Elephant Sized PHP

Scaling up your PHP application

Andrew Beak

Elephant sized PHP

Scaling your PHP applications

Andy Beak

For Duckems, Beastly, and Bratface

For whom I would change the world if only I had the sourcecode

Contents

Introduction

This book was written because I wanted a place to centralise my research and experience. I've been blogging for a number of years and have always enjoyed the value of having a reference to go back to. If I solve a problem I make a note of how I did it so that future Andy doesn't have to waste time doing what I did. This book is very much about that, it's about putting all the bits and pieces that I've learned into one place.

This book is divided into three sections. I firstly look at software architecture ideas that help when scaling an application. Ideally an application is designed with scaling in mind, this section of the book introduces some of the design concepts.

The next section focuses on gathering information about the running state of your application. Unless you're able to properly measure performance then you can't accurately measure the effect of a change.

Lastly we look at various server configurations and conclude with a general look into load balancing which also brings together some of the other concepts we dealt with in the book.

Nothing in this book should be done to a live server until you have completely and thoroughly tested it in dev and staging.

Your mileage may vary, be sure to test and consider everything properly, and anything you do to your server is your decision. This book does not constitute advice or any form of contract.

Don't experiment on servers that matter to you, spin up a temporary EC2 instance and experiment on that!

Architecture

"Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

Lewis Carrol, *Through the Looking-Glass*

Scaling should be part of your thought when architecting your solution. Scalability is not something that can be easily added as a feature to an existing product, but rather should be an intrinsic property of your application.

We should design from the start with the expectation (or hope) that our product will become successful enough that scaling will become a concern.

I'm not going to talk to micro-optimizations within code as these are most likely going to be a false economy. Spending too much time worrying about optimizing code at the outset of your project is deadline poison.

Later in this book we will be looking at ways to profile running PHP projects. This will help you to identify small areas of code that could benefit from optimization, and you can iterate on this as features of your working product.

Rather, this section is about designing and writing PHP code that is able to scale. My rationale is that before we start looking at scaling hardware we should make sure that our code will make our job easier to do so.

Divide and Conquer

Divide and Conquer is a key concept in scaling. I can imagine a Sun Tzu quote being appropriate here, but we're not trying to kill our servers with fire.

Rather I'll state the obvious: That if we plan our application to be deployed as a number of relatively independent modules we are able to scale each component independently.

It is far easier to build redundancy into modules that are independent of each other. This helps us improve durability of our application, but also allows us to more easily spawn more of them.

N-tier Architecture

You're almost certainly familiar with the concept of n-tier architectures. This style of architecture separates your application into broad functional layers that are abstracted from each other.

Most commonly you'd expect to see a persistence layer, a business logic layer, and a presentation layer. Each of these could be further sub-divided or include multiple components.

Splitting up your application into relatively independent layers makes it a lot easier to maintain, test, and refactor.

The most common oversight that I've seen in n-tier applications I've worked on is that not enough planning was made with regard to caching. When the application became popular the decision was made to "use caching".

Caching should never be an afterthought, it should be baked into your architecture from the get-go. Even if you don't implement caching to begin with you should have a very clear plan for it.

In one case that stands out in my memory the logic tier was not properly decoupled from the persistence tier. It was pretty clear that there was an intention to create "fat models" that removed business logic from the controller in the MVC pattern that was being implemented.

However there was no attention paid to making sure that there was a clear line between persistence and data logic. It took a surprising amount of effort to go back and insert caching in the persistence tier.

Incidentally, the *repository pattern*[1] is very useful to help avoid this sort of problem in the data layer. I create very thin models that are responsible only for interacting with the persistence layer. On top

[1] https://msdn.microsoft.com/en-us/library/ff649690.aspx

of those I build a repository layer which retrieves the data (from the persistence layer or cache) and provides it to the data service layer which provides the controller with the data it needs to pass on to the view.

Microservices

The Wikipedia definition of microservices is that they are "a software architecture style in which complex applications are composed of small, independent processes communicating with each other using language-agnostic APIs".

An application can be built up out of these relatively small building blocks. The services are designed to be easily replaced and since they're operating with an API can be implemented in any language.

They can be compared to the monolithic style in which an application is built as a single unit. A monolithic application uses one process to provide all of its functionality. The microservices architecture seeks to split each element of functionality into a separate service.

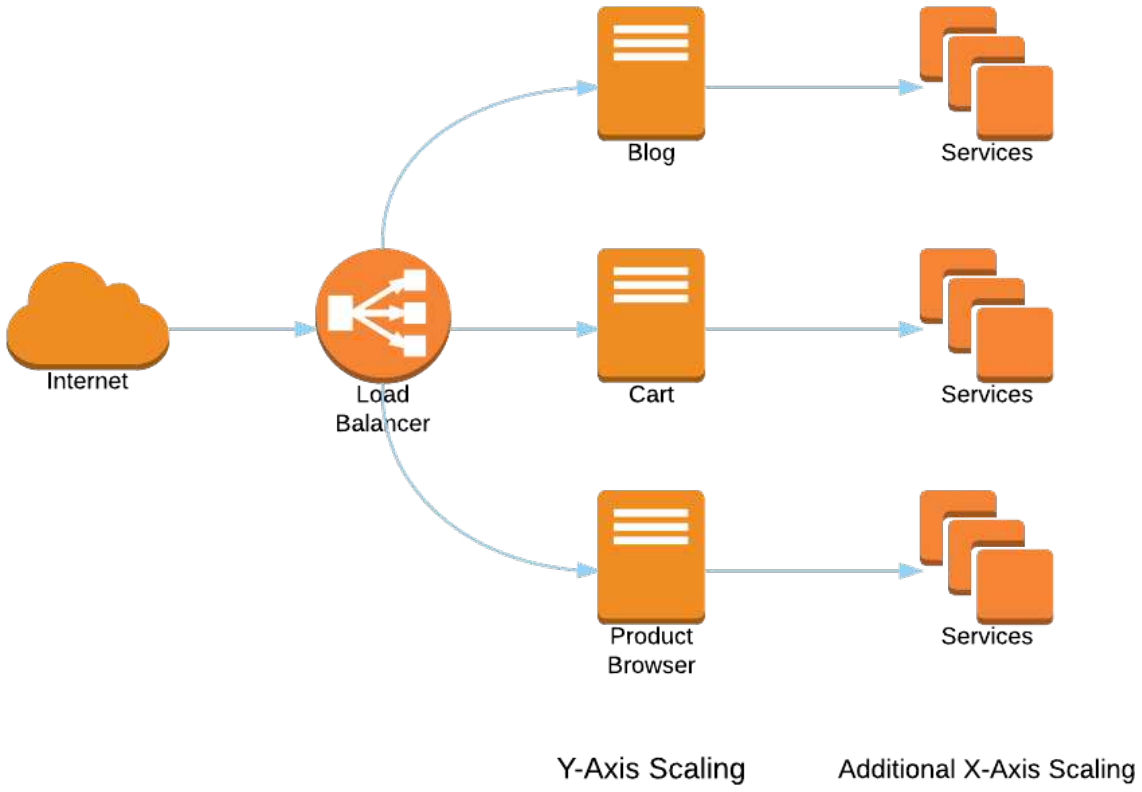
The book "The Art of Scalability[2]" introduces the idea of a scale cube where scaling occurs on three axes.

