

There should be no single points of failure in your file upload system. Failure will happen and it needs to be handled gracefully. Uploads should be processed immediately and not by

¹¹⁰<http://aws.amazon.com/s3/>

¹¹¹<http://www.rackspace.com/cloud/public/files/>

¹¹²<http://www.edgecast.com/services/storage/>

¹¹³<https://github.com/mogilefs>

¹¹⁴<http://www.mongodb.org/display/DOCS/GridFS>

a centralized system. I avoid NFS like the plague because it's typically very slow and creates a devastating single point of failure—when it's not available, we lose the ability to handle ALL uploads. If uploads are core to your business, you know how much of a problem that can be. Facebook has a great (although slightly outdated) [article¹¹⁵](#) about how they moved off of NFS to their own system called Haystack. It's great reading material, if only for inspiration.

Don't Upload the file directly to S3 (or any long-term storage)

Most people handle file uploads by directly uploading the users file to Amazon S3 in the context of the web-request. That is to say, they upload the file to long-term storage in the same web-request that the file is originally uploaded in. In embarrassingly simple psuedo-code, it looks something like this:

```
1 <?php
2 if ($_FILES['upload']) {
3     upload_to_amazon_s3($_FILE['upload']);
4 }
```

Although doing it this way is simple, and works for smaller services, there are two major problems.

1. What happens if Amazon S3 is down or running slow? Well, unless there's a retry system in place, we lose the user's uploaded file.
2. It significantly increases the time of the HTTP request, which chews up one of your PHP-FPM processes for a longer period of time. One less PHP-FPM process means one less concurrent connection that your application server can handle. So now the user has to wait not only for the file to upload from their computer to your server, but also from your server to Amazon S3 (although this can be mitigated by cutting the user loose early with `fastcgi_finish_request()` or using a separate PHP-FPM pool for uploads).

Handling user uploads the right way

Since handling user uploads is part of the core-business of Twitpic, I've thought long and hard about how to handle the problem in the best way. Our solution scales horizontally, is robust enough to handle failures while avoiding any single points of failure, and is able to deal with retries. So here's what it looks like:

1. **The file is uploaded by the user and immediately saved to disk** Duh, right. :) The file is uploaded to our application cluster and received by PHP. You should know that nginx buffers file uploads to disk until all of the data is received from the client, before sending it to PHP. This means that PHP's upload progress features will not work, because PHP isn't receiving the uploaded data in realtime.

¹¹⁵https://www.facebook.com/note.php?note_id=76191543919

We create a database record for the file and immediately store the file to disk in a data directory on the application server. We use the `id` column of the database record as the filename. We have a column in our database for the file named `location`. In the `location` column, we store the hostname of the server that the file was originally uploaded on.

Finally, after the file is stored to disk, we create a new job in a Resque queue to eventually push this upload to long-term storage.

The data directory that we store the file in is web accessible. What this means is that if a file is uploaded to `app01`, and gets a database `id` of `123456`, it can be accessed from the web at `http://app01/data/123456`.

This method gives us the ability to display the file immediately, without waiting for it to be on Amazon S3, by linking directly to the application server that handled the upload. Note that this does require a database lookup to determine the file location before linking to it, however. If your PHP code running on `app04` wants to show image `123456`, it needs to lookup `123456` in the database, check the `location` of the file, and then it can use that `location` to generate the link `http://app01/data/123456`.

In the example below, we're going to pretend we're dealing with image uploads, and we'll add a `url()` method to our image model to handle the URL logic.

```

1 <?php
2 if ($_FILE['upload']) {
3     $image = new Image;
4     $image->location = php_uname('n'); // Returns the hostname
5     $image->save();
6     move_uploaded_file($_FILES['upload']['tmp_name'], "/u/apps/data/{$image->id}\
7 ");
8     Resque::enqueue("uploads", "Upload_Job", array("id" => $image->id));
9     echo "<img src='{$image->url()}'>";
10 }
```

And the `url()` method would look something like this

```

1 <?php
2 class Image extends Model {
3     public function url() {
4         if ($this->location == "s3") {
5             return "http://s3.amazonaws.com/my_bucket/{$this->id}";
6         } else {
7             return "http://{$this->location}.mysite.com/data/{$this->id}";
8         }
9     }
10 }
```

2. Push the file to long-term storage In step one we handled the user's upload, stored it temporarily on a single application server, and made it immediately available for access on the web. That's great, but we also need to quickly push it to long-term storage, because if `app01` crashes or becomes unavailable, we'd lose access to that user's content. No bueno!

We queued up a job in Resque to push the file to long-term storage when it was originally uploaded, so let's talk about how we should implement the job worker (Resque is covered in-depth in Chapter 7).

The first thing that the job should do is send the file to Amazon S3. But where are we going to get the file from? We can't directly access the data directory on the filesystem because the Resque job won't be running on the application servers. Of course, you could tie specific jobs to specific servers, and run Resque workers on the application servers, but that's an overall bad design. Jobs should be able to run *anywhere*. Wait! All we have to do is use the `url()` method discussed in the example above, and we can make an HTTP request to download the image—allowing the Resque job to work from *any* location the job is run.

Our Resque job would look something like this:

```

1 <?php
2 class Upload_Job {
3     public function perform() {
4         $image_id = $this->args['id'];
5         $image    = Image::find($image_id);
6
7         $data    = file_get_contents($image->url());
8         $status  = upload_file_to_s3($data, $image->id);
9
10        if ($data == false || $status == false) {
11            throw new Exception("Couldn't upload to S3, retry");
12        }
13
14        $image->location = "s3";
15        $image->save();
16    }
17 }
```

We're even handling failure! When a Resque job throws an exception, it is queued up again and retried. If there is a hiccup and your long-term storage or application server becomes momentarily unavailable or times out, the job is tried again and no data is lost.

In our real-life usage of this setup, the Resque jobs run very fast—almost immediately, so images are moved to long-term storage about 1-second after upload. What this means is that if an application server crashes, we would lose (at most) about 1-second worth of user uploaded content from that server. Not too bad.

3. Cleanup The last thing to think about is cleanup—removing the uploaded data from your application servers after it has made its way to long-term storage. You obviously don't want this old, stale data to grow uncapped because eventually it'll fill up the disks on your app servers.

You could do something complex like using the [nginx HttpDavModule](#)¹¹⁶ to create a private/internal WebDAV endpoint for your data directory, allowing the Resque job to delete the file from the application server over HTTP after successfully storing it in long-term storage.

I prefer the simpler solution though :) – a simple cron to scan the data directory on each application server and delete uploads older than X-days. You could write a script for this, but it's do-able with a UNIX one-liner that will delete the files in /data directory older than 7 days.

```
1 0 4 * * * find /data -type f -ctime +7d -exec rm -f {} \;
```

How I used to handle this the wrong way

Pushing uploads to long-term storage is really a great fit for a queue. The original system we used at Twitpic (amazingly only 3-years ago) was really horrible and unreliable (I'm almost too embarrassed to write it here). We skipped the queue and used the database directly. That is to say, we ran a cronjob that executed a query which looked something like `SELECT * FROM images WHERE location != "s3" LIMIT 100`, and then looped over the result, uploading each file to Amazon S3.

This was a horrible design for a couple of reasons.

1. It didn't scale at all. Running the query multiple times returns the same result (unless you add in an `ORDER BY RAND`, which is terrible for performance), so scaling to multiple servers would mean lots of duplicated images uploaded to Amazon S3, a waste of system resources and expensive AWS bandwidth.
2. We didn't have our `location` column indexed (it really doesn't make sense for it to be for our other queries), so running `SELECT * FROM images WHERE location != "s3"` causes a full-table scan. A full-table scan means that the database needs to look at ALL of your data. If you have 10 million rows, it needs to scan them all. Every. Single. Row. Ugh!

¹¹⁶<http://wiki.nginx.org/HttpDavModule>

Case Study: How DNS took down Twitpic down for 4 hours

Believe it or not, DNS resolution actually caused a significant amount of downtime at Twitpic in 2010.

At the time, we weren't using our own DNS servers and relied on our datacenter's DNS servers to handle DNS resolution. That morning, I pushed a code change to move our sessions from a single memcache server to be split across multiple memcache servers. The configuration line listing the memcache servers looked like this:

```
1 $SESSION_SAVE_PATH="192.0.2.5:11211,192.0.2.6:11211,\n\r192.0.2.7:11211";
```

Notice the newline after the second host. In my text editor, it just wrapped, so I didn't notice it before deploying.

I deployed the code and the site went down. I noticed the extra newline, figured that it caused the issue, reverted the change, and everything was back to normal. Great.

A few minutes later our application processes started hanging. This caused a huge domino effect, which took down the entire site. What broke? I had already reverted the bad change...

It turns out, due to a bug in the memcache library, when it saw the host \n\r192.0.2.7 in \$SESSION_SAVE_PATH, it parsed it as a domain name and not an IP address. In an attempt to resolve the domain \n\r192.0.2.7, it made millions of invalid domain lookups to our hosting provider's DNS resolver.

They really didn't like this— it's a shared service and we were effectively DOSing them with invalid requests, so they promptly blacklisted our servers from their DNS resolver, even after the change was reverted. Being blocked from DNS, with the default system settings, caused a 5-second timeout every time we tried to do a DNS lookup, which we do quite frequently as our code uses several 3rd party APIs, including Twitter, Amazon S3, ZenCoder, etc. Blocking for 5-seconds ate into our available PHP-FPM daemons and eventually brought everything down to a halt.

This was a strange, unexpected, and rather weird problem, so it took several hours to debug and determine the root cause. And that was promptly followed by an hour of frantically setting up our own internal DNS solution. So, recognize that if your application makes any 3rd party API calls, you need to take control of your DNS situation BEFORE it bites you in the ass.

Obviously, using a caching resolver like `nsqd` or `dnsmasq`, coupled with a low timeout in `/etc/resolv.conf` would have helped to avoid the issue entirely.

Case Study: Scaling MySQL Vertically (10 to 3 MySQL Servers w/ SSDs)

If scaling out horizontally is all the rage these days, why would you ever want to scale vertically? Hardware is getting faster and cheaper these days and it's getting cheaper to scale vertically, to a point.

Scaling horizontally is when you scale by adding more servers to your cluster.

Scaling vertically is when you scale by adding faster hardware (CPUs, RAM, etc) to your existing hardware.

There's a fine balance between the two, it's an art. The goal is to maximize the cost/performance ratio, not choose one mantra over the other. If you scale horizontally without doing any vertical scaling, you end up with more complexity. If you scale vertically without doing any horizontal scaling, you end up with very expensive hardware and single points of failure, all of your eggs are in one basket.

It's getting way cheaper to build out huge systems these days, even compared to two years ago. [Basecamp purchased 864GB of Memory in January.¹¹⁷](#) Total cost: \$12,000.

Even Amazon EC2 now offers servers with ridiculously fast SSD drives and 240GB of memory.

I'm not saying you should always scale vertically, but it's worth weighing the costs of "yet another low-powered server" versus making an investment into purchasing better hardware.

At Twitpic, we found ourselves evaluating that very question earlier this year. One of our very read-heavy database clusters had grown over the past years from just a handful of MySQL servers to 10. Each of these 10 servers had a very expensive RAID-10 with a \$1000 Adaptec Hardware RAID Card, several 15,000 RPM SAS Hard Drives, and a meager amount RAM. And, because of the RAID card and drives, these servers were *expensive*.

Anyways, we found ourselves peaking the capacity of all 10 servers. During busy hours, they were running at 100% I/O capacity, slowing down response times and causing unpredictable query speed. I predicted that at least another 5 servers were needed to keep performance at an acceptable, healthy range.

We had gotten bitten by the "scaling horizontally is always better" bug and hadn't even considered the performance benefits or cost of using better hardware— we had always just ordered the same hardware that we were already using for our other MySQL servers.

A different approach

Having to purchase yet another 5 servers, I knew that I'd be back down this road in just a few more months, having to purchase even more servers. It was unsustainable.

¹¹⁷<http://37signals.com/svn/posts/3090-basecamp-nexts-caching-hardware>

I decided, instead, to take a step back and evaluate the hardware decisions we had made and why were we using this specific build.

Scrap the RAID-10

My first thought- we can scrap the RAID-10. It requires double the hard-drive capacity and we don't need such high data integrity on our MySQL slaves. Since we took backups and scaled MySQL slaves with our HAProxy load balancer, we can just rebuild slaves when a drive fails.

In fact, we could get even better of a performance by using a RAID-0, normally unsafe, but since our master server is already using a RAID-10 and we take frequent backups, it's completely safe in our scenario.

More RAM!

Our MySQL server build was configured with a tiny amount (16GB) of memory, a relic from the past when we were many orders of magnitude smaller. The fact we had gotten so far without ever considering to add more RAM was crazy. Nowadays, memory is so cheap that adding 64GB, 128GB, even 512GB to a single server is both cheap and completely normal on regular server hardware.

SSDs here we come

The last thing I evaluated were SSDs. I hadn't expected the difference in performance between an expensive 4 drive RAID-10 setup and a cheap Intel SSD to be so staggering. In fact, further testing and benchmarking showed that a ~\$2000 RAID-10 Setup and ~\$2000 in Enterprise SSDs had about a 30x performance difference. This was in 2011. It's even more staggering and ridiculous today.

The old RAID-10 setup maxed out around 5,000 IOPS while the SSDs dominated by pushing over 150,000 IOPS. Mind = Blown.

The end result

After evaluating our hardware and deciding to scale both vertically *AND* horizontally, we ended up not only saving a significant amount of money on servers, but we also sped up our infrastructure and reduced the overall complexity by having a lower total number of servers.

In fact, we were able to actually scale down from 10 servers to just 3 with the new hardware configuration, reducing our overall infrastructure costs. On top of that, with just those 3 servers, we're only using 50% of our overall capacity. The 10 older servers were struggling to run at 100% of their capacity.

Less servers to monitor, administer, and host equals less complexity in your infrastructure and is a big win for everyone.

Case Study: Hot MySQL Backups w/ Percona

Being able to automatically back up MySQL to an off-site location is an important and often neglected part of building a redundant failure-proof setup. The reason most people neglect taking backups is that once your database grows to a large size, it becomes very difficult to work with due to the massive file sizes. Like, it's easy to back up 1GB of data but becomes much more difficult when we're talking about hundreds or even terabytes worth of data.

Backups are important for two reasons:

1. Allows you to restore your database in case of corruption or accidental loss of data (DELETE FROM foobar, DROP TABLE foobar, etc.)
2. Allows you to spin up a new slave server quickly. Instead of having to remove an existing slave in order to clone it, you can just restore it from a backup and have it catch up to the current replication.

Furthermore, most of the built-in backup solutions for MySQL suck! Let's talk about some of the not-so-good options.

mysqldump

`mysqldump` is a tool that comes with MySQL for backing up the database. Running `mysqldump` produces a text file backup of your database in the form of `CREATE TABLE` and `INSERT` statements. It's the most basic way to take a backup and works well for users with small amounts of data.

The downsides: it's painfully slow, the backups take up much more space than the underlying data, and it's extremely slow to restore from because all of the SQL statements need to be parsed to recreate the underlying data files. `mysqldump` also requires write-lock to create consistent backups. A 100GB backup with `mysqldump` can take several hours, if not days.

mysqlhotcopy

`mysqlhotcopy` is another tool that comes out of the box with MySQL. It can make a hot copy of the raw data files while the server is running. **It doesn't support InnoDB, though, so it's pretty much unusable for most people.**

MySQL Enterprise InnoDB Hot Backup

Oracle provides an InnoDB backup solution that works similarly to `mysqlhotcopy`, but it costs \$5000 per server. Unfortunately, that's too expensive for most startups.

Backup Slave

Some people believe that having MySQL Slaves is “good enough” as a backup solution. Relying on MySQL Replication is a bad idea, though, because although it protects you from data corruption, it doesn’t protect you from rogue queries. For example, if you relied solely on replication as your backup, a `DROP TABLE foobar` would not only destroy your working dataset, but also your backup, since all statements are replicated.

That being said, **it’s a great idea to have a dedicated slave for taking backups**. When you combine having a backup slave with a *correct* backup solution, taking backups won’t impact the I/O performance of your production systems.

Copying the MySQL/InnoDB Data Files

One low-tech solution is to just make a copy of your InnoDB data files inside of your MySQL data directory. It’s able to back up the data as fast as the disk drive can spin and allows for quick restores.