X-Scaling is horizontal scaling where we create multiple copies of the application and switch between them using a load balancer.

Y-Scaling splits the application into multiple components each of which is responsible for a set of related functions.

Z-Scaling involves having multiple copies of the application running, but each taking responsibility for a subset of the data. The database is sharded between multiple servers and a router sends requests to the server that is responsible for the data the request needs.

[2] http://amzn.to/1LiM7YH



Microservices as X-Scaling

In the diagram above we're using a load balancer to divide traffic up by its function. We then have a number of services, and replicas of these services, behind each server to support its functionality.

You can see that scaling this architecture is going to involve a lot of devops-fu.

When scaling a monolithic application you're forced to scale horizontally by creating identical copies of the whole application. Microservices allow you to use more flexible scaling approaches. This does come with the cost of complexity because you're faced with a higher volume of services you need to coordinate when scaling.

The excellent book Building Microservices[3] by Sam Newman goes into depth in discussing how to build microservices.

Services versus Libraries

Libraries are a way to encapsulate functionality into a component that can be plugged into your software. You're able to upgrade your library or replace it with another that offers the same

[3] http://amzn.to/1V23yPh

functionality.

Services are a different way to create components of software. Instead of being a library that is called from within your code they are called as external components using an API or remote procedure call.

Libraries are tightly bound to your codebase. They are deployed as part of your application and if you need to change them you'll have to rebuild and redeploy your application. Services exist as separately deployed objects and can even be written in an entirely different language to the rest of your application.

One problem with using microservices is that using an external call is more expensive than an in-process call. Networks are not always reliable and they can have latency issues. When building microservices one should be very aware of the Fallacies of distributed computing[4].

Netflix wrote their Simian Army[5] software to create failures in their infrastructure. Simian Army can create the havoc that you'd expect from an army of monkeys running through your data centre and helps Netflix test their monitoring services and their application resiliance.

Using microservices instead of libraries requires that you're able to monitor each of your services and be able to quickly respond to bring it back up if it fails. This means that real-time monitoring and automated deployment processes are very important in microservice architecture.

Dividing up software

We looked briefly at N-tier architecture earlier in this book. This is a traditional way of splitting up an application and is often based on splitting staff into teams based on their specialty skills. A company might have a frontend team, a middleware team, and a team of DBA's. Each team focuses on a tier of the application and a change that requires changes in multiple levels of the application requires teams to work together.

There is an alternative way to structure your company, which will translate into a different way to structure the software you build.

Consider an organisational structure that creates teams that are designed to solve a business capability. In companies I've worked for we've often had a team assigned to a particular product or large client. A cross-functional team of people brings all the skills needed to serve the needs of the client.

Microservices approach software based on this organisational structure - by bringing a broad-stack implementation of a solution to a business requirement.

Microservices versus SOA

Microservices at first blush sound a lot like Service Orientated Architecture. The chief difference between the two approaches is that SOA focuses a lot more on the enterprise state bus as a way to

[4] https://blogs.oracle.com/jag/resource/Fallacies.html

[5] https://github.com/Netflix/SimianArmy

control the messaging between monolithic systems.

Microservices focus on using very simple messaging and keeping complexity contained in the service. This is in stark contrast to SOA architectures which define complicated messaging protocols.

Shared Sessions

By default PHP stores session information in files on disk. When we're scaling up an application we're going to be using multiple servers which are not going to be able read files on each others disks.

The problem is that if a person makes a request to a server that cannot read their session then to that server the person is a fresh visitor.

We could consider using the cookie to store session data. The users browser would supply all the session information to the server. This is a Bad Idea because cookies exist outside of our control, which means that they can be read by other applications, tampered with, or just deleted.

Our load balancer may have a feature like session affinity whereby it takes the responsibility of remembering which server a user session is stored on. Of course this places restrictions on your load balancer on how it can distribute the load between servers.

The final way of sharing sessions is to use a common session store that each server can access. We want something fast with low latency that servers freshly added to our load balancing pools can reach with minimal configuration.

Using a database as a shared session storage might seem appealing, but this will mean hitting the database for every request. This places greater load onto the database.

A better alternative for storing session data is Memcached or Redis, both of which are easy to set up and offer lightning fast key-value storage.

The chief difference in the context of sessions is that Redis offers persistent storage. Memcached is a little faster but if your server restarts then your sessions are lost.

To use either of these as session storage you'll need to install the PHP extensions. Memcached comes in the package php5-memcached and Redis in phpredis[6].

Make sure that you don't use the deprecated memcache extension and rather choose the newer memcached extension (note the d).

Once you've installed the extension they make available the option to change your session storage location. You can do so either in your PHP config file[7] or by calling ini_set("session.save_handler",

[6] https://github.com/phpredis/phpredis

[7] https://secure.php.net/manual/en/session.configuration.php#ini.session.save-handler

"memcached"). Setting it in code is slightly preferable as changing it in your config will affect all PHP running on your server.

If you want to roll your own session storage the PHP function session_set_save_handler[8] lets you specify a custom method to handle how your sessions are stored. The class that you use must implement the SessionHandlerInterface[9]. There are lots of sample classes on Github[10] that you can use as inspiration. I personally wouldn't consider using my own class when Memcached and Redis are so easy to get working.

Logging

If we distribute the load among our servers it is useful to centralise logging. Gathering logs into a central place allows us to search them and also helps us retain logs even after the instance in our server pool has been decommissioned.

Services like Loggly[11] and Splunk[12] (autocorrect is not your friend!) offer easy to use log management. I haven't used Splunk because I was reluctant to install the agent software needed to gather logs. Loggly has been really great to work with and also offers a browser extension that lets you link it to New Relic.

There are open source alternatives like Graylog[13] that have professional support packages that work out costing a lot less than you'll pay with commercial offerings.

[8] https://secure.php.net/manual/en/function.session-set-save-handler.php
[9] https://secure.php.net/manual/en/class.sessionhandlerinterface.php
[10] https://github.com/search?utf8=%E2%9C%93&q=php+session+handler
[11] https://www.loggly.com/
[12] http://www.splunk.com/
[13] https://www.graylog.org/

Asynchronous Workers

Some tasks, like sending an email, can take time to run but your user should not have to wait for them. These tasks should rather be processed by a separate worker that runs in the background. This allows your application to continue running without needing to wait.

When architecting your solution take a look and try to identify jobs which could be performed asynchronously. Usually the first place to look is for any sort of network request. If the network request times out then your user would be stuck waiting for 30 seconds or however long your timeout is set to.

Because your code is being run by a worker process you're able to completely decouple your job from your application code. This has benefits for software design and also allows you to scale your workers independently of your core application.

If a worker processing a job fails then your application is not affected which improves your app's resilience.

Queues

The best way to plan for including asynchronous workers in your application is to use a message queue. Message queues are scalable, durable, and can reliably deliver jobs to worker processors.

Queues provide the ability for your application to asynchronously process jobs. You can queue something up now, and process it later in a different worker process.

Databases are not Queues

Using a database to simulate a queue might be tempting but it is actually going to be more effort than using a dedicated queue. Consider a table like this:

id	Job	Status	Num Retries
1	Retrieve Twitter posts	Done	1
2	Send confirmation email	Done	1
3	Watermark S3 images	Waiting	2
4	Call data enrichment API	In progress	0

Our plan is to search the table for jobs that are waiting to be processed. We then update the database to mark them as "in progress" and start working on them. If a job finishes successfully we mark it as "done" or otherwise return it to "waiting" status while incrementing the number of retries.

Our worker can include logic to manage the maximum number of retries a job can have before being marked as failed permanently.

We'll need to build in row locking logic if we wanted to have more than one worker. Consider the case where we have five workers looking at the table. If a worker takes a job off the queue but doesn't update the status before another worker queries the table then both workers will process the job. To prevent this we'll need to make sure that we use database locks in our workers.

We haven't even begun to consider some of the other features that queues provide and we already have quite a lot of coding overhead.

Amazon SQS

Amazon Simple Queue System offers a 100% guarantee that a message it accepts will be delivered to a worker at least once. It scales automatically and can handle firehose volumes of messages. It's also trivially easy to set up monitoring on your queue that will automatically spin up new worker instances if a backlog is building up. In my opinion the ability to autoscale by adding more workers is the killer feature for cloud based message queues.

Setting up SQS

To setup SQS we'll follow these steps:

1. Create an IAM role to allow EC2 instances to access SQS
2. Spin up an EC2 instance and assign this role to it
3. Include the AWS SDK and connect to SQS

Create IAM role

Login to your Amazon dashboard and navigate to your IAM console. Add a new role and select "Amazon EC2" as the service role. This allows EC2 instances to call AWS services on your behalf. Attach the policy to allow full access to SQS to the role.

You should never store your AWS credentials in your application. Rather use Amazon's IAM roles to allow your worker instances access to the queue.

Your review screen should look something like this:

Example SQS setup

Spin up EC2 instance

You can't assign a role to an existing instance; you can only specify a role when you launch a new instance so spin up a new EC2 instance to experiment with. On step 3 where you configure the instance details make sure that you assign it to the role you just created.

Open up the Amazon SQS console and create a new queue. Leave the default settings for now, we'll be looking at queue options a little later.

Use the SDK to connect to SQS

We'll need the Queue URL, once you have it include the PHP AWS SDK[14] into your project with composer. My composer file for the example PHP code below looks like this:

```
{
    "require": {
        "php": ">=5.5.9",
        "aws/aws-sdk-php" : "*"
    }
}
```

You're now able to create instances of the SQS client in your code like this example:

[14] https://github.com/aws/aws-sdk-php

```php
<?php
require('vendor/autoload.php');

use Aws\Sqs\SqsClient;

$params = ['region' => 'eu-west-1', 'version' => '2012-11-05'];

$sqsClient = SqsClient::factory($params);

$sqsClient->sendMessage(array(
    'QueueUrl'    => 'https://my_sqs_queue_url',
    'MessageBody' => 'Test Message',
));
```

Note that my credentials are not in my code, my instance is given permission to access the queue and so my PHP code does not need to authenticate itself.

Queue Settings

Amazon SQS offers a number of settings that can be tweaked.

Setting	Default	Used for
Default Visibility Timeout	45 seconds	The length of time (in seconds) that a message received from a queue will be invisible to other receiving components.
Message Retention Period	14 days	The amount of time that Amazon SQS will retain a message if it does not get deleted.
Maximum Message Size	256KB	Maximum message size (in bytes) accepted by Amazon SQS.
Delivery Delay	0 seconds	The amount of time to delay the first delivery of all messages added to this queue.
Receive Message Wait Time	0 seconds	The maximum amount of time that a long polling receive call will wait for a message to become available before returning an empty response.

The default visibility timeout setting is used to hide jobs that have been accepted by a worker from other workers. You don't want a job to be run more than once. Once a worker has pulled the job off the queue it is no longer available to other workers until the time in this setting has passed.

It should be set to be longer than the time you expect a worker to take to process the job. If a worker

that took the job off the queue is still busy on the job and the default visibility timeout expires then the job becomes eligible for another worker to pick it up, despite the fact that it is already running.

We do not typically use long polling processes in PHP workers. PHP is not designed to be a long running process and typically we spawn workers using a cron job or in response to other events, such as a notification service.

Dead letters

Messages on a queue that cannot be delivered can be placed onto a dead letter queue[15]. This is supported by SQS, RabbitMQ, Beanstalk, MSMQ, and other queue providers.

Storing the dead messages on a separate queue allows you to examine them to try and establish why they failed. Of course the real advantage is that your workers will no longer keep wasting compute resources by pulling the job from the queue.

To set this up on SQS you first add a new queue to hold the dead letters. Open up the SQS console, and click "Create New Queue". Name your queue something like "dead-letter" and save it.

You should be returned to the list of queues for the current region in your account. Your new queue should appear in the list, but select the queue you want dead-letter support for and select "Configure Queue" from the "Queue Actions" button.

In the popup window:

1. check the "Use Redrive Policy" box,
2. provide the name of the queue you just created, and
3. specify the number of times that a message can be picked up by a worker and returned to the queue before it is considered failed.

Now messages that workers fail to process will be moved to the dead letter queue.

Autoscaling PHP Workers in AWS Elastic Beanstalk

A huge advantage of using cloud services is the ability to provision server resources on the fly. AWS, Azure, and Google Cloud all allow you to spin up worker instances in response to events such as a backlog of queue items.

We'll have a look at how to set up an autoscaling group of workers that can connect to our SQS queue. I'm assuming that you have the role and the SQS queue set up from the preceding section.

In order to accomplish this we will:

1. Install the Amazon Elastic Beanstalk CLI tool
2. Pull the example from Github
3. Deploy it to Elastic Beanstalk

[15] https://en.wikipedia.org/wiki/Dead_letter_queue

Installing Amazon Elastic Beanstalk CLI

You can follow the instructions on the Amazon site[16] but if you're on Ubuntu all you need to do is:

```
apt-get install python-pip
pip install --upgrade awsebcli
```

You'll need to provide the CLI with authentication details. You can add these in the file at `~/.aws/config` (see the Amazon manual[17] for details). Your config file should look something like this:

```
[default]
aws_access_key_id=your_user_key
aws_secret_access_key=you_user_access_key
```

Pull the example from Github

I've created an example project based on Lumen, the micro-framework from the Laravel guys. You can clone it into a directory with Git:

```
git clone git@github.com:andybeak/sqs-worker.git
```

You should not need to run composer to install the dependencies. Elastic Beanstalk will do so for you when you deploy.

Deploy with Beanstalk

You first need to set up an application in Elastic Beanstalk. We do so by like so:

```
eb init sqs-worker-application
```

If you were logged into your Amazon web console you would be able to see this application listed in your Elastic Beanstalk section.

Now that we have an application we want to create an environment. For purposes of demonstration we're just going to create a production environment, but you can just as easily use different environments for different Git branches.

We're in the master branch of our Git repo and we want to associate this with our production stack.

Use the following command:

[16] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-install.html

[17] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-configuration.html

```
eb create sqs-worker-prod --single --branch_default --tier worker
```

In production you will likely not use the `--single` flag in which case Elastic Beanstalk will deploy an autoscaling load balanced group. You may also need to specify the VPC and subnet(s) to use if you have removed your account default. You'll find information on these and other parameters in the manual for eb create[18].