The downsides: You need to shut down your MySQL database or grab a write lock on the database while the copy is being made. Failure to do so will result in a corrupted backup.

Taking it one step further with LVM You can use this technique, which isn’t terribly bad (just risky), by using LVM to create snapshots of your data. LVM adds a copy-on-write layer to the Linux File System, so backups become almost instantaneous. Again, the downsides are that you need to be absolutely sure MySQL is shutdown or that you have a write-lock because failure to do so will cause corrupt backups.

More on LVM Backups here.¹¹⁸ (but don’t bother, the best solution is next)

The BEST (and free) Solution: Percona XtraBackup

Percona XtraBackup¹¹⁹ is a free and open-source hot-backup solution that works with both InnoDB and MyISAM database. You don’t even need to be using Percona Server to use it! Percona XtraBackup is able to make backups of your database while it’s running, without having to lock tables or block writes.

On top of that, it’s extremely fast (runs as fast as your disks), can backup multiple tables in parallel, and is easy to use.

```
1 $ apt-get install xtrabackup
2 $ innobackupex /backup/
3 $ innobackupex --apply-log /backup
4 $ tar -zcf backup.tar.gz -C /backup
```

¹¹⁸<http://www.lullabot.com/articles/mysql-backups-using-lvm-snapshots>

¹¹⁹<http://www.percona.com/software/percona-xtrabackup>

What sort of magic does XtraBackup use? It's actually pretty clever. When you first run `innobackupex`, it makes a copy of the raw InnoDB files. This has the advantage of being extremely fast since it's just a file system copy and is just a sequential write.

But, you're thinking, "Won't the data be corrupted? Don't you need to shutdown MySQL or get a lock?" You're right, *the data will be corrupted*, and that's okay. The magic is in the next command.

The second command, `innobackupex --apply-log` **must be run to uncorrupt the backup**. Basically, what this command does is starts up a second copy of the MySQL server pointing to the backup data. MySQL sees that the InnoDB data files are corrupted and uses the InnoDB write ahead log to fix the data files and put them into a consistent state. Viola! Your backup is done. Lastly (if you want), you can `tar + gzip/1zop` it up to save space (we see around a 100% decrease in space usage when compressing MySQL backups, more on `1zop` later in this case study).

Moving from `mysqldump` to Percona XtraBackup

Before we discovered Percona and their awesome XtraBackup software, we used to use `mysqldump` to back up our database. When I say that it's slow, know that I really mean it. Backing up terabytes of MySQL data with `mysqldump` is downright painful.

Before we had a dedicated slave server for making our backups, we would use one of our production slaves. It got to the point where making backups would take over 24 hours and would severely impact the performance of the slave making the backup. Furthermore, `mysqldump` requires a write-lock to make backups, which would be unacceptable to do on a production system, so our backups were coming out inconsistent and not truly "point-in-time". It was an all-around bad situation.

Taking over 24 hours to perform a backup is nasty and it's at this point I decided we needed to move to something more robust.

Going into our new backup system, I had a couple of goals in mind for a "perfect" solution.

1. It should not impact the speed of the production systems.
2. Backups should be fast, ideally take less than 1 hour, so we can back up multiple times per day.
3. Recent backups should be stored on-site. All backups should be retained off-site.
4. The system should be very simple. Less complexity means it's easier to verify our backup process is working.

Dedicated Backup Server

The first step was to get a dedicated backup server. The hardware doesn't need to be anything special- a large RAID-1 for storing local backups and a fast drive for running the MySQL instance. It doesn't even need much memory, since a write-only server doesn't need a big InnoDB Buffer Pool.

Percona XtraBackup

I ended up using a highly modified version of [this script by Owen Carter](#)¹²⁰, running via a Cron Job, to make the database backups. The backups are consistent, point-in-time, and don't slow down or block replication. Multiple backups can even be taken at the same time.

I highly recommend adding the `slave-info` option to the `USEROPTIONS` variable in the script, which will include the necessary `Master_Log_Pos` and `Master_Log_File` options in the backup so you can use the backups to bootstrap new slave servers.

More XtraBackup configuration options can be found [here](#)¹²¹

Data retention and long-term storage

I decided that I wanted to keep 7 days' worth of backups on our local system and an indefinite amount of backups on Amazon S3 for long-term archival. The 7 days' worth of backups are useful for spinning up new slave servers.

Since I don't want to deal with being able to retain terabytes worth of MySQL backups locally, we use Amazon S3 to retain a subset of our database backups. Using `boto`¹²², the process is pretty easy to script inside of our backup cron. Gotcha here! You need to use the multi-part upload API when sending files to Amazon S3 that are bigger than 5GB.

We store the data compressed and see over 100% space savings when compressing our MySQL backups. You can use `gzip` to gain the most space savings, but it's very slow. Instead, for large backups, I recommend using `lzip`¹²³. `lzip` is a file compressor that runs significantly faster, but is slightly less efficient at compressing. Additionally, `lzip` can compress in parallel, allowing it to run even faster.

Notifications

Lastly, I wanted the backup system to be very visible. Since we use [Campfire](#)¹²⁴ for our team communication, I was able to use the API to send a message in Campfire every time the backup script ran. It's reassuring to wake up every morning and see "MySQL Backup Finished" posted in Campfire.

It's as easy as adding something like this inside of the backup script:

```
1 SIZE=`ls -sh $THISBACKUP.tar.gz | cut -f 1 -d " "`
2 curl -i -u $KEY:X -H 'Content-Type: application/json' -d "{\"message\":{\"body\"
3 \":\"MySQL Backup Finished (size: $SIZE)\"}}" $CAMPFIRE_URL
```

¹²⁰<https://gist.github.com/931358>

¹²¹http://www.percona.com/doc/percona-xtrabackup/innobackupex/innobackupex_option_reference.html

¹²²https://github.com/chapmanb/cloudbiolinux/blob/master/utils/s3.multipart_upload.py

¹²³<http://www.lzop.org/>

¹²⁴<http://campfirenow.com/>

Case Study: Async APIs w/ Kestrel saved my butt

In terms of the scaling websites, background jobs is a pretty new technique. 3 or 4 years ago, Social APIs weren't as common and running parts of the web request in the background was pretty much unheard of at all but the biggest websites.

Twitpic depends on the Twitter API for many site functions. When you upload a new picture, you have the option of posting it to Twitter or Facebook. When you comment on a photo, we push that comment to your Twitter stream too.

In the beginning, it was coded the naive way. Why? We simply didn't know better. We were grassroots, bootstrapped programmers working at scale that was beyond our imagination. You can imagine our code looked something like this-

```
1  <?php
2  class Image_Controller extends Controller {
3
4      // When someone uploads a new image...
5      public function create() {
6
7          $image = new Image();
8          $image->process_image_upload();
9          $image->save();
10
11         // Generate the tweet text to send to Twitter..
12         // i.e, "Check out this cool pic. http://twitpic.com/abc123"
13         $tweet = $image->tweet();
14
15         // Post a tweet to Twitter through the API.
16         Twitter::post_tweet($image->user, $tweet);
17
18     }
19 }
```

Programmatically, this makes sense. But it's the *wrong* way to tackle interacting with 3rd party APIs at any scale. *EVEN* if you're still small. Why? Because even the best case scenario sucks. Since PHP blocks on I/O, API requests are going to take 1-2s to process, no matter what. Everytime a user uploaded an image, we'd tack on 1 or 2 seconds to post the tweet to Twitter.

You can argue it's not a big deal because they're uploading a file- they're not going to notice an extra few seconds. But what about commenting? We want to post a tweet when they make a comment, a normally fast operation, made slow because of the API interaction.

And it's not a dig at Twitter. **ALL** HTTP APIs are slow. It's the nature of the beast, when you go over the network to hit an API across the country, it's going to add latency.

The worst-case scenario

Forget user experience, doing it this way leads to some serious problems. Your website becomes *dependent* on someone else. Remember when Twitter used to crash on a weekly basis? If you were working with the Twitter API like us, Twitter going down would crash your website too! Fail whales for everyone.

Remember how I said that each pesky API call takes 1-2 seconds because PHP blocks and waits for the response? Well, what happens when the API requests take a very long time (5+ seconds) to respond? It sits there waiting for a response, unable to process any new HTTP requests. And when Twitter goes down, these "blocked" PHP processes, sit there.. waiting for a response or to hit their timeout.

On a busy website, this can **QUICKLY** eat up all of your available PHP processes. When you're posting 100+ images per second to Twitter, and Twitter goes down, that means you're effectively blocking 100+ PHP processes every second as they wait for the Twitter API to respond or time out- making them unable to serve any other incoming requests.

It causes a domino effect of sorts and will eventually take down the entire site when you run out of available (non-blocked) PHP processes.

When Twitter went down hard, we went down hard. It was nasty.

The road to kestrel

After being fed up with having our uptime dependent on another companies uptime, I decided enough was enough- and started researching options.

My first inclination was to use a database (*don't do this*). I setup a table, inserted tweets into, and posted them with a process running in the background. As discussed in Chapter 8, this causes a race condition when you run more than one background worker unless you use MySQL Locking. MySQL Locking works, but it creates a global lock in your queue and is a poor solution for something that should be parallel and concurrent. Not to beat a dead horse, but **the database is not a queue**.

So, my first foray into background posting caused duplicate tweets and all around failure. I told you had no idea what I was doing.

I started researching other solutions. I found some articles written about [Starling¹²⁵](#) by Twitter- it was their first queuing daemon, written in Ruby. Unfortunately, Ruby (or PHP) isn't the best fit for concurrent, highly-available queueing system, so there was a ton of bad press surrounding it. It lead me to find their open-source queue, Kestrel, which replaced Starling.

Kestrel is written in Scala and runs on the JVM. Furthermore, it uses the memcached protocol, so no extra client necessary- just point your Memcached client to it! At this point in time, Resque

¹²⁵<https://github.com/starling/starling>

wasn't around so Kestrel was the best (and least complicated) solution. Eventually, we moved all of this over to Resque, though. The concepts still apply regardless of the queue server you use.

Anyways, I setup Kestrel and starting pushing our Twitter API calls into it. Instead of doing them during the web request, I had moved them into the background to be processed "eventually".

Win

Using Kestrel was a huge win and solved many of early downtime problems. Instead of crashing when Twitter went down, our site would run FASTâ€”we'd just see jobs pile up inside of Kestrel queues. And once Twitter came back online, all of those jobs would get processed! No more downtime, no more data loss.

Already it was a huge win. But it also improved our the user experience. Remember how I said posting to Twitter, best case, takes 1 or 2 seconds? That meant 1-2 seconds to post new images or comments. With background workers in place, new comments went through INSTANTLY and images were uploaded 25% faster.

Case Study: The missing guide to Memcached

Memcached is cool. It was one of the original bad-boys in the scaling world and sits at the core-foundation of many of the biggest websites on the internet. Why? It's simple, fast, and convenient. Memcached can be installed easily on almost every platform and is simple to configure.

It's tempting to consider many of the Memcached-alternatives out there. I've seen some "modern-takes" that speak the same protocol but are implemented in Golang or Scala. I would avoid any serious usage of Memcached servers written in a Garbage collected language— you don't want your site's bottleneck to be caused by a stop-the-world garbage collector in your Memcached server.

The original C-based memcached is still a great choice— it's simple, stable, and reliable.

Choosing Memcached over Redis?

So when should you use Memcached over Redis? For **MOST** purposes, I prefer to use Redis. Redis is very fast, has a bunch of convenient data types not found in Memcached like Sets and Lists, and stores data to disk. Redis is extremely convenient for keeping cached data persistent across restarts. For instance, it's common for sites to run very slow after a Memcached restart when the cache is cold and MySQL is getting hammered because of the empty cache. You avoid this problem with Redis entirely because the cache is immediately hot (although, Redis can take a significant amount of time, 60s+, to initially read the stored data into memory on restart).

For most deployments, I recommend only using Redis. It's not worth adding a new moving part to your infrastructure— since Redis can do everything that Memcached can (and more). However, Memcached does bring these two key advantages to the table:

1. It's multi-threaded. Redis, being evented, can only work with a single connection at a time. Even without threads, Redis is DAMN fast, but multi-threaded Memcached means that it can accept and process more connections, and take full advantage of all of the CPU cores available to it (Redis can only use one core).
2. It uses less memory. Redis has to keep several pointers for each item it stores. Memcached allocates memory in slabs, avoids `malloc` entirely, and can keep down the bookkeeping overhead. There are some ways to mitigate this using Hashes in Redis (Hint: it's ugly), but item-for-item, Memcached will use less memory. While barely apparent in small numbers, this can make a difference over hundreds-of-millions of keys and save you GBs of expensive memory.

Installing Memcached

On Ubuntu, installing Memcached is dead simple, and gets you a very recent version.

```
1 $ apt-get install memcached
```

Edit the config file `/etc/memcached.conf` and change the maximum memory (`-m`), maximum number of connections (`-c`), and maximum number of threads (`-t`). I like to use the number of CPU cores as a starting point for number of threads. You'll also want to bump up the open-file limit and tune the network settings via `sysctl` as previously covered in earlier chapters. Remember, the memory limit (`-m`) is a hard cap— when memcached hits this limit, it'll start throwing out old data.

The configuration file, after tuning, should look something like this:

```
1      # Run memcached as a daemon. This command is implied, and is not needed for t\
2 he
3      # daemon to run. See the README.Debian that comes with this package for more
4      # information.
5      -d
6
7      # Log memcached's output to /var/log/memcached
8      logfile /var/log/memcached.log
9
10     # Be verbose
11     # -v
12
13     # Be even more verbose (print client commands as well)
14     # -vv
15
16     # Start with a cap of 64 megs of memory. It's reasonable, and the daemon defa\
17 ult
18     # Note that the daemon will grow to this size, but does not start out holding
19     # this much memory
20     -m 24000
21
22     # Default connection port is 11211
23     -p 11211
24
25     # Run the daemon as root. The start-memcached will default to running as root
26     # if no -u command is present in this config file
27     -u memcached
28
29     # Specify which IP address to listen on. The default is to listen on all
30     # IP addresses. This parameter is one of the only security measures that
```

```

31      # memcached has, so make sure it's listening on a firewalled interface.
32      -l 10.0.1.10
33
34      # Limit the number of simultaneous incoming connections. The daemon default
35      # is 1024
36      -c 4096
37
38      # Lock down all paged memory. Consult with the README and homepage before
39      # you do this
40      # -k
41
42      # Return error when memory is exhausted (rather than removing items)
43      # -M
44
45      # Maximize core file limit
46      # -r

```



What happens when Memcached runs out of memory? Memcached uses a LRU (least-recently used) algorithm to track the number of times a key is requested. When Memcached runs out of space and there are no expired keys to delete, it will start evicting the least-used keys available. You can track the evictions key under the stats to see what your churn rate looks like. Typically, a high eviction count will signify that you need more cache space than you have.

Hooking up PHP to Memcached

If you look in PHP's PECL Library, you'll find two different Memcached extensions available to you. The difference is subtle, but it can make a difference—there's the `memcache`¹²⁶ extension and the `memcached`¹²⁷ extension (notice the **d**).

The `memcache` extension is older, has less features, and uses its own homebrew library for connecting and interacting to Memcached. There are new beta versions that can be installed by specifying the flag `state=beta` when installing, but the latest stable version is several years old. **I recommend avoiding this extension.**

The newer, and better, extension to use is the `memcached` PECL extension. It uses `libmemcached` to connect to Memcached, so it has a well-supported and tested mechanism for connecting to Memcached. In addition, the `memcached` extension offers a host of new features that otherwise is not available in the older `memcache` extension.

To install, just run this command:

¹²⁶<http://pecl.php.net/package/memcache>

¹²⁷<http://pecl.php.net/package/memcached>

```
1 $ pecl install memcached
```

Protocols

Clients connecting to Memcached server can talk to it in two different ways— via an ASCII protocol and a Binary protocol. The default method is ASCII, which is convenient for debugging (because you can easily read the communication back-and-forth), but it's completely unnecessary for production and just increases network chatter. Change it to Binary Protocol—

```
1 $m = new Memcached;
2 $m->setOption(Memcached::OPT_BINARY_PROTOCOL, true);
```

By the way, in case you were curious the ASCII protocol is dead simple and I'll show you how you can use it to debug Memcached from telnet at the end of this chapter.