This means that your command may end up including the following flags:

```
--vpc.id=vpc-f61a9493 --vpc.ec2subnets=subnet-51711326 --vpc.publicip
```

Elastic Beanstalk will automatically create a security group to apply to your instances. If you selected to setup SSH when creating the application port 22 will be open in this group. You can add the instances to your own security groups by giving a comma separated list on the command line (see the manual).

After you've got your environment created you'll see it in your Elastic Beanstalk web console. If you click it you'll see more details, including a log of the various actions that have been taken on your behalf.

When Elastic Beanstalk set up the environment it automatically created a new SQS queue for you. It also set up CloudWatch metrics and triggers for the autoscaling.

All that you need to do now is use `eb deploy`. This will use git to zip up the project and then deploy it to the EC2 instances in the load balancing group.

Terminating your environments will tear down all of the infrastructure they created, as indeed will deleting the application.

Other Queue Providers

IronMQ, RabbitMQ and Beanstalkd are all strong contenders for your attention in the PHP world. They have well supported libraries and if you plan your architecture carefully you should be able to swap out a Queue provider without impacting your entire application.

The first big question to ask yourself is whether you want to self-host a queue (RabbitMQ[19] or Beanstalkd[20] or use a provider (IronMQ or SQS). There are pros and cons for both use cases.

There are costs associated with using a provided service, but whether these costs are greater than the cost of your engineers manually installing and maintaining the software is for you to decide.

[18] https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb3-create.html

[19] https://www.rabbitmq.com/download.html

[20] https://kr.github.io/beanstalkd/download.html

MySQL design optimisation

Designing your MySQL schema carefully can help improve your database performance. Not all data types are equal and not every column should be indexed. This chapter takes a look at the MySQL optimisations that are possible, but it's important to note that other engines may not have similar performance profiles.

The different storage engines in MySQL behave differently as well. The treatment of strings in MyISAM and InnoDB is quite different for example. You should make sure you know which storage engine your table is using.

Choosing Data Types

In general smaller data types are usually better. Smaller data types take less space in memory and disk and can be retrieved faster. They use less space in the CPU cache and often can be processed quicker.

Similarly, using simple data types is more efficient than complex. Integers are computationally cheaper to compare than characters. If MySQL has a built-in type (like `DATETIME`) then it will invariably be better than using characters.

MySQL offers a number of aliases for data types in order to maintain compatibility with the SQL standard. `BOOL`, for example, is an alias of `TINYINT(1)` and is not its own native datatype in MySQL. Using an alias does not impact performance.

Integers in MySQL can have a width specified which manages how MySQL internal tools display the value. `INT(1)` is identical to `INT(11)` in storage and computation, and the only difference between them is how MySQL will display them.

`FLOAT` and `DOUBLE` are used to support *approximate* calculations using standard floating point arithmetic. You can use `DECIMAL` for precise calculations. The implication is that `FLOAT` and `DOUBLE` will be faster to perform at the cost of accuracy.

The most common use of `DECIMAL` I've seen is for currency calculations. It may be faster to use `BIGINT` and store your data multiplied by a factor of ten that represents your desired level of precision. For example if you require three digits of financial precision you would store $ 123.456 as a BIGINT with value 123456. This helps to avoid the tradeoff between speed and accuracy.

`VARCHAR` saves space and so improves performance, but it can result in extra work for the database to manage storing them on disk when they change size. A `VARCHAR` column will use an extra byte

to store the length of the string. It uses two bytes to store the length if the string is longer than 255 bytes.

When using `VARCHAR` it is still best to try and restrict the maximum length of the field. Don't allocate 255 characters if you are certain a persons first name won't be longer than 30 characters. MySQL will often use the width of the column as a hint when allocating memory for internal operations and having columns that are too big wastes memory at these times.

You can make `JOIN`s more efficient by using the same data types to store related values used in a join. This reduces the internal load on MySQL when computing the join.

Where possible try to avoid allowing NULL values. NULL values interfere with the selectivity of indexes, something we'll be looking at later in this chapter.

Schema Design

Normalizing your data so that a fact is stored only once makes it faster to update your database. Because you're not duplicating information your storage footprint on disk and in memory is reduced, which improves performance.

The drawback to a normalised schema is that you'll often need to join tables together. You're not able to index across tables and so it may be impossible to place some columns into the same index even if they belong together.

Denormalized tables can be a lot quicker to read from. Because you're avoiding a join you're not going to be doing random I/O to seek into points of the joined table. Instead you'll be doing a sequential read of the table and using that to operate on.

Summary Tables

One way to denomalize data is to create a special table to store it in. Such a "summary" table may store pre-calculated aggregates of data that you need.

Whenever you need an aggregate you can perform a quick select on this small table, instead of performing the much more expensive `COUNT` or `GROUP` queries on your source table which may contain millions of rows.

If you can tolerate stale summaries you can calculate the data for the table on a regular basis. Otherwise you would need to update it whenever you update the source tables.

Lets consider a large eCommerce store with millions of products that wants to display the most popular selling items in a category. It is quite computationally expensive to make this calculation when the user visits a webpage so instead they create a separate table that contains just this information, such as this example:

id	Category	Product	Last Updated
1	Electronics	Fitbit	2016-03-06 11:53:02
2	Books	Zen Programming	2016-03-06 11:45:23
3	Mens Apparel	Plain white tee	2016-03-06 12:03:31
4	Women's Apparel	Orange Summer Dress	2016-03-06 11:42:38

Now when a user lands on the site and we want to display popular items we don't need to do any calculations. We can offload those calculations to Amazon spot instances that prepare the data independently of our page load. This approach performs a full recalculation of the summary data and we would want to do this only periodically.

A different summary table could contain, for example, the total number of items in each category and is grouped by sub-category. When we add an item to a sub-category we would at the same time update the summary table. We take the hit for write performance associated with denormalizing but will enjoy much quicker reads when we want to display summaries.

Indexing

Indexes are data structures that are implemented in the storage engines to help it locate rows quickly.

Each engine may have its own characteristics or implementation of indexes. The following table shows the index types available for each engine:

Storage Engine	Permissible Index Types
InnoDB	BTREE
MyISAM	BTREE
MEMORY / HEAP	HASH, BTREE

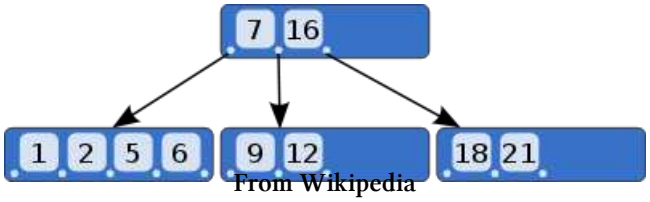
Indexes reduce the amount of data that the server has to examine and turn random I/O into sequential I/O. They also help the server to avoid having to use sorting and temporary tables.

B-Tree indexes

MySQL most commonly uses B-Tree[21] indexes. A B-tree works by defining pointers that indicate where ranges of a value may be found.

This example, from Wikipedia, shows that values less than 7 are contained in the block pointed to on the left. Values greater than 16 are pointed to the right hand block. Values between the two are contained in the middle block.

[21] https://en.wikipedia.org/wiki/B-tree



From Wikipedia

When looking for a value the storage engine starts at the root node which contains pointers to leaf nodes. The engine examines the value it is looking for and follows the pointer to the next block. The next block will either contain a further sub-division of values pointing to blocks, or a pointer directly to where the data are stored.

You should not index your table without knowing what your queries are, or being willing to refactor some of your queries.

Composite B-Tree Indexes

Indexes may contain one or more columns. An index that includes more than one column is known as a *composite index*.

A single index with two columns in it behaves differently from the two columns each having their own index. For example, when you use a `SELECT` statement to find rows and specify all the columns that exist in a composite key then the rows can be fetched directly by the engine. If each column were independently indexed then MySQL would have to merge the keys.

Here is the employees table from the MySQL sample database[22]:

Employees table

We'll add a composite index (one which has more than one column) as follows:

```
CREATE INDEX `example`
        ON `employees`
(first_name, last_name, hire_date)
USING BTREE;
```

[22] https://dev.mysql.com/doc/employee/en/

Note the order in which we're adding columns to the key. This will be important when we write our queries to take advantage of the key.

It's instructional to refer to the MySQL manual[23] which explains how indexes are used.

We can see from the manual on composite indexes[24] that because we've used first_name as the leftmost index then the index will be used for queries that

For example this query will use the index:

```
SELECT * FROM employees
  WHERE first_name='Wikus' AND last_name='Van Der Merwe';
```

But it won't be used for this query because the OR condition does not include the leftmost column in the index:

```
SELECT * FROM employees
          WHERE first_name='Wikus' OR last_name='Van Der Merwe';
```

The index won't be used in the following query because we've skipped a column in the key:

```
SELECT * FROM employees
          WHERE first_name='Wikus' AND hire_date >= '2016-01-01';
```

Hash Indexes

Hash indexes are formed by calculating the hash of the columns in the index. They are only available in tables backed by the memory storage engine, but you can create table structures yourself to emulate them in other storage engines.

The MySQL manual[25] has a page dedicated to understanding the difference between them and B-Tree indexes but we'll go through them here too.

The hash index structure arranges the computed hashes in order. This makes it quick to look up a hash value. The hash value points to a row in the indexed table. This means that when MySQL uses the index to find a row it reads the index and then has to read the table to get the values. This is in contrast to the B-Tree index where the final leaf node points directly to the data.

Because the index is stored ordered by hash you cannot use it to assist with sorting the table by row. You can also only use them with equality operators that test for direct matches. In other words you can only use them for =, IN, and the spaceship operator <=>. You cannot use hash indexes for testing greater than or less than.

[23] https://dev.mysql.com/doc/refman/5.5/en/mysql-indexes.html
[24] https://dev.mysql.com/doc/refman/5.5/en/multiple-column-indexes.html
[25] https://dev.mysql.com/doc/refman/5.5/en/index-btree-hash.html

Emulating hash indexes

A very big advantage to hash indexes is the fact that the hash size stays the same no matter how big the values you're indexing becomes. A B-Tree index can become very big if it is used to index a long string, but hash indexes will remain the same size.

You can emulate a hash index in storage engines that don't directly support it. This involves creating a column to hold the hash value and then indexing that column with B-Tree. The hash column will be of a reasonably small size and the B-Tree index size will be manageable.

For example lets consider a rather artificial example where we want to search a table for a particular url:

```
CREATE TABLE IF NOT EXISTS `homepage` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(20) NOT NULL,
  `url` varchar(255) NOT NULL,
  `hashindex` char(10) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `id` (`id`),
  KEY `hashindex` (`hashindex`),
  KEY `id_2` (`id`)
) ENGINE=InnoDB  DEFAULT CHARSET=latin1 AUTO_INCREMENT=2 ;

DROP TRIGGER IF EXISTS `pseudohash_crc_insert`;
DELIMITER //
CREATE TRIGGER `pseudohash_crc_insert` BEFORE INSERT ON `homepage`
 FOR EACH ROW BEGIN SET NEW.hashindex=crc32(NEW.url);
        END
//
DELIMITER ;
DROP TRIGGER IF EXISTS `pseudohash_crc_update`;
DELIMITER //
CREATE TRIGGER `pseudohash_crc_update` BEFORE UPDATE ON `homepage`
 FOR EACH ROW BEGIN SET NEW.hashindex=crc32(NEW.url);
        END
//
DELIMITER ;
```

We create the `hashindex` column to contain MD5 hashes and add a B-Tree index to that column. We add triggers to the table so that whenever we insert or update a row MySQL automatically sets the hashindex column correctly.

You can use the `CONCAT` function if you want to use multiple columns in the hash.

Now when we insert a row MySQL automatically calculates and indexes the hash.

```
INSERT INTO `employees`.`homepage` (`id`, `name`, `url`, `hashindex`) VALUES (NU\
LL, 'trigger_manual', 'https://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.ht\
ml', '');
```

When we want to search for a URL we manually include the indexed hash column. MySQL will notice that there is a specific index for our query and use that to retrieve the row:

```
EXPLAIN SELECT id
FROM homepage
WHERE url =  "https://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.html"
AND hashindex = CRC32(  "https://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.\
html" )
```

This `EXPLAIN` will show that we're using the index defined on the hashindex table.

We're using the CRC-32 hash because it provides a shorter string than MD5. This reduces the size of the B-Tree index needed to index the hash column.

Indexing Strategies

We've seen that it is important to craft indexes and queries together. Lets look at ways to create and use indexes in order to benefit as much as possible from them.

Avoid expressions and functions

MySQL cannot use an index if the column is part of an expression or function.

For example lets imagine that our employees table from the example database is indexed on hire_-date. We want to search our employees table for people who were hired more than 25 years ago.

This `EXPLAIN` will show that the query will not use the index because it is part of a function:

```
EXPLAIN SELECT *
FROM  `employees`
WHERE ADDDATE(hire_date, INTERVAL 25 YEAR) <= CURDATE()
```

This query, however, will use the index because the column the index is created on is not part of a function or expression:

```
SELECT *
FROM  `employees`
WHERE hire_date < SUBDATE(CURDATE() , INTERVAL 25 YEAR)
```

Partial Indexes

Indexing very long character columns can result in bloated indexes. What if you were to index just the first X number of characters in each string? Your indexes would take less space and MySQL would still benefit from being able to filter out rows.

The selectivity of an index is a measure of how much it helps in filtering out rows. It is defined as the ratio of unique index values to the total number of rows in the table. A unique index on a table that does not allow NULL values will have the selectivity of 1, which is the highest value this ratio can have.

An index on just the first X number of characters will have a lower selectivity ratio than 1. As we include more characters our selectivity increases but so does our index size. We need to find a way to measure this tradeoff and make a decision about how many characters to index.

I'm going to use a database of UK postcodes that includes city names. You can download a copy of the database from here[26]. It's a little old but we're only using it to demonstrate indexes and don't need up to date information about the postcodes.