As an example, the ASCII protocol looks like this:

```
1 $ telnet localhost 11211
2 > GET hello      # Retrieve the key named 'hello'
3 < VALUE world 0 5 # Memcached server returns the value 'world'
```

Compression and Serialization

A common use-case for Memcached is to use it to store more complicated values, like PHP arrays or objects. Typically, if possible, you want to avoid this altogether. Serialized objects take up quite a bit of space, and the bigger objects, the more memory you'll chew through. If you find yourself needing to store complex objects or arrays often, I suggest considering Redis as an alternative, as it has more complex data types better designed for this type of work.

That being said, if you absolutely must store arrays and objects in Memcached, it's easy. Just pass them into the `set` function. The default behavior of the Memcached client is to use PHP's built in `serialize` function on the object before sending it. This allows you to take advantage of custom `__serialize()` functionality provided by PHP's magic object methods. You can add a method named `__serialize()` to any class and add your own custom logic to properly serialize nested objects or only return a limited subset of values.

Years ago, I worked with a guy who insisted on comma-separating arrays with `explode` before storing them in Memcached. He would join the arrays back together when retrieving the data. Don't be that guy. PHP does the serialization for. If you find yourself often needing a more sophisticated data type— you need Redis.

PHP's built-in serialization can be pretty heavy. Since it's an ASCII based serialization, it takes up an non-optimal amount of bytes. There is a drop in serialization replacement extension called `igbinary`¹²⁸ that allows you to use a more-complex but smaller footprint form of binary serialization. It "just works" seamlessly once you install the module. If you're going to be storing a bunch of serialized data in Memcached, `igbinary` is a must have.

¹²⁸<http://pecl.php.net/package/igbinary>

```
1 $ pecl install igbinary
```

And then enable it for the Memcached extension

```
1 <?php
2 $m = new Memcached;
3 $m->setOption(Memcached::OPT_SERIALIZER, Memcached::SERIALIZER_IGBINARY);
```



One downside of using serialization and compression is that generally you won't be able to read the values from Memcached using any language besides PHP. If building a Service-Oriented-Architecture, with multiple backend languages, this can be a deal breaker. You may want to look at `Memcached::SERIALIZER_JSON` for this type of work.

Similarly to Serialization, PHP's Memcached extension can also compress the values of your items using standard gzip. This is convenient because it allows you to use less memory when storing bigger objects. Enable it with the following code:

```
1 $m = new Memcached;
2 $m->setOption(Memcached::OPT_COMPRESSION, true);
```

With regards to compression, it will only be enabled on larger values (small values will skip compression), and benchmarks generally show an all-around performance improvement when compression is enabled.

Server Pools

It's often desirable to spread your Memcached usage over several servers, both for increased memory access and to avoid a single point of failure.



Note: Out of the box Memcached doesn't offer anything based around replication. As you know, data in Memcached is considered ephemeral, so loss is considered "okay". There are some alternative Memcached servers and plugins that offer a replication option.

We use the concept of Server Pools to enable multiple Memcached Servers. In code, it looks like this:

```
1 <?php
2 $m = new Memcached;
3 $m->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE, true);
4 $m->addServers(
5     array(
6         array('192.168.0.1'),
7         array('192.168.0.2'),
8         array('192.168.0.3')
9     )
10 );
```

Remember, that the data is distributed over the various servers— it's NOT replicated. We are sharding the data over several servers and no single server holds all of the data. The amount of memory available is addative— if all three servers in the pool have 32GB of memory available to them, you effectively have 96GB of usable cache space.

Consistent Hashing

If you're wondering how Memcached knows which server to get which keys from, since any of the three servers in the pool can only hold 1/3 of the total keys, GOOD QUESTION!

If you're also wondering why we set `Memcached::OPT_LIBKETAMA_COMPATIBLE` in the previous example, another good question!

The Memcached client is responsible for handling all routing. That is to say, if `KEY1` is on `192.168.0.1`, and you ask `192.168.0.2` for `KEY1`, it will tell you the value doesn't exist. The servers don't talk between themselves and have no idea what keys exist besides the particular set they have stored locally. There is no cross-communication between Memcached servers in a pool.

When you ask the PHP Memcached client for `KEY1`, it has to figure out which server to connect to, otherwise it won't get the right answer back (and it would be really bad for scaling if it had to check all of them, imagine if you had 50!). Since keys can be stored on any server in the Memcached pool, we need a way to figure out which server "owns" a particular key. It does this by hashing the key— that is, running the key through a function that will always return a consistent output. In the case of Memcache, the input is the key string and the output is the server in the pool.

Here is a very simplistic memcached key hashing function using `strlen` and modulo math.

```

1 <?php
2 $servers = array("192.168.0.1", "192.168.0.2", "192.168.0.3");
3
4 function memcache_hash($key) {
5     $key_length = strlen($key);
6     $no_of_servers = count($servers);
7
8     return $servers[$key_length % $no_of_servers];
9 }

```

Obviously, this is a poor hashing algorithm because most of our keys are going to be short and won't distribute over Memcached evenly, but you get the idea— with the same key, we will always output the same server. Thus, we can determine which server “owns” a key when setting and retrieving it without having to check all of the servers.

Great, we've got hashing down, but what the heck is consistent hashing?

Consistent hashing is the same concept, except with a slightly different twist. In the example above, if the number of servers change (we add or remove one), the \$no_of_servers count changes, which impacts the end hash that's produced. That is to say, when you change the count of \$servers, all keys will generate a new output, effectively wiping your cache pool clean. Even though the data is still there, the hashing algorithm will produce a different output for previously stored keys, so it won't map the keys correctly, and PHP won't be able to find previously stored data.

Consistent hashing fixes this. Using different techniques (pre-defined bucketing, token ring, etc), when the number of servers changes, only a small percentage of keys are remapped, allowing us to add and remove servers without effectively wiping our entire Memcached cluster.

Understanding the underlying algorithm isn't total important. What is important is that you set the following option when using multiple Memcached servers in order to enable consistent hashing. By default, PHP will use Modulo-based key distribution, which will remap all keys when adding or removing a server.

```
1 $m->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE, true);
```

Enabling this does a few things— it turns on Consistent Hashing and sets the key hash algorithm to use MD5. Libketama is a general hashing algorithm for Memcache that is available in most languages, so using Libketama means your hashing system will be easy accessible from almost every modern programming language.

Adding Weight

One option that the Memcached client provides us with when adding servers the choice to specify *weight*. Weighting allows you to influence the proportion of keys that are mapped to a particular server. This is useful for circumstances when your Memcached servers have different amounts of resources, particular memory, available to them.

```
1 <?php
2 $m = new Memcached;
3 $m->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE, true);
4 $m->addServers(
5     array(
6         array('192.168.0.1', 11211, 25),
7         array('192.168.0.2', 11211, 25),
8         array('192.168.0.3', 11211, 50)
9     )
10 );
```

In the above example, the last server, 192.168.0.3, will get twice as many keys to store as the other two servers, because it has double the weight. This would be ideal, if, for example, 192.168.0.3 had double the memory of the other two boxes.

Persistent Connections

Instead of opening a new memcached connection on each PHP request, we can use persistent connections and re-use the same connection for multiple requests. Each PHP-FPM worker gets its own persistent connection, so if you have 50 workers, you will have 50 static connection to memcached.

Persistent connections are generally preferred as they are slightly faster because of reduced per-request connect/disconnect overhead and also keep the number of connections to memcached static. They can be especially helpful when you have several memcached servers in your pool that you have to connect to.

The way persistent connections are defined with the memcached extension is slightly confusing, so let me explain how it works.

First, you need to define a persistent connection pool name, by passing a string into the constructor, like so:

```
1 <?php
2 $m = new Memcached("my_pool");
```

Just like without persistent connections, the next step is to add the servers to the pool, but with one slight change. When we pass a pool name to the `Memcached` constructor, we're asking it grab the existing persistent connection pool named `my_pool` if it already exists, otherwise give us back an empty pool. This is a different PHP paradigm than we're used to be, because the pool lives across multiple requests.

So, before calling `addServers`, we need to check if the `Memcached` constructor has returned an already configured persistent connection pool or an empty connection pool. If it's already configured, we don't need to do anything! If it's empty, we need to call `addServers` and any `setOption`'s that we may have.

The code looks like this:

```

1 <?php
2 $m = new Memcached("my_pool");
3
4 if (empty($m->getServerList())) {
5   $m->addServers(array(
6     array("192.168.0.1"),
7     array("192.168.0.2")
8   ));
9 }

```



It's extremely important to do this check when using persistent connections. If you just call `addServers` on every request, it will re-add the same servers to the connection pool over and over. It doesn't do any duplication checking, so you can potentially add thousands of the same server to a single connection pool.

Atomic Counters

One great use case of Memcache is for counting things— page visits, image views, active users, etc. This is for two reasons. The first is that if you're currently using MySQL to count something that is update frequently, you can save a huge amount of resources by moving that counter out of the database and into a cache. Remember, everytime you write to your MySQL database, you're destroying the entire query cache. If you happen to still have the query cache enabled and don't see many writes otherwise, moving frequently-updated counter columns out of MySQL can make the query cache useful. Likewise, less writes to MySQL just means less load. The second reason why counting things is a great use for Memcached is also because it offers atomic counters, that is— lock-free (in your code, anyways).

You may be tempted to write your counter using Memcache like so:

```

1 <?php
2 $memcached = new Memcached();
3 $count = $memcached->get("counter:{$page->id}");
4 $count += 1
5 $memcached->set("counter:{$page->id}", $count);

```

This is wrong because it creates a race condition. If this code is run in two separate requests at the same exact time, it's possible that one of the requests will overwrite the others increment. You can fix it by using CAS tokens (explained later), or, more easily, by using the atomic counters in Memcached. Consider this:

```
1 <?php
2 $memcached = new Memcached;
3 $count = $memcached->incr("counter:{$page->id}");
```

Likewise, Memcached also implements a command for decrementing

```
1 <?php
2 $memcached = new Memcached;
3 $count = $memcached->decr("counter:{$page->id}");
```

And you can increment or decrement by multiple steps by passing in a second argument with a numeric value.

```
1 <?php
2 $memcached->decr("counter:{$page->id}", 4);
```

Also worth noting— incr and decr return the newest value of the counter, so it reduces a roundtrip from the badly-designed GET/SET pattern described earlier.

Avoiding the Rat Race: CAS Tokens

CAS Tokens, also known as Check-And-Set Tokens, allow you to use Memcached for more-advanced use cases while avoiding potential race conditions. To use them, you must have a *modern* version of Memcache (1.4 or above). They work like this:

1. Before making any changes, call `getCas` to generate a CAS token for the key you plan on changing.
2. Calculate the new value you'd like to set for the key.
3. Send the new value of the key, along with the CAS token, to Memcached. Think of the CAS token as a timestamp of when the key value was last modified. Memcached compares the CAS token you sent with its own CAS token for that key. If they match, it means there have been no changes and it's safe to change the value. If they do not match, it means there has been a change since you got your CAS token, and the value is not safe to change.
4. If the operation returns false (CAS tokens did not match), we try the same operation again.

For example, we could implement a simple counter like so (although, in practice, please use increment and decrement).

```

1 <?php
2 $m = new Memcached;
3
4 $tries = 0;
5 while ($tries < 10) {
6     $token = $m->getCas("counter:1");
7     $value = $m->get("counter:1");
8
9     $value += 1;
10
11    if ($m->setWithCas("counter:1", $value, $token)) {
12        break;
13    }
14
15    $tries++;
16 }

```

Two things to notice. The first is that we have to wrap our algorithm in a while loop, because it's possible it can fail multiple times in a row if it's a key that sees many updates. This is why it's better to use atomic operations if possible. The second is to notice that we add a number of limits to the tries. This is to prevent the loop from just spinning and taking too much time. A preferred alternative, is to use a time-based deadline rather than a simple counter.

A deadline based CAS algorithm is implemented below, with a 20ms deadline.

```

1 <?php
2 $m = new Memcached;
3
4 $begin = microtime();
5 $deadline = 20;
6 while ($begin - microtime() < $deadline) {
7     $token = $m->getCas("counter:1");
8     $value = $m->get("counter:1");
9
10    $value += 1;
11
12    if ($m->setWithCas("counter:1", $value, $token)) {
13        break;
14    }
15 }

```

For convenience, it's generally a better idea to wrap this sort of functionality into a helper instead of reinventing the wheel every time you need to write this code.

```

1 <?php
2 function cas_deadline($deadline, $block) {
3
4     $begin = microtime();
5
6     while ($begin - microtime() < $deadline) {
7         if ($block()) {
8             break;
9         }
10    }
11 }
12
13 // Used as so
14
15 $m = new Memcached;
16 cas_deadline(20, function() use ($m) {
17     $token = $m->getCas("counter:1");
18     $value = $m->get("counter:1");
19
20     $value += 1;
21
22     return $m->setWithCas("counter:1", $value, $token);
23 });

```

Watch out for timeouts

My favorite topic of discussion! Timeouts! Luckily, the `memcached` PECL extension has very fine-grain support for timeouts down to the millisecond level. Perfect. In older versions of the `libmemcached` library, there were some bugs around timeouts not working properly, but they are fixed now and we can use timeouts reliably with high precision (just make sure you have a recent version of `libmemcached` installed on your box).

```

1 <?php
2 $m = new Memcached;
3 $m->setOption(Memcached::OPT_CONNECT_TIMEOUT, 50);
4 $m->setOption(Memcached::OPT_RETRY_TIMEOUT, 50);
5 $m->setOption(Memcached::OPT_SEND_TIMEOUT, 50);
6 $m->setOption(Memcached::OPT_RECV_TIMEOUT, 50);
7 $m->setOption(Memcached::OPT_POLL_TIMEOUT, 50);

```

The default for each of these is either 1000 (1s) or 0 (forever), which is bad news on a production server. Depending on your setup, you may want to change my setting of 50ms, but I find that Memcached typically responds within milliseconds across local ethernet, so I keep it at 50ms to allow some breathing room without impacting the speed of the site. Remember, if PHP-FPM starts blocking on Memcached connections you will run out of PHP-FPM workers and your app will go down.

Using Memcached for Sessions

The Memcached extension has a session handler which allows you to use PHP's built-in session handling, except it stores the sessions in Memcached instead of on the filesystem. This is necessary for scaling your app servers horizontally. Some bad guides on the internet will tell you to use NFS or MySQL. Don't do that. Using Memcached for your sessions is as easy adding some php ini values.

```

1 <?php
2 ini_set('session.save_handler', 'memcached');
3 ini_set('session.save_path', '192.168.0.1:11211:50,192.168.0.2:11211:50');
4
5 ini_set('memcached.sess_consistent_hash', 1);
6 ini_set('memcached.sess_binary', 1);
7 ini_set('memcached.sess_prefix', 'app_session.');
8 ini_set('memcached.sess_remove_failed', 1);
9 ini_set('memcached.sess_number_of_replicas', 1);
10 ini_set('memcached.sess_connect_timeout', 50);

```

I like to use `ini_set` for these types of per-app values instead of defining them in `php.ini`, but it's certainly an option to put them in `php.ini` if you'd like.

Notice that `session.save_path` can include multiple servers, port number, and weight. This gives you the ability to spread your session storage across multiple servers for failure and scalability.

The rest of the values are pretty self explanatory but I'll run through them quickly.

- `memcached.sess_consistent_hash`: Turn on consistent hashing using libketama. A must with multiple memcache servers.
- `memcached.sess_binary`: Use the binary protocol for session connections.
- `memcached.sess_prefix`: Set a unique prefix for these session keys. Necessary if you have multiple PHP apps sharing your Memcached pool, without it you may have session collisions.
- `memcached.sess_remove_failed`: If one of your Memcached servers becomes unavailable, remove it from the pool automatically.
- `memcached.sess_number_of_replicas`: One cool features that the PECL Memcached extension has is the ability to replicate the session data to multiple memcached servers, so if one dies, you don't lose the data. Set it to the number of replicas (extra copies) of that you want.
- `memcached.sess_connect_timeout`: The connection timeout before the server is marked bad and skipped. Default is 1000 (1s), which is too high. On local ethernet, the connection time should be not be more than a couple of milliseconds. Set it to 50ms to give yourself some breathing room, but still quick enough to not slow down your app if a server goes down.

Anyways, after you set your session handler to `memcached`, you can use PHP sessions like normal—

```
1 <?php
2 session_start();
3 $_SESSION["foo"] = "bar";
```

Debugging Memcached from Telnet

Even if you're using the binary protocol to access Memcached from PHP, that doesn't mean you can't use the ASCII protocol for testing! It makes it very easy to test Memcached from Telnet.

```
1 telnet 127.0.0.1 11211
2 >
```

From here, you can run any commands, like GET, SET, STATS, and FLUSH.



If you don't know what flush does, be careful! It flushes the entire cache from memory, without warning or confirmation. You've been warned.

The most useful debugging command is STATS. It will give you a bunch of useful information such as connections used, number of items, hit/miss ratios, etc.

```
1 telnet 127.0.0.1 11211
2 > STATS
3 STAT pid 4061
4 STAT uptime 9
5 STAT time 1390339143
6 STAT version 1.4.14
7 STAT pointer_size 64
8 STAT rusage_user 0.000000
9 STAT rusage_system 0.004994
10 STAT curr_connections 5
11 STAT total_connections 6
12 STAT connection_structures 6
13 STAT reserved_fds 20
14 ....
15 STAT threads 4
16 STAT conn_yields 0
17 STAT hash_power_level 16
18 STAT hash_bytes 524288
19 STAT hash_is_expanding 0
20 STAT expired_unfetched 0
21 STAT evicted_unfetched 0
22 STAT bytes 0
23 STAT curr_items 0
24 STAT total_items 0
```

```
25 STAT evictions 0
26 STAT reclaimed 0
27 END
```

You can also get and set values pretty easily from the command line for debugging purposes.

```
1 telnet 127.0.0.1 11211
2 > set foo 0 100 4
3 > data
4 < STORED
```

Here's a quick ASCII protocol primer. `set` is the command, `foo` is the key name, `0` is an arbitrary unsigned 16-bit integer flag that Memcache lets us set (it's how PHP marks a key as compressed, so it can uncompress the value later, for example), and `4` is the number of bytes of the value for this key. Lastly, we have `data` which is the value we want to store for the key `foo`.

Retrieving an item is just as easy.

```
1 telnet 127.0.0.1 11211
2 > get foo
3 < VALUE foo 0 4
```

Similar to the `set` command, we just pass the key name to `get` and it returns the value of the key, the flag, and the byte size of the data. Easy.

My Memcached Setup

What would a case study be without some real-world knowledge? This is how we have our Memcached server setup in it's current incarnation.

Hardware

- Dual Xeon 2620 (12x 2.0GHz)
- 128GB Memory
- 1Gbps Ethernet

We have a handful of these servers— lots of cores because Memcached is multi-threaded and lots of memory because we have an incredible amount of data to cache.

Why run Memcached on its own servers?

One common misconception is that you can simply run a Memcached daemon anywhere that you have free memory. Let me tell you why this is not always a great idea—I used to do this. I ran memcached on all of our app servers, and after much trouble switched to dedicated Memcached boxes.

My motivation was to take advantage of all available resources. I noticed that the app servers had quite a bit of free memory, and across 25 or so boxes, it amounted to an extra 250GB that could be used for memcached. Score!

Except it caused problems. And the worst kind of problems, they were non-obvious and took almost a year to track down.

Let's say there are 5 servers—192.168.0.11 through 192.168.0.15. These servers are running both Memcache and PHP-FPM, serving web requests. We're sharding our memcache load across all 5 boxes, like so:

```

1 <?php
2 $m = new Memcached;
3 $m->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE, true);
4 $m->setOption(Memcached::OPT_CONNECT_TIMEOUT, 100);
5
6 $m->addServers(
7     array(
8         array('192.168.0.11', 11211, 20),
9         array('192.168.0.12', 11211, 20),
10        array('192.168.0.13', 11211, 20),
11        array('192.168.0.14', 11211, 20),
12        array('192.168.0.15', 11211, 20),
13    )
14 );

```

All is great, timeouts are set, sharding is working, and we have a ton of extra cache available to us now.

Here's the problem: load can be highly irregular on any of the individual app servers. Load spikes on 192.168.0.11 because an increase in web requests and the box slows down. Normally, this would be no problem—haproxy would simply fail it out of the pool gracefully and life would continue on.

But that doesn't work as cleanly as we'd expect with this setup. Even though 192.168.0.11 is out of web pool, memcached requests are still going to it, and they're taking longer to respond because of the high load on that box. Connections start timing out, consuming more time in PHP-FPM workers across the cluster, and “cause the infection to spread”.

Now with more PHP-FPM workers across the entire cluster slowing down because 192.168.0.11 is taking longer to respond, it's causing the load to increase across the entire cluster. It gets worse. As the load increases across the cluster, 192.168.0.12 also starts to respond a little bit slower to

memcache requests. And so on and so forth—the end result is that increased load on one server perpetuates to the entire cluster like a domino falling and they all go down.

This is because we've broken a principal rule of scaling! We want things to be horizontally scaled but when we add multiple roles to a single server, and introduce cross-communication (for instance, a single app server ends up talking to all app servers), we're no longer horizontally scaled.

Recommended Options

Here are all of the PHP Memcached Client options that I recommend using for the best performance and future scaling options (ability to add and remove servers, for instance).

```

1  <?php
2  // Use persistent connections
3  $m = new Memcached("pool");
4
5  if (empty($m->getServerList())) {
6      // Set timeouts to 50ms
7      $m->setOption(Memcached::OPT_CONNECT_TIMEOUT, 50);
8      $m->setOption(Memcached::OPT_RETRY_TIMEOUT, 50);
9      $m->setOption(Memcached::OPT_SEND_TIMEOUT, 50);
10     $m->setOption(Memcached::OPT_RECV_TIMEOUT, 50);
11     $m->setOption(Memcached::OPT_POLL_TIMEOUT, 50);
12
13     // Turn on consistent hashing
14     $m->setOption(Memcached::OPT_LIBKETAMA_COMPATIBLE, true);
15
16     // Enable compression
17     $m->setOption(Memcached::OPT_COMPRESSION, true);
18
19     // Enable Binary Protocol and igbinary serializer
20     $m->setOption(Memcached::OPT_BINARY_PROTOCOL, true);
21     $m->setOption(Memcached::OPT_SERIALIZER, Memcached::SERIALIZER_IGBINARY);
22
23     $m->addServers(array(
24         array("192.168.0.1"),
25         array("192.168.0.2")
26     ));
27 }
```

Further Reading and Tools

twmemproxy

An open-source tool by Twitter, twmemproxy acts as a proxy server for memcached. Can shard data automatically, handle persistent connections, provide stats, and lower the connection count

to the backend memcached daemon when you have many, many web workers.

[Link to GitHub¹²⁹](#)

peep

A tool for dumping the memory contents of a live memcached server to peek at what values it is storing. Great if you want to know exactly how memcached is being used.

[Link to GitHub¹³⁰](#)

PHP igbinary / compression / memcached benchmarks

Benchmarks showing different combinations of serialization, compression and memcached extensions (spoiler: igbinary + binary protocol + compression + memcached extension wins).

[Link to GitHub¹³¹](#)

twemperf

Another open-source tool by twitter that can be used to benchmark memcached server performance.

[Link to GitHub¹³²](#)

¹²⁹<https://github.com/twitter/twemproxy>

¹³⁰<https://github.com/evan/peep>

¹³¹<https://github.com/carlosbuenosvinos/memcache-memcached-benchmarks>

¹³²<https://github.com/twitter/twemperf>

Case Study: HTTP Caching and the Nginx Fastcgi Cache

This entire book has been, for the most part, dedicated to scaling our backend services—making our frontend faster by improving server performance, database caching, and getting the result back to the client as fast as possible. Generally this is a good strategy, but we've been skipping talking about an entirely different side of caching, HTTP Caching. What if we could have the client cache our output, in the browser or api client, and skip the trip to the server all-together? That would be most excellent.

This is exactly what HTTP Caching does— it provides a mechanism for us to define rules on when the client should check back for a new version in the future.



In the examples below, > designates the start of a **client request**, while < represents the start of a **server response**.

In the most simplistic version, HTTP Caching works by sending back an `Expires` header that tells the browser that this response will be valid until the timestamp specified.

```
1 > GET /some_page.html
2
3 < HTTP/1.1 200 OK
4 Expires: Thu, 1 Jun 2015 01:00:00 GMT
5 Cache-Control: max-age=29557199
```



Wait, what is `Cache-Control`? It's a newer header. Use both `Expires` and `Cache-Control` with `max-age`. They both essentially do the same thing and you'll be covered for every browser. `Cache-Control: max-age` will overwrite `Expires` in newer browsers. Think of the two synonymously.

This works great for static assets (images, stylesheets, etc) that we know we have a long lifetime. Sometimes, we don't always know when content will expire—an example being the front page of a news website. The homepage is updated several times per day when a new story is posted, but we can't predict the news and don't know when that will be. Guesstimating a timestamp for the `expires` header means that people with the homepage cached in the browser could miss breaking news. We need something more robust.

To do more complex-type caching, we use the `ETag` header—it works like this:

1. On the first GET request, the server generates a unique string value for the `ETag`. In our news homepage example, it can be something as simple as `md5($latest_article->created_at)`.

2. On subsequent requests, the browser will first make a HEAD request, with the HTTP Header If-None-Match: {\$ETag}. On the server, you can take this value, compare it to md5(\$latest_article->created_at), and if they match just return HTTP status 304 Not Modified with no body to the client. This will instruct the browser to use the page that they've cached, and skip having to fully generate the page (you would return a 200 OK with an empty body to instruct the browser to refetch the page).

An example showing ETags in practice is below.

```

1  # Make the first/uncached HTTP request
2  > GET /some_page.html
3
4  < HTTP/1.1 200 OK
5    ETag: "098f6bcd4621d373cade4e832627b4f6"
6    Cache-Control: public, max-age=86400
7
8  # Sometime in the future we go to the page again, instead
9  # the browser checks for a newer version than the one in its cache
10 > HEAD /some_page.html
11   If-None-Match: "098f6bcd4621d373cade4e832627b4f6"
12
13 < HTTP/1.1 304 Not Modified

```

Simple enough. By the way, notice the Cache-Control header. This is useful to add more caching instructions—the public part is useful later when we talk about proxy cache servers, but essentially it tells everyone in between the server and the client whether the content is public (cacheable) or private (uncacheable). For example, you may not want a page with private or personalized content (think /timeline on twitter or /account on facebook) to be cached the same for everyone. The max-age part tells the browser to invalidate the cache and make a full request after a specific amount of seconds.

Caching Static Assets

For simple files (images, stylesheets, webfonts), we can have nginx handle caching headers for us automatically—just need to a location block to our nginx server configuration.

```

1  location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {
2    expires max;
3  }

```

Easy. But what about when you make a change to these assets? They are cached in the browser and outside of our control, how do tell the browser to download the newest version?

Assets in the browser are cached based on their full URL—including the query string! All you have to do is append a “cache buster” to the query string of the asset to force the browser to do

a cold pull. Incrementing a version on deploy is easy (some frameworks, like Rails, even do this for you).

The simplest solution is something like `<link rel="stylesheet" type="text/css" href="main.css?v=5">`, where `?v=5` gets updated everytime you make a change to your stylesheet and want to force the browsers to update their cached version immediately.

Ok, we've got static asset caching down, but how do we handle the magic in our PHP controllers? How do we handle caching for our code paths that can be cached? Here's an example of HTTP Caching in some framework psuedo-code.

```

1  <?php
2  class UsersController {
3      public function show($id) {
4
5          $user = $memcache->get("user:{$id}");
6
7          $type = $_SERVER["HTTP_METHOD"];
8
9          if ($type == "HEAD") {
10
11              $theirs = trim($_SERVER["HTTP_IF_NONE_MATCH"], '""');
12              $mine   = md5($user["updated_at"]);
13
14              if ($theirs == $mine) {
15                  return header("Status: 304 Not Modified");
16              } else {
17                  return header("Status: 200 Ok");
18              }
19          }
20
21          // ... Do normal action stuff
22          $user->expensiveDatabaseCall();
23          $view->render("users/show");
24      }
25  }

```

Notice how we are able to quickly compare the Etag with some data from memcached and potentially skip our expensive database calls? Also worth pointing out that you should wrap `$_SERVER['HTTP_IF_NONE_MATCH']` in a call to `trim()` to remove double quotes from the ETag value—some browsers send them, some do not.

We can also set the `expires` header here, just as easily, with `header("Expires: $timestamp")` if you know that the action will not expire for a long time (maybe a contact page or terms of service).



A quick note about Nginx and ETags—there is currently a bug with Nginx where if you have `gzip` on set, Nginx will eat your ETags and never send them to the client. Annoying!

Caching Proxy

Ok, now that we've got client caching down, in practice we'll find that it's incredibly frustrating to not be in control of the end client. Browsers don't always obey caching rules, each implements the rules slightly different, and they don't always do what you expect them to. While caching is fantastic and can greatly reduce the load on yours server, it's annoying to leave your server load up to the mercy of Internet Explorer.

Enter the caching proxy— our own middle-man that acts as a middle-layer HTTP page cache between your backend php-fpm workers and the browser. It's even built into Nginx! Basically, we'll Nginx looks out for cache headers being set from PHP, and when it sees them, it stores a static copy of that data for the future. When a future web request is made, if it has a cached version it will skip PHP entirely and send the static copy to the client. All while respecting `Expires`, and `Cache-Control`. Wahla! For the right use cases, it's like magic.

I really like using Nginx for this sort of job— it's already part of our stack on our app servers, we know how to configure it, and it's fast as hell. You might hear people talk about Varnish, which is also decent, albeit much more complex to configure for a tiny bit of a extra functionality while being equally as fast. Stick with Nginx unless you have a specific need that Varnish can solve better.

Which pages can I use my caching proxy for?

A caching proxy really works best for public, global pages. Remember, the cache is shared for all users, without any custom logic. Think, user profiles, unique generates content, etc. Anything with a public, globally accessible URL. This limitation can be tricky for some scenarios that I've outlined below.

Logged in / logged out navigation bars

It's pretty common to have a navigation bar on every page that changes depending on if the user is logged in or not. If we cache the logged out version of the page, for instance, a logged in user that gets the same page from the proxy cache (instead of from PHP-FPM), will see the navigation bar in the logged out state. This is actually easily solvable— toggle the state in javascript by looking at some type of session cookie instead of doing it in the PHP view.

Authenticated views (private messages, etc)

Another concern is authenticated views with unique URLs. An example might be private messages— `/messages/123456` is unique but we only want authenticated users (like, sender and receiver) to be able to view the page. Unfortunately, we have to skip the cache for this scenario. We'll have to send it from PHP with the header `Cache-Control: private; no-cache; no-store`, which tells the caching proxy to skip caching the page.

Highly Dynamic Content

One more common scenario is handling content that's highly dynamic, like a view counter that is increased on every page load. We don't want something small like this hindering us from seeing the huge benefits of HTTP caching. The answer? Simple—move this type of dynamic content to AJAX and load it from the browser with a separate request.

Configuring Nginx to be a Cache for PHP-FPM

We can easily enable HTTP caching on our app server by using the built-in nginx fastcgi cache. It's pretty simple to do, just add the following to your nginx server configuration.

```
1  # Set the path where the cache is stored; Set the zone name (my_app),
2  # total size (100m), and max lifetime (60m)
3  fastcgi_cache_path /tmp/cache levels=1:2 keys_zone=my_app:100m inactive=60m;
4
5  # Set the cache key used, in this case: httpsGETtest.com/somepage.html
6  fastcgi_cache_key "$scheme$request_method$host$request_uri";
7
8  server {
9      listen 80;
10     root /u/apps/my_app;
11
12     location ~ \.php$ {
13         fastcgi_pass http://127.0.0.1:8080;
14
15         # Use the zone name we defined on line 1
16         fastcgi_cache my_app;
17
18         # Only cache HTTP 200 responses (no error pages)
19         fastcgi_cache_valid 200 60m;
20
21         # Only enable for GET and HEAD requests (no POSTs)
22         fastcgi_cache_methods GET HEAD;
23
24         # Bypass the cache if the request contains HTTP Authorization
25         fastcgi_cache_bypass $http_authorization;
26
27         # Bypass the cache if the response contains HTTP Authorization
28         fastcgi_no_cache $http_authorization;
29
30         # Add a debugging header to show if the page came from cache
31         add_header X-Fastcgi-Cache $upstream_cache_status;
32     }
33 }
```

This configuration sets up a fastcgi cache in front of our PHP-FPM server. We define the location of the cached files with `fastcgi_cache_path`, setting a 100MB cache limit, with an automatic 60 minute purge for inactive items.