We can construct a query that calculates the selectivity of using a full column and increasing numbers of characters, like so:

```
SELECT COUNT( DISTINCT region ) / COUNT( * ) AS whole_column,
COUNT( DISTINCT LEFT( region, 3 ) ) / COUNT( * ) AS sel3,
COUNT( DISTINCT LEFT( region, 4 ) ) / COUNT( * ) AS sel4,
COUNT( DISTINCT LEFT( region, 5 ) ) / COUNT( * ) AS sel5,
COUNT( DISTINCT LEFT( region, 6 ) ) / COUNT( * ) AS sel6,
COUNT( DISTINCT LEFT( region, 7 ) ) / COUNT( * ) AS sel7
FROM uk_postcode_05
```

This gives a result like the table below. I've included the difference as the second row in the table so that we can see the effect of adding an extra character to the prefix.

	sel3	sel4	sel5	sel6	sel7	whole_column
Selectivity	0.0523	0.0579	0.0586	0.0639	0.0678	0.0734
Change in Selectivity		0.0056	0.0007	0.0053	0.0039	0.0056

[26] http://www.dangibbs.co.uk/journal/free-uk-postcode-towns-counties-database#download

It seems that there is a big difference between a 5 and a 6 digit prefix, but that choosing a 5 digit prefix does not offer much gain over a 4 digit prefix. So initially we would look at investigating the 4 digit and 6 digit prefixes with the next test.

We can't just look at the average selectivity and must also look at how the data are distributed. Consider this query:

```
SELECT COUNT( * ) AS cnt, LEFT( region, 4 ) AS reg
FROM uk_postcode_05
GROUP BY reg
ORDER BY cnt DESC;
```

The first five results look like this:

Count	Region
255	Grea
141	Nort
81	High
75	East
66	Kent

This demonstrates that rows that are in places that begin with "Grea" or "Nort" will have much lower selectivity than the rest of the rows. However it's pretty obvious that "Nort" is the first four characters of "North" so if we choose to include 6 characters we'll flatten the selectivity distribution somewhat.

Now that we've decided on how many characters to index we create the index like this:

```
ALTER TABLE uk_postcode_05 ADD KEY (region(6));
```

Choose column order carefully

When building a composite index the order of the columns in the index is vitally important. A multicolumn B-Tree index will sort the rows first by the leftmost column, then by the next column, and so on.

As a general rule of thumb you should place the most selective column leftmost in the column order. To calculate the cardinality of columns you can use a query like this:

```
SELECT
COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
COUNT(*)
FROM payment
```

This can give you a starting point to consider how to order columns in your index.

Covering indexes

MySQL is able to use an index to retrieve the data that is indexed directly. It doesn't have to go back to the original row because the index points directly to the data. If your index covers all of the data that a query needs then it's called a *covering index*.

Index entries are smaller than the whole row so MySQL needs to process less data than if it had to read the entire row. It's a lot less expensive for MySQL to read data from an index than to look up the rows.

You can see when you're using an index-covered query by using `EXPLAIN`. If you see "Using index" in the `Extra` column you're using a covering index.

MySQL version 5.6 and above includes a feature called Index Condition Pushdown[27]. This feature checks if parts of the WHERE condition can be covered by the index and if so it pushes this condition down the storage engine. The engine then uses the index to determine whether to read and return the row.

[27] https://dev.mysql.com/doc/refman/5.6/en/index-condition-pushdown-optimization.html

Redis and Memcached

Redis and Memcached are both awesome products. Redis has some nifty additional features but is slightly slower than Memcached. Both of them are supported by Amazon ElastiCache and both are very easy to use with PHP.

Memcache versus Memcached

PHP has two extensions to work with Memcached - one is called *Memcache* and the other is called *Memcached*. This sometimes leads to confusion.

The actual program that provides the cache is called Memcached[28]

In Linux placing a d at the end of the name usually means it is a daemon, a background running task.

The *Memcache* (no d) extension is older and you should rather use the *Memcached* extension. So within PHP you should always be referencing the "Memcached" class.

The executive summary is that "Memcache" refers to an old PHP extension and "Memcached" is both the name of the cache program and the name of the improved PHP extension.

What do they do?

Both of them offer key-value storage that your application can rapidly read and write. You can cache things that take time to generate, like database queries or network lookups like API results.

For example if you show a list of Tweets on your page you don't want to make a network call to Twitter for every single visitor on your page. Rather you can say that the account you're following only tweets every five minutes on average (for example), so you can store the results of fetching the tweets for anywhere up to that long. Now when a visitor comes to your site they don't have to wait while you make a network connection and fetch the data from Twitter.

They are also excellent locations for session storage.

[28] http://memcached.org/

What do they do differently?

Redis is able to persist to disk, which some people say makes it a better choice for session storage. It also reduces the cache warmup time for your application if your cache persists, which can be very important at scale.

Memcached lets you store values that are up to 1KB long. If you want to have larger values stored you need to come up with a strategy to split your data into multiple keys. Redis supports values up to 2GB.

Redis supports a wider range of types of values that it can store. Where Memcached only lets you store strings and integers, Redis additionally lets you store binary-safe strings, lists of binary-safe strings, sets of binary-safe strings and sorted sets.

A Redis cluster can support Master/Slave replication but Memcached does not have this ability. Memcached can operate on multiple servers, but they do not communicate with each other and so they're much more of a farm than a cluster.

Memcached is threaded whereas Redis uses a mostly single threaded design. All Redis requests are served from a single thread using multiplexing.

Redis has native support for the pub/sub communication model which offers some interesting opportunities for PHP.

Installing Redis and Memcached

Both of these packages are very easy to install and are included in most distro repositories.

Remember Memcached doesn't require authentication so protect it from the public Internet.

Redis cluster was released with version 3.0.0 in 2015 and introduces the ability to automatically shard your data across several servers. This also improves your availability since your cluster can continue to run if some of the cluster servers are unavailable.

Each server in the cluster can be set up as a master and have slave nodes attached to it.

The quickest way to get a cluster up and running to play with is to use the create-cluster script that ships with Redis. You'll find Predis[29] a feature complete PHP library on Github.

Using a cache

In its most simple form of caching you can store the results of a database query. Instead of hitting your database for a list of popular users you can have that information readily available in your cache.

[29] https://github.com/nrk/predis

This was the approach that I tried when I first tried caching and it did really help a lot. The problem that quickly becomes apparent though is that changes to the database need to result in the cache becoming invalidated.

Expiring old cache data

I use Laravel which luckily allows the use of tagging cache entries as a way of grouping them. I started the habit of tagging my cache keys with the name of the model, then whenever I update the model I invalidate the tag, which clears out all the related keys for that model.

In the absence of the ability to tag keys the next best approach to managing cache invalidation is to use *key-based expiration*.

The idea behind key-based expiration is to change your key name by adding a timestamp. You store the timestamp separately and fetch it whenever you want to fetch the key.

If you change the value in the key then the timestamp changes. This means the key name changes and so does the stored timestamp. You'll always be able to get the most recent value, and Memcached or Redis will handle expiring the old key names.

The practical effect of this strategy is that you must change your model to update the stored timestamp whenever you change the cache. You also have to retrieve the current timestamp whenever you want to get something out of the cache.

Dogpile locks

A "dogpile" lock is one which allows a single thread to generate an expensive resource while other threads use the stale value, until the updated value is ready.

The metaphore[30] package is a good example of implementing a semaphore lock to prevent dogpiling. The basic logic around this approach is as follows:

1. Retrieve the value from cache
2. If it is not expired serve it up
3. If it is expired try to get a lock for updating it
4. If you get the lock, generate the expensive resource and cache it
5. If you don't get the lock then serve the stale data

You might end up serving a small portion of slightly stale data, but you will prevent load spikes on your servers while every single process performs the expensive operation.