The cache key is defined with `fastcgi_cache_key`— this can be changed depending on your use-case. For instance, notice how `$scheme` is included in the cache key? The `$scheme` variable holds the HTTP scheme (i.e, http or https), which by default will cache the pages differently depending on if they are accessed over SSL. This may not be what you want. An example cache key would look something like `httpsGETmy_app.com/index.php?foo=bar`.

Next up, we define what types of HTTP responses are valid with `fastcgi_cache_valid`— just 200 responses. We don't want to cache error pages. Likewise, with `fastcgi_cache_methods`, we limit caching to only GET and HEAD requests. Generally, it doesn't make sense to cache the response of a POST, which is more often than not going to have dynamically generated content.

Lastly, we put in some definitions for `fastcgi_cache_bypass` and `fastcgi_no_cache`. These two settings allow you to either bypass checking the cache or skip storing something in the cache, depending on the presence of an arbitrary cookie, header, or nginx variable. In this case, we are skipping the cache if the HTTP Header `Authorization` is present.

Remember, though— nginx will respect and obey your `Expires` and `Cache-Control` headers. If a page is sent from PHP-FPM with the `Cache-Control: private; no-store` header set, it will not be stored in the nginx Cache.



Notice we also added a dynamic header, `X-Fastcgi-Cache`, which is really helpful for debugging. This header will display `HIT` if the file was served from the cache and `MISS` if it was not and was served from PHP. You can log this in your `access_log` and use it to calculate cache hit ratios.

PHP-FPM with and without a proxy cache

I quickly benchmarked this on an i2.8xlarge EC2 server really simply— I setup a blank laravel application (no database), with one page (`/public`) coded to always return `Cache-Control: public` and the other (`/private`) setup to always return `Cache-Control: private`. I ran `ab` (Apache Benchmark) and benchmarked 100,000 requests— here are the number of requests per second:

```
1 /public - 24690 requests per second, 4ms per request
2 /private - 3253 requests per second, 30ms per request
```

The takeaway here is— it is much, much, much faster to skip PHP and serve files directly from an HTTP cache if at all possible. The results are incredibly visible with the most basic of PHP scripts, and we'd see an even more substantial difference if the PHP code was doing more complex work like hitting the database multiple times. That being said, sometimes, it's just not possible to do if the data is too dynamic. But for the pages where it does make sense— boy this can be a huge win.



**How is the fastcgi cache stored on disk?

The cached responses that the nginx fastcgi cache saves are stored on disk and can be easily manipulated. The output of the response from PHP-FPM is captured and stored in `fastcgi_cache_path` (`/tmp/cache` in our example). The filename is predictable and derived from the derived cache key of the request. For instance, for a GET request to `https://domain.com/index.html`, the derived cache key is `httpsGETdomain.com/index.html`. The response body will be stored in a file that is named `md5(httpsGETdomain.com/index.html)`, or `d5c94ba8b944742f64c115fa8c8e65ea`.

The cache files aren't just stored at the base of `/tmp/cache`, though. Nginx uses a two level folder structure to avoid having millions of files in a single directory which is bad for performance. The last three characters of the filename are used to build the folder structure, so for the example request above, the response body will be stored in `/tmp/cache/a/5e/d5c94ba8b944742f64c115fa8c8e65ea`. Why is this useful? If you delete this file, it will remove it from the fastcgi cache and force nginx to re-pull the page next time it's requested.

Case Study: CakePHP Framework Caching

This is going to be a really awesome case study— I’m really pumped about it because it’s an interesting problem that not only took an incredible amount of blood, sweat, and tears to solve, but is also extremely applicable to the theme of this book because it’s something that could *only* be replicated at scale.

So first things first, I’ll set the stage with the some background and context and walk you through how I debugged, found the core issue, failed at solving it several times, and finally found the perfect solution.

CakePHP, you devilish dessert

One of our several backend applications that I work on at [Life360¹³³](#) is written in CakePHP. While I don’t think CakePHP is awesome, it gets the job done well enough and is fairly easy to work with.

CakePHP is a *large* framework. It handles everything from autoloading your PHP files for you, to introspecting your database models and learning about the different columns, to providing a faux query cache to cut down on the number of SELECT queries you make to MySQL. Typical framework stuff.

Without some type of cache to remember all of this information about your code between requests, Cake would have to regenerate it for each pageload, adding a significant amount of overhead. In order to remember the data, Cake provides a pluggable system cache which can store this data using APC, flat files, Memcache, or MySQL.

While the exact mechanics are specific to CakePHP, the problem itself is generic— almost every substantial PHP and non-PHP framework does something similar, persisting framework and system cache data somewhere.

APC gets removed from PHP 5.4 (aka, “the core problem”)

Our CakePHP framework cache was configured to use PHP’s APC cache. APC works great for this type of workload because it’s extremely fast and the data doesn’t need to be shared between multiple servers. Sadly, APC has been removed in PHP 5.5 in favor of supporting the much more advanced (and now, open-source) Zend OpCache.

Remember that APC provides two different functions to PHP— it provides a userland cache, similar to Memcache, which is how our CakePHP framework cache is using it, but its primary

¹³³<http://www.life360.com>

function is to cache compiled PHP bytecode, allowing our code *execute* faster. The faux-memcache functionality is a secondary feature to its primary role as a bytecode cache.

Zend OpCache is better than APC— it benchmarks faster and is all around more modern. Hands down, it makes a ton of sense and it's why it has become the standard for PHP 5.5. Zend OpCache does one thing and does it well— it **only** does bytecode caching and doesn't offer a comparable userland cache like APC.

The reason I mention this is because when upgrading from PHP 5.4 to 5.5, we could no longer use APC for our CakePHP system cache. At first glance, it doesn't seem like it's that much data and it's not like it changes very much, so we just modified our configuration and told CakePHP to store its system cache data in flat files instead of APC— easy enough, right?

This worked seemingly well. No problems, it worked well at scale (400 million requests per day), everything was seemingly great. Little did I know, this small change would cause me to pull my hair out for months.

The life and death of a CakePHP request

Let's briefly run through how a PHP request is dispatched in CakePHP (or really, any MVC framework), and where the framework caching happens.

1. Web request is sent from the browser to Nginx, Nginx sends the request to PHP-FPM
2. PHP-FPM loads the bootstrap file of the framework, `index.php`
3. Bootstrap file loads framework configuration, **reads the framework cache data from the filesystem (if it doesn't exist, it has to regenerate it)**, and eventually invokes your controller.
4. Your controller code runs and generates a response for the client.
5. The framework **saves the newest system cache data to the filesystem** and sends your response upstream to nginx and eventually the browser.

There are two points at which the framework needs the cache data— at the beginning and end of the request. If the data doesn't exist, it has to regenerate it (figure out which classes map to which files, figure out which columns are in which tables, etc). Simple enough.

If it smells like a race condition...

So, with a fair understanding of how the cache works, what it accomplishes, and when it runs, let's move on to discussing where things started to fall apart.

It's typical Wednesday afternoon. We have some new code that we want to merge from our development branch to `master` and subsequently deploy to production. No big deal. Luckily, this is pretty automated and as soon as `master` gets updated, `chef`¹³⁴ kicks off, updates the production servers with the latest code and reloads PHP-FPM.

¹³⁴<http://www.getchef.com/>

But here's the weird part— when new code is deployed, we start seeing PHP-FPM processes pile up, quickly hit 100% CPU usage and spike server load to 80. We run out of free PHP-FPM children in no time, exceed the PHP-FPM backlog, and nginx starts throwing 502 Bad Gateway HTTP responses. Bad news! And as quickly as it came, the problem would silently retreat and go away, never to be seen again until the next deploy.

It sort of became a ritual. Deploy new code, see a jump in errors, watch it mysteriously go away. It plagued me for months— no one wanted to deploy code, worried about causing brief downtime. We had a list of assumptions why this was happening (none of them were true), and it didn't go away until I sat down for 4 days straight and tested each one.

Still with me? Great. I want to walk you through this problem like we're working on the project together. I think the **process** it's what's helpful here— the answer itself isn't that valuable, that's why I'm walking you through the entire problem from start to finish instead of just giving you the solution.

My initial assumptions were that the increase in 502 Bad Gateway errors were caused by one of the following problems.

1. Too many connections to Memcached due to reloading PHP
2. Reloading PHP-FPM (`kill -USR2`) wasn't working as expected
3. Zend OpCache taking time to warm up, causing many PHP requests to run without compiled bytecode and spiking CPU usage

Busting the assumptions

The first step to being able to test the three assumptions was to be able to replicate the problem. Unfortunately, it was nearly impossible to replicate this on a development server, because without a heavy flow of real traffic, code would deploy without issue. I segregated a production server that was serving traffic and force deployed to it over and over. Bingo— CPU spike, 502 errors, the works.

I can replicate the issue over and over again. Now I'm ready to start testing my assumptions, one by one.

Too many connections to Memcached from reloading PHP

My first assumption was that during the graceful reload of PHP, the old workers might still be holding onto their old connection to Memcached while the new ones start up. Memcached has a connection limit, if we exceed this, the memcached client will start throwing exceptions and these exceptions could be causing the problems we're experiencing.

Easy to test— if we force a deploy on our test box with 256 PHP-FPM workers, we should see the number of connections on our Memcached server go up by exactly 256. We can easily monitor this with telnet.

```
1 telnet memcache_server.ip 11211
2 > STATS
3 < STAT curr_connections 3840
4 ...
```

Okay, we have 3840 connections to our memcache server, if we watch it while force deploying on our test box, we should see it blip up to 4096.

Deploy.. wait... and.. Nope. That wasn't it, after the deploy I see the number of connections hold steady at 3840. Myth busted.

Reloading PHP-FPM (`kill -USR2`) doesn't work as expected

When we deploy code, we gracefully reload PHP-FPM. Why reload PHP-FPM on deploy? We have `opcache.validate_timestamps=0` and `opcache.revalidate_freq=0` set in our `php.ini`. What these settings do is make it so that once Zend OpCache compiles the bytecode for a particular PHP file, it will never check the file for updates again. That is to say, if the file is modified or changed (manually or by a deploy), PHP will not read the latest updates until PHP-FPM is restarted or reloaded.



What does graceful reloading do? When you send the PHP-FPM master process the `USR2` signal, it waits for each child process to finish the current request and then replaces the old process with a new one. This allows PHP to gracefully restart without dropping any requests.

This one is easy to test. Instead of force deploying PHP, I just need to manually `kill -USR2` it on my test box and check for a spike in errors. So, I run:

```
1 $ kill -USR2 `cat /var/run/php-fpm5.pid`
```

And what do I see! A small jump in `502 Bad Request` errors. Is this it? Did we figure out our culprit? No, but it led me to discovering a small contributor. After reading the PHP-FPM source, I discovered that I needed to set `process_control_timeout` in my `fpm.conf` file, otherwise PHP-FPM will stall and wait forever for all of the children to finish their request. Setting `process_control_timeout=10s` made gracefully reloading PHP much smoother. But the main issue still persisted—a deploy, even with this change, still causes high CPU and 502 errors. Onwards.

Zend OpCache is taking a long time to warm up

The assumption here is that the OpCache takes a little while to warm up and that without the compiled PHP bytecode, the servers are running the PHP code slower and using much more CPU.

I was writing a bunch of logic in Chef to pre-warm the cache before deploying, I decided I should spend a few minutes testing this assumption before investing a ton of time trying to fix it.

I proved this assumption wrong in two ways:

The first (and easiest) way—I disabled Zend OpCache completely with `opcache.enable=0` in `php.ini` and reloaded PHP. While I saw that load was certainly higher, the server was not throwing 502 Bad Gateway errors nor was the CPU at 100%. Busted.

The other way I tested was restarting PHP and after the first request printing the contents of `opcache_get_status()`. This function shows all of the files currently cached by Zend OpCache. It turns out that even with a very heavy framework like CakePHP, almost 75% of the code was compiled to bytecode after the first request.

Bummer! We're out of assumptions. Back to the drawing board.

Now what? I'm out of ideas

All of the assumptions that I had went down the tube as I systematically debunked each one. I was SURE that I knew what the problem was, but as it turns out, I was completely wrong. I'm out of ideas and what do I besides give up and pull out my hair?

strace to the rescue!

When you need a partner to help you debug, `strace` has got your back. It's one of my favorite tools on Linux and while you can spend an entire semester learning about all of the features it has, the simplest use-case can often be all that you need.

`strace` gives you the ability to attach to a running, live process (while marginally impacting performance—okay for our use, but don't `strace` a production MySQL server, for instance) and view what system calls it is making. Sure, you can't see the exact code that's executing, but you can see external network requests, filesystem calls, and other low-level stuff.

So, I deploy once again, and find myself a PHP-FPM worker to attach to:

```
1 $ ps aux | grep php-fpm
2
3 myapp 128131  2.9  0.0 324132 57396 ?          R      10:54  11:50 php-fpm: pool w\
4 ww
5 ...
```

Perfect, quickly I attach to this process 128131 and start `strace`'ing it, to see what it's up to. Lo and behold, I see something interesting right away. The output flies by at mach-1, but sure enough it keeps pausing for several seconds after this block of system calls... it's stalling here!

```

1 open( "/tmp/cake_core_file_map", O_RDWR|O_CREAT, 0666) = 10
2 fstat(10, {st_mode=S_IFREG|0666, st_size=29199, ...}) = 0
3 lseek(10, 0, SEEK_CUR)          = 0
4 close(15)                     = 0
5 flock(10, LOCK_EX)           = 0

```

So what the hell is this? Well, PHP is stalling at the last call to `flock`. What's `flock`? It's a system call to obtain a lock on the file, so no other programs can write to it. Why, it's waiting to get an exclusive file lock and blocking until it can get one. What file is it trying to obtain an exclusive lock on? OUR CAKEPHP SYSTEM CACHE FILES! Somewhere in the internal CakePHP code, it's locking the files before writing to them!

And that's the core problem. We found it. When we deploy a new code release, we're wiping out our `/tmp` directory—we WANT the CakePHP cache files to be refreshed after a deploy because a database table or model may change. But what's happening is that a stampede of 256 PHP-FPM workers are trying to update the cache simultaneously, each requesting an exclusive file lock to do so. And that's where it's blocking—because of the competition over obtaining a file lock. The blocking causes requests to pile up, which causes our socket backlog to fill up, and forces Nginx to start throwing 502 Bad Gateway errors. That's it!

The solution(s)

Now that we know the issue, it's much easier to solve. We need to speed up `flock` (or avoid it all together) so that during a deploy, we don't have 256 PHP-FPM workers all competing on trying to update the same files. There are a couple of different ways to do solve this problem, and I'll quickly cover which one I used (and the ones that failed).

Screw files, use Memcache!

My first reaction was to just use Memcache. Remember earlier we talked about the pluggable backend for the CakePHP system cache? Well one of the backends is Memcache, so screw it, let's just use that. I modified the configuration, plugged in my memcache server details, and it all seemed to work. But, curiously got the best of me, and I wondered how much extra traffic this was adding to the Memcached server.

So I removed the change, and used `ifstat` to get a baseline for network traffic on this app server...

```

1 $ ifstat -i eth0
2 KB/s in  KB/s out
3 3154.15  2660.01
4 3003.02  2561.97

```

Ok, about 3MB/s in and out. Next, I re-enabled the memcached backend in CakePHP and checked again.

```

1 $ ifstat -i eth0
2 KB/s in  KB/s out
3 41931.34 33154.11
4 39129.12 31929.23

```

Oh wow, we added almost 40MB/s of traffic to and from our memcache server by using it for the CakePHP system cache. FOR ONE SERVER. Multiply it across all of our app servers and wow, it quickly becomes unsustainable. The CakePHP cache is only a couple of kilobytes but it needs to read it out and write back out the full contents for each request. Bummer, memcached isn't go to work for this.

APCu, the userland cache for PHP 5.5

Next up, I found [APCu¹³⁵](#), the userland part of APC reimplemented for PHP 5.5. I tested this, and while it worked just fine, it felt dirty to me. This is a beta-level package, without much history or testing, and who knows if it will continuously be updated to work with PHP 5.6, 5.7, and so on. Might as well make the clean cut instead of hacking the functionality back in with a legacy module. I will say this— out of all the solutions, this was probably the easiest and most straightforward. APC worked great for this solution in PHP 5.3 and 5.4.

An independent cache for each PHP-FPM worker

I had to wake up at 4:45AM on a Saturday morning to catch a flight to San Francisco. As I was flossing in my bathroom, it hit me in the face. Instead of relying on a *global* CakePHP system cache for all 256 of my PHP-FPM workers, why don't I have a unique file cache per-worker? This will get rid of the competition for the file lock, since each worker is single threaded and can only serve one web request at a time.

It gets even easier— it would only require one change:

```

1 <?php
2 // Old Configuration
3 Cache::config('_cake_core_', array('prefix' => 'cake_core'));
4
5 // New Configuration
6 Cache::config('_cake_core_', array('prefix' => getmypid() . 'cake_core'));

```

All we did was add `getmypid()` to the filename of the CakePHP framework cache— since each PHP-FPM worker runs as a child processes and has its own process id, each PHP-FPM worker gets its own unique cache.

The best ideas come when you're flossing.

I was excited to land on the west coast and test my theory. Within minutes of landing, I tested my theory, and it worked. Except for a small problem. When I checked filesystem usage with `iostat`, I was saturating the disks! It was too much load on the disk to have 256 independent PHP-FPM caches! Blah!

¹³⁵<http://pecl.php.net/package/APCu>

```
1 $ iostat /dev/xvdb -x 1
2 Device:      rrqm/s   wrqm/s     r/s     w/s     svctm %util
3 xvdb        1273.00  893.00  1263.00 893.00   1.21  100.00
```

Instead of going back to the drawing board AGAIN, I just iterated on this solution. The cache files were pretty small, less than 50KB each, so even with 256 independent caches we are only talking about 15MB or so— it's just getting read and re-written several thousand times per second.

I created an in-memory filesystem with `tmpfs` and mounted it in the location of my cache directory— giving CakePHP the *illusion* of using the filesystem, while really storing the cache in memory. This solved all of the problems and was ultimately the solution I used.

```
1 $ mount -t tmpfs -o size=1000M,mode=0755 tmpfs /tmp/cache
```

What a ride it was. I hope this case study showed you some real-world debugging, reasons to floss your teeth, and why at scale, sometimes even the obvious solutions don't always go as planned :)

Case Study: Optimizing image handling in PHP

As you can probably imagine, resizing images is a huge part of the workload that our systems spend time on at Twitpic. With the 8MP camera in an iPhone 5 (and up to 13MP in some of the newer Android phones), even mobile users are capable of snapping and uploading pictures that easily exceed 10MB.

Almost anytime an image is uploaded by a user into a web application, it needs to be resized and post-processed for a few different reasons.

1. Reduce the file size, sending a 10MB image to the browser is *not* a best practice :)
2. Standardize on a dimension size. For Twitpic, it's 600px wide
3. Standardize on a particular image format (usually, but not always, JPEG)
4. Generate multiple image versions (several thumbnails, retina versions)
5. Rotate the image to align it with the orientation of the camera when it was taken
6. Strip out EXIF, Camera Model, GPS Coordinates, and other sensitive information from headers

Processing images in your stack can be tricky, though, since post-processing and resizing tend to be CPU intensive and moving large files around consumes a noticeable amount of Disk I/O and Network Bandwidth.

The naive “PHP Tutorial” way

If you look for information on basic image processing in PHP, you'll almost exclusively find references telling you that you should be using [ImageMagick¹³⁶](#)— and for good reason! ImageMagick is a very robust, well-supported library can convert, resize, and modify almost any *any* image format. It's like Adobe Photoshop™ for the command-line.

What does the naive, unoptimized image upload and resizing code look like with ImageMagick? The example below shows you some example code to receive a file upload, scale it to a 600px JPEG, and create a 150x150 thumbnail.

¹³⁶<http://www.imagemagick.org/script/index.php>

```
1 <?php
2
3 class Image_Controller extends Controller {
4
5     // Image upload is posted to this method
6     public function create() {
7
8         if (isset($_FILES["upload"])) {
9
10             $tmp = $_FILES["upload"]["tmp_file"];
11
12             // Create the Scaled 600x400 Size
13             exec("convert -resize 600x400 {$tmp} /u/scaled.jpg");
14
15             // Create the 150x150 Thumbnail
16             exec("convert -thumbnail x300 -resize 150x< -resize 50%
17                   -gravity center -crop 150x150+0+0 +repage
18                   {$tmp} /u/thumbnail.jpg");
19
20             // Delete the temporary file
21             @unlink($tmp);
22         }
23     }
24 }
```

Well, what's so bad about this code? It's just taking the raw image (from `$tmp`) and running some ImageMagick commands on it. Performance-wise, there are a handful of problems wrong with the code. Can you guess them?

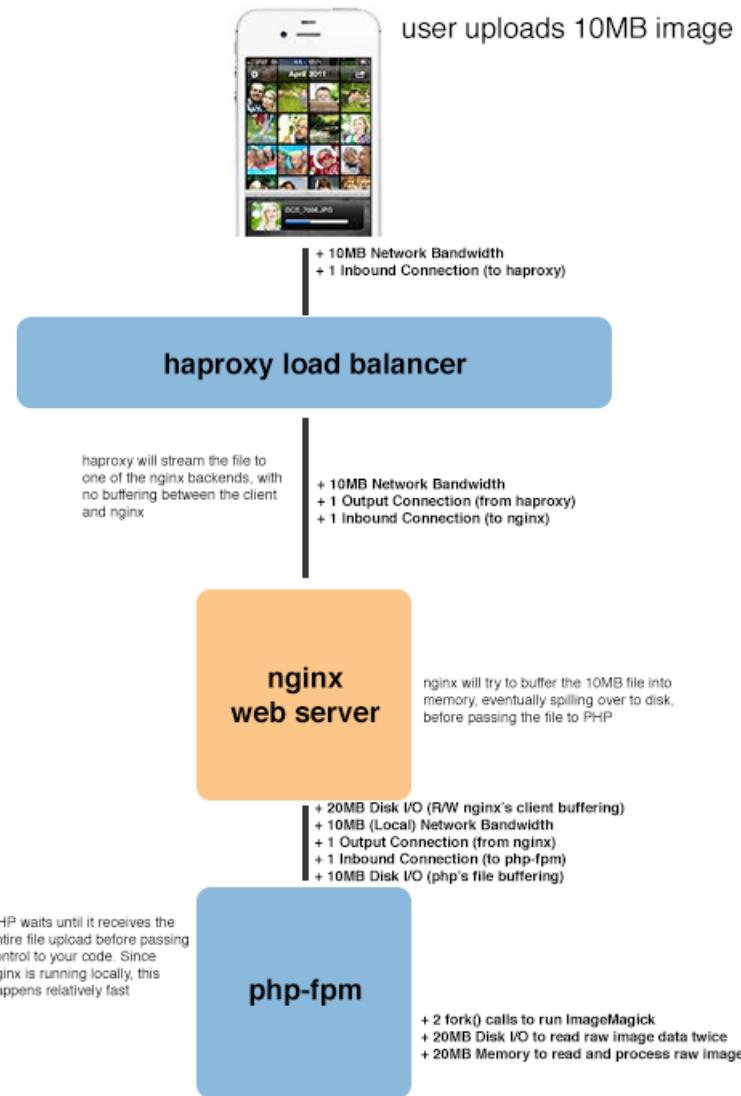
1. Image resizing is a CPU-heavy and slow process, especially on big images. If possible, resizing and post-processing should be moved to a Resque worker. Admittedly, in most scenarios the user expects the image to be resized and available immediately, so a background worker is not always possible. We'll assume, for the rest of this case study, that using a Resque worker is not an option.
2. ImageMagick doesn't have a very actively updated PECL extension, so using the C library is out of the question—we have to use `exec()` to fork a new process and run it from the command-line. Calling `exec()` is a SLLOOOWW call, and on EC2 (or any Xen-based virtualization), can take up to 1 or 2 seconds JUST FOR THE process to `fork()`! Not only is it slow, but it can also be a vector for DOS attacking your servers.
3. When making multiple versions of an image, it's better to use the smallest size available instead of resizing the raw image multiple times. It's much faster to create the 150x150 thumbnail from the newly created 600x400 instead of re-processing the raw 10MB, 4096x2048 pixel image each time.

It's not all in the code— tracing file uploads through the stack

Fixing the three issues previously mentioned will certainly improve your image handling performance, but there's a less apparent problem hidden in the LHNMPRR stack.

Let's assume a 10MB image is uploaded by one of your users. This is how it would flow through your stack if you designed it exactly as described in this book.

Deconstructing a file upload



Moving a 10MB image through the LHNMPRR stack

1. The user kicks the process off by uploading their file, via web browser or mobile app, by sending an HTTP POST with the file (**+10MB Network Bandwidth**). The HAProxy load balancer receives this file upload and immediately starts streaming it to one of the nginx

App Servers (+10MB Network Bandwidth). HAProxy does not buffer any of the data from the client in memory or onto disk, it passes straight through.

2. Nginx begins receiving the file upload from HAProxy and begins buffering it into memory. Depending on the size of `client_body_buffer_size` (8KB out of the box), nginx will spill over to disk and stop buffering the file in memory until the entire file has been transferred. **(+10MB Disk Write I/O)**
3. Once the file upload has been completely transferred from HAProxy, nginx will begin reading the file back **(+10MB Disk Read I/O)** as it transfers it to PHP-FPM using FastCGI **(+10MB Local Network Bandwidth)**. Since PHP-FPM is running locally and (hopefully) using Unix Sockets, this is a quick operation.
4. PHP-FPM notices the file upload and begins streaming it, you guessed it, back onto disk **(+10MB Disk Write I/O)**. This PHP-FPM worker will be stuck in a “busy” state and your PHP code will not begin executing until the entire file is received from nginx.
5. Once the file upload *finally* makes it into PHP and your code starts running, the ImageMagick resizing runs, reading the file into memory twice to process it. **(+20MB Disk Read I/O, +20MB Memory)**.

It turns out, moving that image around actually consumes far more than just 10MB of resources! In fact, at the end of the day, it takes 50MB of Disk I/O, 30MB of Network Bandwidth, and 20MB of Memory. Crazy! Imagine if you were working with videos!

Resource	Amount
Network Bandwidth	30MB
Disk I/O	50MB
Memory	20MB

What an inefficient process that just seems to juggle the exact same data around from system to system. We can optimize this by avoiding Disk I/O and trying to keep the data in memory for as long as possible.

There are two places the file goes in and out of memory— the first is during the nginx buffering process and the second is when it’s received by PHP. We’ll tackle each one individually.

On the nginx side, we can change the `client_body_buffer_size` setting to something larger than 8KB, which would mean larger amounts of data staying being buffered in memory before hitting disk.

Instead of doing it this way, I prefer to keep `client_body_buffer_size` at the default and setup `tmpfs` as covered in Chapter 5. With `tmpfs` setup, the `client_body_temp_path` setting can be changed to the `/tmp` directory, optimizing out the Disk I/O overhead while still keeping the nginx memory footprint low.

On the php side, it’s easy since the hard work has been done for us with the `tmpfs` change. If we just modify `php.ini` and change `upload_tmp_dir` to somewhere in `/tmp`, the file data just gets written back out into memory when it’s moved from nginx to PHP. Not ideal, but much better than getting pushed back-and-forth from disk 3 times.



Unlike HAProxy, when using nginx with the HTTP Proxy Module or FastCGI, the entire client request (from a load balancer or even directly from a web user) will be buffered in nginx before moving upstream—the data will never be streamed in real-time. On the plus side, this prevents [Slowloris attacks](#)¹³⁷, but will break PHP’s file upload progress meter. Nginx offers a [Upload Progress Module](#)¹³⁸ if upload progress is an essential feature.

The Nginx HTTP Upload Module

Even with both of the `tmpfs` changes described above, it still feels gross to me because no matter which way you slice it, the data gets copied around more times than it needs to be. And, `tmpfs` won’t work for everyone! If you’re handling very large file uploads (i.e. videos), you don’t really want those large files buffered into memory anyways.

The more efficient, albeit more complicated and involved, solution is to use the [HTTP Upload Module](#)¹³⁹ that comes with nginx. Instead of sending the raw file upload to PHP, the HTTP Upload Module will just send the *file location* to PHP, completely avoiding the double copy.

If you’re using the Dotdeb Apt Repository as mentioned in Chapter 5, great news—you just need to install `nginx-extras` to get the HTTP Upload Module. Otherwise, you’ll have to compile it from source, which I’ll leave as an exercise for the reader.

You can quickly determine if you have the HTTP Upload Module with `nginx -V` and `grep`. It will show `1` if you have the module, `0` if you don’t.

```
1 > nginx -V 2>&1 | grep -c "nginx-upload-module"
```

If you don’t have it, install it with `apt-get`.

```
1 > apt-get install nginx-extras
```



I had some trouble installing `nginx-extras` on Ubuntu 12.04 (because of `perlapi` errors), but was able to install it fine on Debian Squeeze. If you’re on Ubuntu, you may need to compile nginx from source to get the HTTP Upload Module.

Hold up, before you get going, the module needs to be configured to intercept the file uploads. Pop open `my_app.conf` from Chapter 5 and add the following `location` block.

¹³⁷<http://en.wikipedia.org/wiki/Slowloris>

¹³⁸<http://wiki.nginx.org/HttpUploadProgressModule>

¹³⁹<http://wiki.nginx.org/HttpUploadModule>

```

1 > vi /etc/nginx/sites-available/my_app.conf
2
3 location /upload {
4     upload_pass /index.php;
5     upload_store /tmp/nginx/upload 1;
6     upload_set_form_field $upload_field_name.name "$upload_file_name";
7     upload_set_form_field $upload_field_name.content_type "$upload_content_type";
8     upload_set_form_field $upload_field_name.path "$upload_tmp_path";
9
10    # Delete the file if PHP returns any of these errors
11    upload_cleanup 400 404 499 500-505;
12 }
```

Alright, all configured, just have to reload nginx and create the temporary directories needed for upload_store.

```

1 > service nginx reload
2 > mkdir -p /tmp/nginx/upload/{1,2,3,4,5,6,7,8,9,0}
```

Lastly, the only change we have to make to the code is to use the `$_POST["file_path"]` variable instead of `$_FILES["upload"]["tmp_file"]`.

```

1 <?php
2
3 class Image_Controller extends Controller {
4
5     // Image upload is posted to this method
6     public function create() {
7
8         if (isset($_POST["file_path"])) {
9
10            $tmp = $_POST["file_path"];
11
12            // Create the Scaled 600x400 Size
13            exec("convert -resize 600x400 {$tmp} /u/scaled.jpg");
14
15            // Create the 150x150 Thumbnail
16            exec("convert -thumbnail x300 -resize 150x150 -gravity center -crop 150x150+0+0 +repage
17                  {$tmp} /u/thumbail.jpg");
18
19            // Delete the temporary file
20            @unlink($tmp);
21        }
22    }
23 }
24 }
```

We also need to change our form to POST the file to `/upload` instead of the normal `index.php` endpoint (specified by the `location` block that we added to `my_app.conf`).

Benchmarking the file upload process

I have something embarrassing to admit. I wrote this entire chapter based on theory alone. I've implemented all of these techniques at Twitpic, because on paper— IN THEORY, it makes complete sense. These changes will cut our the cruft, reduce useless data shuffling, and optimize the whole process. Right? Right!?

Well, when I went to benchmark each individual change, it turns out... not so much. In fact, each optimization described (`tmpfs`, `client_body_buffer_size`, and `HTTP Upload Module`) only added, at best, a 5% performance improvement over the “default” setup. Oops. And Ouch. It's a lesson in premature optimization, for sure, and why you should benchmark everything! Regardless, I've kept this case study for two reasons:

1. It's really important to understand *EXACTLY* how data finds its way to your PHP code. Following the code path is a journey that everyone should make. This knowledge will help you debug in the future.
2. To drive home the whole premature optimization thing and that even if an optimization looks correct on paper, it might not actually pan out in the real world.