Russian Doll caching

[30] https://github.com/sobstel/metaphore

This technique was developed in the Ruby community and is a great way to approach caching partial views. In Ruby rendering views is more expensive than PHP, but this technique is worth understanding as it could be applied to data models and not just views.

In the Ruby world Russian Doll caching is synonymous with key-based expiration caching. I think it's useful to rather view the approach as being the blend of two ideas. That's why I introduce key-based expiration separately.

Personally I think Russian Dolls are a bit of a counter-intuitive analogy. Real life Russian Dolls each contain one additional doll, but the power of this technique rests on the fact that "dolls" can contain many other "dolls". I think that calling it tree caching would be more accurate, but obviously less able to be branded as strongly as Russian Dolls.

Nested view fragments, nested cache structure

Typically a page is rendered as a template which is filled out with a view. Blocks are inserted into the view as partial views, and these can be nested.

The idea behind Russian Doll caching is to cache each nested part of the page. We use cache keys that mimic the frontend nesting.

If a view fragment cache key is invalidated then all of the wrapping items keys are also invalidated. We'll look at how to implement this in a moment.

The wrapping items are invalidated, so will need to be rendered, but the *other* nested fragments that have not changed still remain in the cache and can be reused. This means that only part of the page will need to be rendered from scratch.

I find the easiest way to think about it is to say that if a child node is invalidated then its siblings and their children are not affected. When the parent is regenerated those sibling nodes do not need to be rendered again.

Implementing automatically busting containing layers

We can see that the magic of Russian Doll caching lies in the ability to bust the caches of the wrapping layers. We'll use key-based expiration together with another refinement to implement this.

The actual implementation is non-trivial and you'll be needing to write your own helper class. There are Github projects like Corollarium[31] which implement Russian Doll caching for you.

In any case lets outline the requirements.

Lets have a two level cache, for simplicity, that looks like this:

[31] https://github.com/Corollarium/cachearium

```
Parent (version 1)
        - Child (version 1)
        - Child (version 1)
        - Child (version 1)
```

I've created a basic two tier cache where every item is at version 1, freshly generated. Expanding this to multiple tiers requires being able to let children nodes act as parents, but while I'm busy talking through this example lets constrain ourselves to just having one parent and multiple children.

First lets define our storage requirements.

We want keys to be automatically invalidated when they are updated and key-based expiration is the most convenient way to accomplish this.

This means that we'll have a value stored for each of them that holds the most recent value. Currently all of these values are "version 1".

In addition to storing the current version of each key we will also need to store and maintain a list of dependencies for the key. These are cache items which the key is built from.

We need to be certain that the dependencies have not changed since our current item was cached. This means that our dependency list must store the version that each dependency was at when the current key was generated.

The parent node will need to store its list of dependencies and the version that they were when it was cached. When we retrieve the parent key we need to check its list of dependencies and make sure that none of them have changed.

Now that we've stored all the information we need to manage our structure, lets see how it works.

Lets say that one of the children changes and is now version 2. We update the key storing its most current value as part of the update to the value, using our key based expiration implementation.

On the next page render our class will try to pull the parent node from cache. It first inspects the dependency list and it realises that one of the children is currently on version 2 and not the same version it was when the parent was cached.

We invalidate the parent cache object when we discover a dependency has changed. This means we need to regenerate the parent. We may want to implement a dogpile lock for this, if you're expecting concurrency on the page.

Only the child that has changed needs to be regenerated, and not the other two. So the parent node can be rebuilt by generating one child node and reading the other two from cache. This obviously results in a much less expensive operation.

Cache warmup scripts

A warm cache is one which contains useful data whereas a cold cache is one which is either empty or contains irrelevant data. If you've just restarted Memcached it will be empty, remember that Redis persists to disk but Memcached doesn't.

If your cache is cold then your clients will be forced to run the expensive operation to get whatever resource it is they're looking for. This can place quite a load on your database or whatever resource it is you're caching.

A cache warmup script will populate your cache with useful information. Normally your application will need to run for a bit while it warms up the cache, having it pre-populated helps reduce the time your app is using a cold cache.

Actually generating a warmup script is its own problem though, and will be heavily customised for your particular application. You obviously won't be able to cater for every edge case but if you have a good idea of some of the common expensive queries then you could include them into a script that sets up the cache values for them.

Enemies of scale

Overengineering

Having worked in multi-disciplinary teams I have to admit that it's usually the Java developers who come up with a beautifully complex solution to a fairly simple problem.

Don't get me wrong, it's not about the programming language Java. I think they were probably the best trained of us all when it came to object abstraction. It was simply the hammer that they used to make every problem look like a nail.

The problem I've had with over-engineering is that it wastes time. The product that is delivered is a black box to me and I can't easily deploy it.

Complex code incurs technical debt and trying to scale up an application that is difficult to understand is very difficult. I'm busy with such a project at the moment and a great deal of my effort is going into getting the code into a form that is simple. Once the code is simple it becomes malleable, and so scaleable.

Inelastic infrastructure

I have worked in places where adding a new server was a major consideration that involved procuring the bare metal server and installing it into our rack at the data centre. The cost involved was substantial and we would get a fairly large step up in capacity for the money we spent.

Although we pay a premium for using cloud instances I believe the flexibility that they provide is definitely worth it. I can spin up a server and later decide to change its CPU and memory capacity without having to drive down to my data centre with a screwdriver.

I also really enjoy the fact that I can add much smaller "step ups" in my infrastructure and so respond more accurately to the demand on my network.

I think if I were to take my cloud experience back to the metal I would focus on looking for a way to use cheap commodity hardware and software. This would make the decision to ramp up easier and make it less expensive when something fails. Remember that if adding a server costs a significant amount then a proper procurement process must be followed. This is not really fun for anybody involved and takes time, so if you're in a hurry to respond to demand for your application you're going to sweat.

Not differentiating your persistence layer

Relational databases are great when you need ACID compliance and need to model relationships between data entities. There are however other persistence tools that may be suited better to other tasks.

Even the file system can be used quite successfully as a persistence layer. With modern file systems and the Linux OS cache you'll be able to store very large data sets. If you don't need to track relationships, have data that is not very volatile, and don't want to pay the overhead of ACID then a file system is a great place to store data.

If you need to store sessions then a key-value database such as Redis or Memcached is a great option.

Document stores like MongoDB or CouchDB allow you to create data models based on "documents". These documents contain the metadata about the fields they contain themselves. In other words you don't have to define the schema for a document in the database - the schema is contained within the document itself. Documents are keyed but can also be indexed on one or many of their keys.

Document stores are not ACID compliant but offer advantages over RDMS. MongoDB for example supports partitioning, sharding, and horizontal scaling right out of the box. In fact it was designed with these principles in mind which makes it much easier to scale than MySQL.