The only thing that seemed to make a large, measurable impact was the difference between ImageMagick and GraphicsMagick, which I talk about in the next section.

I benchmarked the different settings, one by one, using `ab` (Apache Benchmark), with 1000 uploads and a concurrency of 20— that is, 1000 independent uploads with 20 happening at the same time. I used a 10MB JPEG as the test upload. The server, a EC2 m3.xlarge, was running Debian Squeeze 6.0.6, nginx 1.2.7, and PHP 5.4.13 for the benchmark.

These are the different tests, explained one-by-one.

tmpfs

I enabled `tmpfs` and changed `client_body_temp_path` in nginx and `upload_tmp_dir` in php to use it. Little to no difference between the stock setup. Same number of requests/second, same time per request.

Here's why:

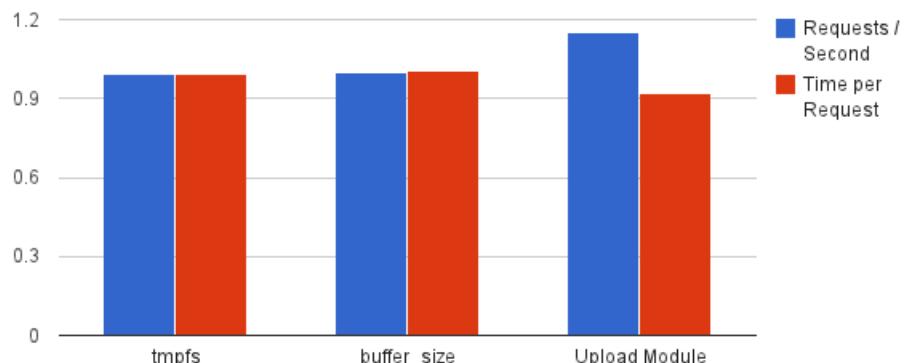
1. Nginx doesn't `fsync` the temporary data to disk, and the writes are mostly are mostly sequential, so writing the upload data to disk doesn't have a huge impact on IO usage.
2. When PHP reads the data back from disk, it's reading the entire file sequentially, most of which is already in-memory via the operating system's disk buffer.

Increasing `client_body_buffer_size`

I increased the `client_body_buffer_size` variable in nginx to 20MB, so nginx would be able to buffer the entire 10MB file upload in memory and never have to hit disk. Again, no performance improvement (due to the reasons above), except it increased the memory usage of my nginx worker processes to 200-300MB each, instead of the typical 25MB. Fail! There seems to be almost no reason to increase `client_body_buffer_size` from the default 8KB.

HTTP Upload Module

My last hope— I knew this *had* to improve performance. Not only does it save the file from being copied, but it saves PHP from having to parse the 10MB of POST data. I saw a small 5% improvement in performance here, mostly in reduction of CPU usage by PHP, as it's less data PHP needs to chug through before it can run your code. Not worth it.



Benchmarks of various changes

Benchmarking file uploads with ab (Apache Benchmark)

I figured I would make a quick note about how you can test this yourself— it took me a long time to figure out a way to easily benchmark/test file uploads from the command line. There are plenty of tutorials on ab, but none really mention how to use it to upload files.

You can only upload raw multipart/form-data with ab, it won't do any magic for you, so you actually need to assemble the raw multipart/form-data to upload an image with ab. Something about "manual" and "multipart/form-data" just sounds awful.

I used this PHP script to generate it for me, you just need to edit the form field name, file path, and file name. It'll take care of everything else. I used a PHP script because I had a hell of a time getting all of the \r\n line breaks to be perfect doing it manually— if the format isn't perfect, your uploads will get ignored by nginx and php.

```

1 <?php
2
3 $boundary      = "1234567890";
4 $field_name    = "upload";
5 $file_path     = "./";
6 $file_name     = "test.jpg";
7
8 echo "---{$boundary}\r\n";
9 echo "Content-Disposition: form-data; name='{$field_name}';";
10 echo " filename='{$file_name}'" . "\r\n";
11 echo "Content-Type: image/jpg\r\n";
12 echo "\r\n";
13
14 echo file_get_contents($file_path . $file_name);
15
16 echo "\r\n";
17 echo "---{$boundary}--\r\n";

```

Put the script above into a file called `generate.php` and create a `post.txt` file holding the multipart/form-data with the following command:

```
1 > php generate.php > post.txt
```

Now, you can use `post.txt` with `ab` to start benchmarking your own file uploads.

```
1 > ab -n 50 -c 10 -p post.txt -T "multipart/form-data; boundary=1234567890" http\ 
2 p://localhost/upload.php
```

ImageMagick vs GraphicsMagick

There's a lesser known image library called [GraphicsMagick¹⁴⁰](#)— a leaner and faster fork of the ImageMagick code. That's a win in itself (given the same source image and settings, GraphicsMagick will often produce a smaller output image in less time), but it also has a regularly updated PHP C Extension. Having a Native PECL extension is a huge win, because it us allows to remove the two `exec()` calls from the code and subsequently the two poorly performing `fork()`s.

The installation is pretty painless, too.

```
1 > sudo apt-get install php5-dev php-pear build-essential libgraphicsmagick-dev
2 > sudo pecl install --force gmagick
3 > echo "extension=gmagick.so" > /etc/php5/conf.d/gmagick.ini" | sudo sh
```

You can grab the docs for [PECL GMagick¹⁴¹](#) here, but it uses very similar options as ImageMagick, except with an object oriented interface instead of command-line arguments.

¹⁴⁰<http://www.graphicsmagick.org/>

¹⁴¹<http://pecl.php.net/package/gmagick>

```
1 <?php
2
3 class Image_Controller extends Controller {
4
5     // Image upload is posted to this method
6     public function create() {
7         if (isset($_FILES["upload"])) {
8             // Create the Scaled 640x800 Size
9             $gm = new GMagick();
10            $gm->readImage($_FILES["upload"]["tmp_file"]);
11            $gm->setCompressionQuality(90);
12            $gm->scaleimage(640,800);
13            $gm->write("./f/scaled.jpg");
14
15            // Create the 150x150 Thumbnail
16            $gm->croptumbnailimage(150,150);
17            $gm->write("./f/thumbnaill.jpg");
18
19            unlink($_FILES["upload"]["tmp_file"]);
20        }
21    }
22 }
```

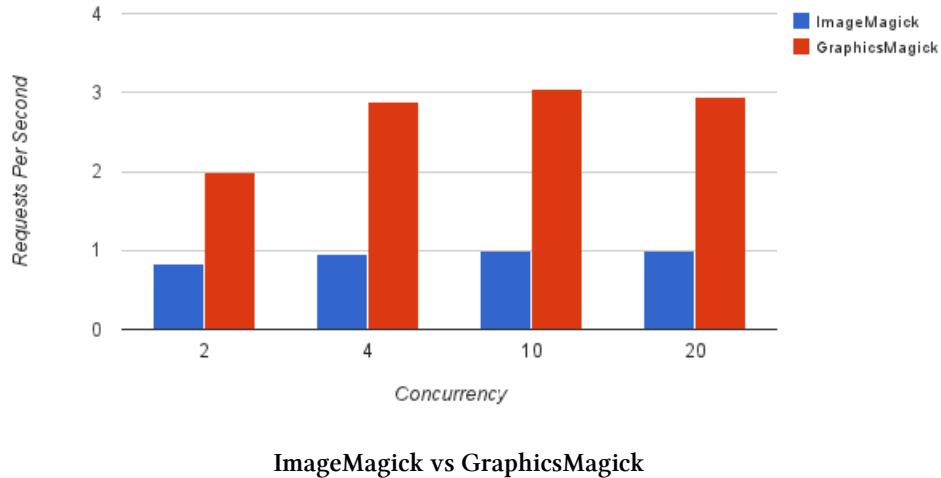
Hooray! No more nasty `exec()` calls, and since we can share the `GMagick` object for multiple resizes, we only have to read the raw image data into memory once opposed to twice with `ImageMagick`. *This will only work if you're downsizing every step.*



Newer versions of `GraphicsMagick` and `ImageMagick` are built with OpenMP, a framework for parallelizing image processing across multiple cores. In theory, sounds great—but I ran into a number of issues with random seg faults and crashes. I prefer to either recompile `libgraphicsmagick` with `--disable-openmp` or set `putenv(MAGICK_THREAD_LIMIT=1)` in PHP to disable OpenMP.

Benchmarking ImageMagick vs GraphicsMagick

In a fairly straightforward benchmark, I found that `GraphicsMagick` was nearly twice as fast as `ImageMagick` in terms of requests per second—jumping to more than 3x faster if you're able to re-use the same `GraphicsMagick` when doing multiple resizes. That's an incredible payoff, with little effort, especially if you're working with an upload-heavy application or have to do the image resizing inside of the web request.



Allowing Large File Uploads

Setting up your stack to handle large file uploads can feel like a pain because there are a few different settings you need to change in `php.ini` and `nginx` to allow big files to pass through.

`nginx.conf`

`client_max_body_size`

Set this to the same as `upload_max_filesize` in `php.ini`. `128M` is a good starting point.

`php.ini`

`upload_max_filesize`

The maximum size of the actual file upload itself. Should be set the same as `client_max_body_size` in `nginx.conf`

`post_max_size`

The maximum size of the entire HTTP POST body, including the multipart file upload. Should be set slightly higher than `upload_max_filesize` to take into account any extra POST parameters and metadata. I usually set it as 1MB more than `upload_max_filesize`.

`memory_limit`

PHP says that it needs enough memory to read the entire file into memory. I typically just disable `memory_limit`, but if you're not doing that, than you need to set it atleast the size of `post_max_size`, preferable a little higher.

`max_file_uploads`

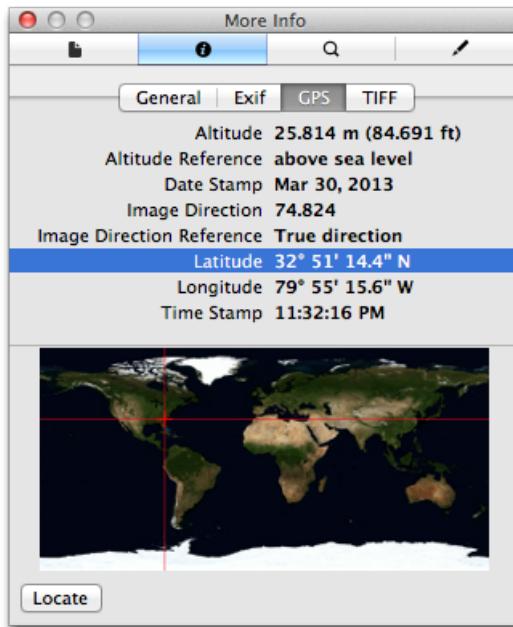
The maximum number of files that can be in a single POST body, used to provide a sane limit and prevent circumventing `upload_max_filesize` with lots of small files. Worth pointing out—PHP does count empty file upload fields in a form against this limit.

Stripping EXIF Metadata

When a user uploads a JPG, they are not only sending a picture, but also a wealth of metadata that's hidden inside of the image. This EXIF metadata can be used for interesting data projects—things like the timestamp, camera type, aperture, shutter speed, and flash brightness are stored within the EXIF data. Cool!

```
1 print_r(exif_read_data("file.jpg));  
2  
3 => Array(  
4     [FileName] => IMG_1043.jpg  
5     [FileSize] => 2464042  
6     [Make] => Apple  
7     [Model] => iPhone 4S  
8     [Software] => QuickTime 7.7.1  
9     [DateTime] => 2013:04:01 09:28:30  
10    [HostComputer] => Mac OS X 10.8.3  
11    [ExposureTime] => 1/20  
12    [ISOSpeedRatings] => 200  
13    [ShutterSpeedValue] => 2779/643  
14    [GPSLatitude] => Array  
15        [0] => 32/1  
16        [1] => 5126/100  
17        [2] => 0/1  
18    ...
```

But there's also a scary amount of private data that gets tagged inside of the image metadata too, including GPS data. On a picture taken with my iPhone 4S, the Lat/Long EXIF data is accurate within 5 feet of my apartment.



Some information contained in the EXIF metadata

If you're handling user uploads and displaying them publicly, it's your responsibility to strip out the EXIF tags from the files that your users upload. Luckily, it's pretty easy with GraphicsMagick and a call to the `stripimage()` function.

```

1 <?php
2 $g = new GMagick("file.jpg");
3 $g->stripimage();
4 $g->write("without_exif.jpg");

```

When you *resize* an image with GraphicsMagick, it will not preserve the EXIF data. However, many sites offer a “raw” or “view original” function, where it is absolutely necessary to strip the EXIF data from the image.

Autorotation

Another common issue that people run into with image uploads in PHP is handling orientation and rotation. When you take a picture with a modern camera, the orientation in which the picture was taken is not how it's saved.

For example, if you take a picture with your iPhone camera upside down, when you open it on your computer or view it in your browser, the picture is going to be upside down. Similarly, if you take a picture in landscape mode, when you view it on your computer or in your browser, it will be displayed vertically instead of as a landscape.

We can fix this by checking the `Orientation` value inside of the EXIF metadata (before stripping it) and rotating the image accordingly. There are 8 different EXIF `Orientation` values, each one representing a different state of rotation.

1	2	3	4	5	6	7	8
888888	888888	88	88	8888888888	88	88	8888888888
88	88	88	88	88 88	88 88	88 88	88 88
8888	8888	8888	8888	88	8888888888	8888888888	88
88	88	88	88				
88	88	888888	888888				

Orientation Value Diagram

When the EXIF Orientation is 1, nothing needs to be done! The image is already correctly rotated. For all of the other values, we need to adjust accordingly. Older PHP tutorials recommend using `exec()` with the command line tool `jpegtran` and `jhead` to accomplish the rotation. You already know the drill. Using `exec()` during a web request is bad form, extra dependencies, potential security hole, DDoS vector, etc.

Instead, we can adjust the orientation with nothing besides GraphicsMagick.

```

1 <?php
2
3 $g = new GMagick("file.jpg");
4
5 // Read the Orientation from the EXIF data
6 $exif      = exif_read_data("file.jpg");
7 $position = $exif['Orientation'];
8
9 switch($position) {
10   case 2:
11     $g->flopimage();
12     break;
13
14   case 3:
15     $g->rotateimage("#ffffff", 180);
16     break;
17
18   case 4:
19     $g->rotateimage("#ffffff", 180);
20     $g->flopimage();
21     break;
22
23   case 5:
24     $g->rotateimage("#ffffff", 90);
25     $g->flopimage();
26     break;
27
28   case 6:
29     $g->rotateimage("#ffffff", 90);
30     break;
31

```

```
32  case 7:
33      $g->rotateimage("#ffffff", -90);
34      $g->flopimage();
35      break;
36
37  case 8:
38      $g->rotateimage("#ffffff", -90);
39      break;
40 }
41
42 // Thumbnail resizing or whatever you need
43 // to do.
44
45 $g->write("oriented.jpg");
```



In the example above, what's the `#ffffff` argument in the `rotateimage()` method?
GMagick requires you to pass a fill color in case it needs to do a non-right angle rotation, in which case it will need to fill in the blank areas with a color. Since we're only doing right-angle rotations, the fill color will never be used.

Case Study: Benchmarking and Load Testing Web Services

Benchmarking and load-testing is ultimately one of the most important parts of Scaling. Why? It's where the rubber meets the road. Where you can see how the changes you make in your architecture or code improve your overall performance. A way to figure out where the weak parts of your application are, and what's likely to crumble first under heavy load.



If you don't want to mess around with setting up a benchmarking environment, you can get started really quickly with a service like [Blitz¹⁴²](http://blitz.io) or [LoadImpact¹⁴³](http://loadimpact.com). These services allow you to send anywhere from 100 - 1,000,000 computer-generated visitors to your site to browse around and generate load.

In some ways, this should have been the first chapter of the book. Benchmarking isn't glamorous, but having hard numbers to compare your changes with is a great advantage instead of just going in blind and assuming that this-or-that config change will make your stack better. Plus—bosses love hard numbers. Saying, "My changes improved site response time by 18%" is much more valuable than "I changed some settings and it should be faster".

Setting up a Test Environment

If you've already launched, it's important that you do your benchmarking and load testing on separate servers from your production ones— we're going to be breaking things, really pushing the limits of the app and it's a bad idea to do that in production.

This is one area where I really recommend using pay-by-the-hour cloud hosting. With [Amazon EC2¹⁴⁴](http://aws.amazon.com), [Rackspace Cloud¹⁴⁵](http://rackspacecloud.com), [DigitalOcean¹⁴⁶](http://digitalocean.com), or [SoftLayer Cloud¹⁴⁷](http://softlayer.com) you can create an almost exact replica of your production servers for a couple of bucks. No hardware investment, no "test" servers that sit idle 99% of the time.

The famous ab (Apache Benchmark)

The most common high-level benchmarking tool has got to be the ab tool that comes with Apache. It's a pretty good tool, really great for just starting out because it's so easy to use.

¹⁴²<http://blitz.io>

¹⁴³<http://loadimpact.com>

¹⁴⁴<http://aws.amazon.com>

¹⁴⁵<http://rackspacecloud.com>

¹⁴⁶<http://digitalocean.com>

¹⁴⁷<http://softlayer.com>

The basic idea behind `ab` is that it given a HTTP URL, a number of concurrent visits, and a number of total requests, it will make lots of requests for that page from your webserver(s).

Of course, requesting the same page over-and-over again hardly represents real-world traffic, but it's a good starting point, especially if you can have it hit page that touches all parts of your stack. **Think of it as brute-force benchmarking.** Obviously, if you only use it to test static or non-database pages, you're going to get unrealistic results.



Watch out, if you're using Mac OS X, the packaged version of `ab` is buggy- you'll see the error `apr_socket_recv: Connection reset by peer (54)` 9 out of 10 times (but, it works sometimes, more often with lower levels of concurrency.) To fix the bug with homebrew:

```
brew install https://raw.github.com/Homebrew/homebrew-dupes/master/ab.rb.
```

Installing `ab` on Ubuntu is easy— `apt-get install apache2-utils`.

To get started, let's run a really small test with 10 concurrent connections and 100 requests. The `-c` flag is for the amount of concurrency (number of simultaneous requests) and the `-n` flag is for total number of requests.

```
1 > ab -c 10 -n 100 "http://192.51.100.100"
2
3 This is ApacheBench, Version 2.3 <$Revision: 655654 $>
4 Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
5 Licensed to The Apache Software Foundation, http://www.apache.org/
6
7 Benchmarking 127.0.0.1 (be patient).....done
8
9
10 Server Software:      WEBrick/1.3.1
11 Server Hostname:      127.0.0.1
12 Server Port:          9292
13
14 Document Path:        /
15 Document Length:      16571 bytes
16
17 Concurrency Level:    10
18 Time taken for tests: 4.768 seconds
19 Complete requests:   100
20 Failed requests:     70
21     (Connect: 0, Receive: 0, Length: 70, Exceptions: 0)
22 Write errors:          0
23 Keep-Alive requests:  100
24 Total transferred:   1678245 bytes
25 HTML transferred:    1657029 bytes
26 Requests per second: 20.97 [#/sec] (mean)
```

```

27 Time per request:      476.843 [ms] (mean)
28 Time per request:      47.684 [ms] (mean, across all concurrent requests)
29 Transfer rate:        343.70 [Kbytes/sec] received
30
31 Connection Times (ms)
32             min   mean[+/-sd]   median   max
33 Connect:        0   20  99.0       0    503
34 Processing:    87  422 156.2     416   1014
35 Waiting:       68  370 143.7     367    830
36 Total:         87  442 207.6     416   1517
37
38 Percentage of the requests served within a certain time (ms)
39   50%    416
40   66%    491
41   75%    526
42   80%    555
43   90%    669
44   95%    747
45   98%   1106
46   99%   1517
47   100%  1517 (longest request)

```

Let's dig into the important parts from the results:

```

1 ^Requests per second:      20.97 [#/sec] (mean)^
2 ^Time per request:        47.684 [ms] (mean, across all concurrent requests)^
3 ^Transfer rate:          343.70 [Kbytes/sec] received^

```

We see above the requests per second, the mean time per request, and the transfer rate in Kbytes. These are good metrics to track and benchmark as you improve your stack and code—the more requests and lower mean time per request, the better.



ab doesn't have many settings besides -c and -r. One useful, but overlooked, setting is -k which turns on KeepAlive requests. I really like using this simultaneously while running another instance of ab without -k on. Why? Because it allows me to test the load balancer network settings discussed in Chapter 4—how many connections stay in TIME_WAIT, etc.

Siege, a more modern ab

I find apache benchmark to be kind of limiting and outdated. A great, modern alternative is siege. You can install it with `apt-get install siege` on Ubuntu. The command line flags are similar to ab, -c sets the number of concurrent users and -t sets the number of seconds to run the test for.

```

1  siege -c50 -t10s http://127.0.0.1
2  ** SIEGE 3.0.1
3  ** Preparing 50 concurrent users for battle.
4  The server is now under siege...
5  Lifting the server siege...      done.
6
7  Transactions:          992 hits
8  Availability:          100.00 %
9  Elapsed time:            9.56 secs
10 Data transferred:        0.36 MB
11 Response time:           0.00 secs
12 Transaction rate:        103.77 trans/sec
13 Throughput:              0.04 MB/sec
14 Concurrency:              0.06
15 Successful transactions: 992
16 Failed transactions:     0
17 Longest transaction:     0.01
18 Shortest transaction:    0.00

```

Easy! 50 concurrent connections in 10 seconds. I generally like the output and format of siege better. You can clearly see the Transaction rate (number of requests per second) and other valuable metrics that you may want to track.

Another cool feature? It can hit multiple URLs in a single test to simulate real traffic. Make a file and put a bunch of URLs in it and siege will hit them all. It's an easy way to simulate real world traffic.

```

1  $ vim urls.txt
2  http://127.0.0.1/index.php
3  http://127.0.0.1/search.php
4  http://127.0.0.1/contact.php
5
6  $ siege -c50 -t10s -i -f urls.txt

```

Bees with machine guns

Sometimes you want a REALLY massive test and running a single siege process just won't cut it. [Bees with machine guns](#)¹⁴⁸ will launch a mini EC2 cluster to hammer your server(s) with as many clients as you want! The install is slightly more complex:

¹⁴⁸<https://github.com/newsapps/beeswithmachineguns>

```

1 $ apt-get install python-dev python-pip
2 $ pip install beeswithmachineguns
3
4 $ export AWS_ACCESS_KEY_ID="YOUR AWS ACCESS KEY"
5 $ export AWS_SECRET_ACCESS_KEY="YOUR AWS SECRET KEY"
6
7 # You will need to put in your private SSH Key that you use for Amazon EC2
8 # into this file. It must have the same name as your SSH Key uploaded to Amazon
9 n.
10 $ vim ~/.ssh/key_name.pem

```

Next, you'll have to go into your EC2 management console and create a security group (if you don't already have one) that allows SSH connections on port 22. I made one called `public`—it's what you have to pass to the `-g` parameter below.

After configuring the security group, `bees` is ready to bring up your test cluster. I'm going to start an attack cluster of 4 instances. The default type of instance is the micro instance (`t1.micro`) which is very cheap (\$0.020/hour). You can change the instance type with the `-t` flag.

```

1 $ bees up -s 4 -g public -k key_name -z us-east-1b
2
3 Connecting to the hive.
4 Attempting to call up 4 bees.
5 Waiting for bees to load their machine guns...
6 Bee i-c941cfe8 is ready for the attack.
7 Bee i-c841cfe9 is ready for the attack.
8 Bee i-0b5ed02a is ready for the attack.
9 Bee i-0a5ed02b is ready for the attack.
10 The swarm has assembled 4 bees.

```

Now that they're up, we're ready to attack! We'll send 100,000 HTTP requests with a concurrency of 250 simultaneous clients.

```

1 $ bees attack -n 100000 -c 250 -u http://ec2-54-204-61-43.compute-1.amazonaws.\
2 com/
3
4 Read 4 bees from the roster.
5 Connecting to the hive.
6 Assembling bees.
7 Each of 4 bees will fire 25000 rounds, 62 at a time.
8 Stinging URL so it will be cached for the attack.
9 Organizing the swarm.
10 Bee 0 is joining the swarm.
11 Bee 1 is joining the swarm.
12 Bee 2 is joining the swarm.
13 Bee 3 is joining the swarm.

```

```

14 Bee 2 is firing his machine gun. Bang bang!
15 Bee 3 is firing his machine gun. Bang bang!
16 Bee 0 is firing his machine gun. Bang bang!
17 Bee 1 is firing his machine gun. Bang bang!
18 Bee 3 lost sight of the target (connection timed out).
19 Bee 1 is out of ammo.
20 Bee 0 is out of ammo.
21 Bee 2 is out of ammo.
22 Offensive complete.
23 Target timed out without fully responding to 1 bees.
24 Complete requests: 75000
25 Requests per second: 616.110000 [#/sec] (mean)
26 Time per request: 302.431333 [ms] (mean)
27 50% response time: 2.000000 [ms] (mean)
28 90% response time: 6.333333 [ms] (mean)
29 Mission Assessment: Target crushed bee offensive.
30 The swarm is awaiting new orders.

```

Just like `ab` and `siege`, we can see the number of requests per second that we were able to serve and the average time per request.

You'll notice I lost one of my bees—the default instance, `t1.micro`, is very underpowered and its CPU gets throttled by Amazon. For a real test, I'd recommend using a larger instance type. When we're all done with the test, just run `bees down` and it'll shutdown your instances.

Sysbench

The last benchmarking tool I'll cover is `sysbench`. It's really useful for testing raw hardware (cpu, memory, disk), but also has a really nice mysql benchmarking suite built in that makes it really straightforward to benchmark mysql configurations as you change settings and test different hardware setups.

```
1 $ apt-get install sysbench
```

Next, we have to prepare our test:

```
1 $ sysbench --test=oltp --oltp-table-size=1000 --mysql-user=root --mysql-host=1\ 
2 ocalhost \
3 --mysql-password= prepare
```

You should see some output about creating the table and records. Lastly, we run the test.

```

1  $ sysbench --test=oltp --oltp-table-size=1000 --mysql-user=root --mysql-host=localhost \
2  --mysql-password= run
4
5  Doing OLTP test.
6  Running mixed OLTP test
7  Using Special distribution (12 iterations, 1 pct of values are returned in \
8  75 pct cases)
9  Using "BEGIN" for starting transactions
10 Using auto_inc on the id column
11 Maximum number of requests for OLTP test is limited to 10000
12 Threads started!
13 Done.
14
15 OLTP test statistics:
16     queries performed:
17         read: 140000
18         write: 50000
19         other: 20000
20         total: 210000
21     transactions: 10000 (110.61 per sec.)
22     deadlocks: 0 (0.00 per sec.)
23     read/write requests: 190000 (2101.54 per sec.)
24     other operations: 20000 (221.21 per sec.)
25
26 Test execution summary:
27     total time: 90.4101s
28     total number of events: 10000
29     total time taken by event execution: 90.3572
30     per-request statistics:
31         min: 4.88ms
32         avg: 9.04ms
33         max: 532.42ms
34         approx. 95 percentile: 13.68ms
35
36 Threads fairness:
37     events (avg/stddev): 10000.0000/0.00
38     execution time (avg/stddev): 90.3572/0.00

```

This output shows us various information about the performance of MySQL read queries, write queries, and number of transactions. You can tweak some settings and re-run the benchmark to see how the changes influenced the overall query speed and performance of MySQL.

The other test suites that come with sysbench are `fileio`, `cpu`, `memory`, `threads`, and `mutex`. I find that the `fileio` test is great for benchmarking different types of drives.

You may also want to check out the amazing [Phoronix Test Suite¹⁴⁹](#) for an even more comprehensive server benchmark suite.

¹⁴⁹<http://www.phoronix.com/>

Sponsors

I launched Scaling PHP Applications as an experiment in 2012— people were able to pre-purchase the book for instant access before it was finished. Thank you to all of my alpha purchasers. Without you guys, I wouldn't have been able to finish this book. Thank you for all of the feedback and support along the way, you're very bright and amazing people that have helped to shape this project.

Here's to never having to google "best mysql config settings" ever again.

I'm putting the names of all of the early purchases below as a way of saying "thank you". When I sent out my first email campaign, I was pretty nervous and only sent it to email addresses starting with the letters a-g, so you'll notice a massive disproportion of people named David and Alex in the list below :).

Cheers!

TomÃ;s Mayr	Bradley Falzon	Angel Alonso
Andrew Pile	Bradley Beebe	Arlen Yan
David R. Jones	Anil Yeni	Brandon Ooi
B Yildirim	Bobby Schuchert	Mr B M J Wigoder
Austin Butler	Andrew Drexler	William Hance
Ben Stahlhood	Jarvis Badgley	Aaron Peterson
Anand Capur	Eric Amundson	Ankur Taxali
Andy Fleming	Joel Santodomingo	David Jeffries
Eric Stern	Daniel Vidaurre	James Edward Dudley
Daniel Monaghan	David Chen	Frank Mullenger
Jason See	Devin Zhang	Daniel Stevens
Dwight Watson	Pierre Chambon	Eric DeMenthon
David Shariff	Rianov purnama	Davi T Alexandre
Adil Shakour	Henry Paradiz	Matthew Davey
Dan Miller	Adam Spooner	Dhawal Shah
David Voong	Dan Sapala	Dave Swift
Armen Baghумian	Dean Sadler	Brandon Smith
Gunnar Lium	Ankesh Kothari	David Smith
Eirik Stavem	Benjamin Peyer	Dominik Belca
Alex Munroe	Aviram Ben Moshe	Feisthammel
Danil Semelenov	Alex Gemmell	Jrgen Hrmann
David Marsn	Hugo Chinchilla	Ed Brown
Benjamin Fleischer	Goran Panic	Aaron Harder
Bastian Widmer	Darren Whitlen	Jordan Patterson
Christian Kirkegaard	Derek Harvey	Nikki Snow
Carl sutton	Bram Vogelaar	Gonzalo Payo
Alexandre Demers	Alexander Brausewetter	Drew Johnston
Bracken King	Austin Howe	Danilo Assis
Ghaus Iftikhar	Brandon Wamboldt	Andreas Smith
Bijan Vaez	Colin Hynes	Mikhail Kofman

Fbio Tadeu da Costa	Alain Russell	Matthew B Hokanson
Dylan Arnold	Audrius Aidukas	Zachary Aaron Gardner
Eric Berg	Henrik Hussfelt	Francois Baligant
Andy Kelk	Michael van Lier	Dominic Black
Arjen Schat	Christopher Wyllie	Antony Zanetti
Andrew Ryno	Chuck Reynolds	Sungjoo Ha
Feliks Beygel	Mnsoft	Maor Bolokan
Emanuele Cesena	Damon Sauer	George Eves
Adam Duren	Joffrey Jaffeux	Sean O'Shaughnessy
Seth Ryder	Gordon Williamson	Benedict Steele
Konstantin Sykulev	Jason A Ward	Forrest Jordan
Luc Gauthier	Jeff Jason II	Pamela Brown
Isern Palaus	Aditya Advani	Sebastien Lavoie
John McDowall	Jason Altekruuse	Grant McNally
Stig Bertil Ingvar	Andreas Karlsson	Thomas Prip Vestergaard
Valeriy E Trubachev	John White	Eric McCormick
Darius Aliabadi	Jake Fournier	Ronald Male
Mark Myers	Thi Nguyen	Harry Grillo
Yvonne Adams	Peter Holberton	Web Monkey Oy
Scott Mansfield	Hans Ahrens	Chris Schuld
Ole Markus With	Scott Warren	Johnny Green
Vamsi Krishna B	Yuji Yokoo	Carlos Buenosvinos
Raul Fraile Beneyto	Komang Arthayasa	Sean Hickey
Javier Montes Martinez	Devon Ostendorf	Mark Limburg
Gavin Staniforth	Jani Elo	Jonathan Kiddy
Gregory H Salmon	Chris Szymansky	Tabare Caorsi
Ben Shakal	Reinder de Vries	Fredrik Nordell
Sampsaa Saarela	David Burrows	Terry Appleby
Martin Naumann	Andrew Elster	Philip Daly
Scott Moss	David Gieger	Michael Orr
Angus G Nelson	Au-Yeung Kok Chue	Rohit Modi
Alexandru Barbulescu	Dan Fisher	Michal Marczyk