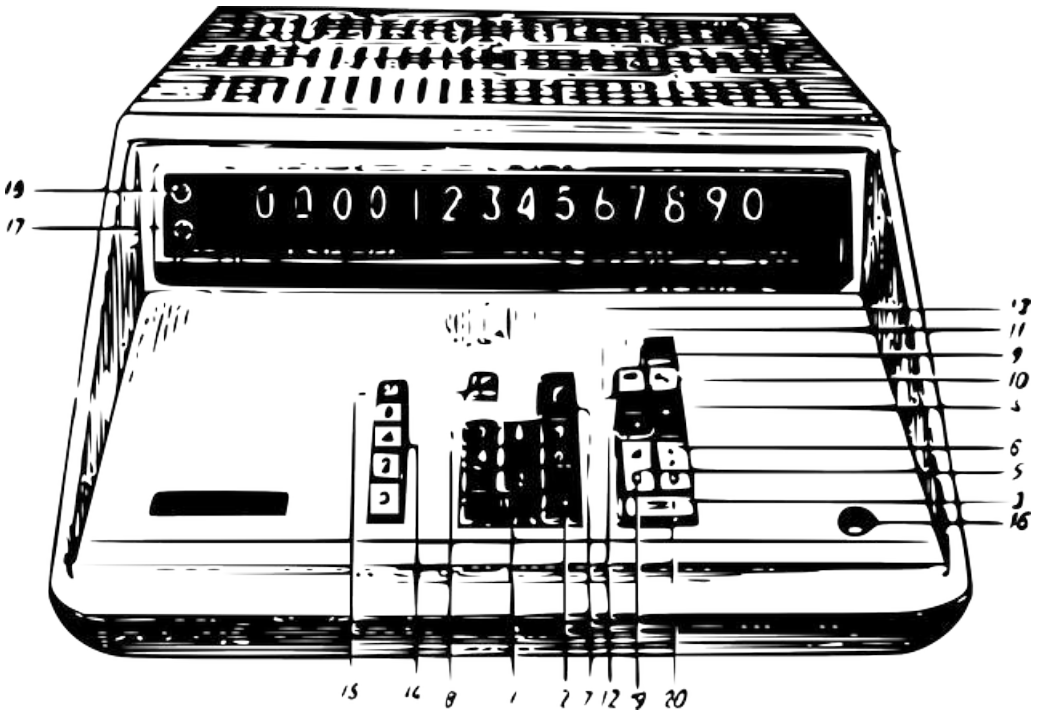
MapReduce applications like Apache Hadoop offer the ability to scale up extremely large data sets. They offer two functions - a Map function that performs filtering and sorting and a Reduce method that performs a summary operation. The key to scalability is the fact that the Map and Reduce functions can be performed on distributed servers.

Trying to solve every data storage problem with MySQL can make it more difficult to scale.

Not measuring and logging

I've dedicated a good portion of this book to metrics so I won't go into detail here. Suffice to say that you should be logging your application and using a service like Loggly or Splunk to centralize and monitor your logs for alert conditions.

Metrics



"I think it's much more interesting to live not knowing than to have answers which might be wrong. I have approximate answers and possible beliefs and different degrees of uncertainty about different things, but I am not absolutely sure of anything and there are many things I don't know anything about, such as whether it means anything to ask why we're here. I don't have to know an answer. I don't feel frightened not knowing things, by being lost in a mysterious universe without any purpose, which is the way it really is as far as I can tell."

- *Richard Feynman*

In this section of the book we will focus on gaining insight as to how our code is performing. We'll identify bottlenecks, find slow running code, and discover why sometimes a page takes longer to load than it should. We need metrics in order to understand and quantify the effects, test our assumptions, and plan for growth. Metrics will inform our tuning efforts and without them we we'll be tuning in the dark.

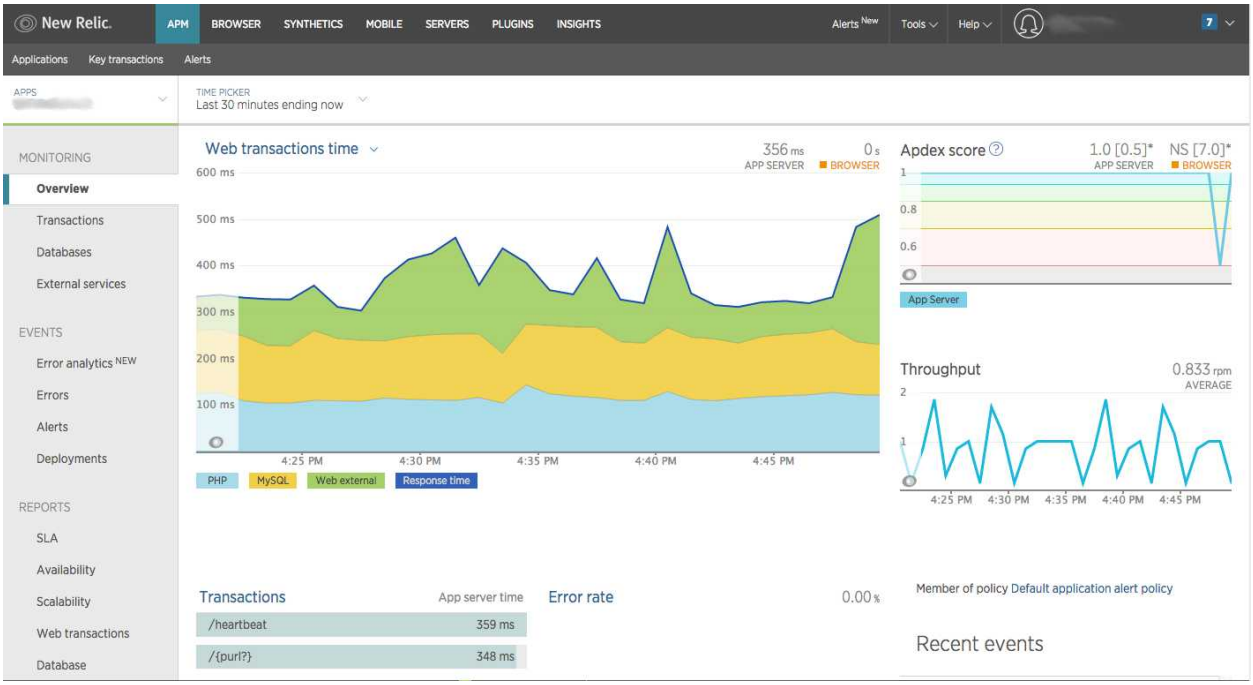
Profiling Your Code

Short of putting a bunch of `var_dump(); die();` statements into your code how do you track the running status of your PHP code?

There is the excellent Kint[32] package which should replace your `var_dumps` but it's pretty disruptive to output information in the middle of your program run. It may help to debug problems but it won't help you identify performance issues.

New Relic

New Relic can be pretty expensive but it does give some very solid insight into your running code. The free version lets you monitor and track performance, but in my opinion the killer feature of New Relic is in the details about database queries that you get with the paid version.



New Relic Screenshot of an unoptimised application

One of the really nice things about New Relic is the ability to see a friendly dashboard that gives a pretty high level overview of your code.

[32] https://raveren.github.io/kint/

In the above screenshot you can see that a fair amount of time is spent waiting for an external web service in this application. A quick few clicks through the New Relic dashboard shows that we're making a call to www.useragentstring.com to parse the user agent. We can then determine whether this information is really worth. In this particular case I'd recommend caching the lookup result.

We can also see that we're spending a fair amount of time hitting the database, and we can evaluate how to improve this either by optimising the queries or using caching.

Using New Relic requires you to install an extension for PHP but is very easy to setup. Once New Relic is installed it will monitor all the requests to your application.

There is a Chrome extension for Loggly[33] that let you link log events to your New Relic dashboard. When you have the extension installed your New Relic dashboard will display a button that takes you directly to a log view that searches for incidents at the same time.

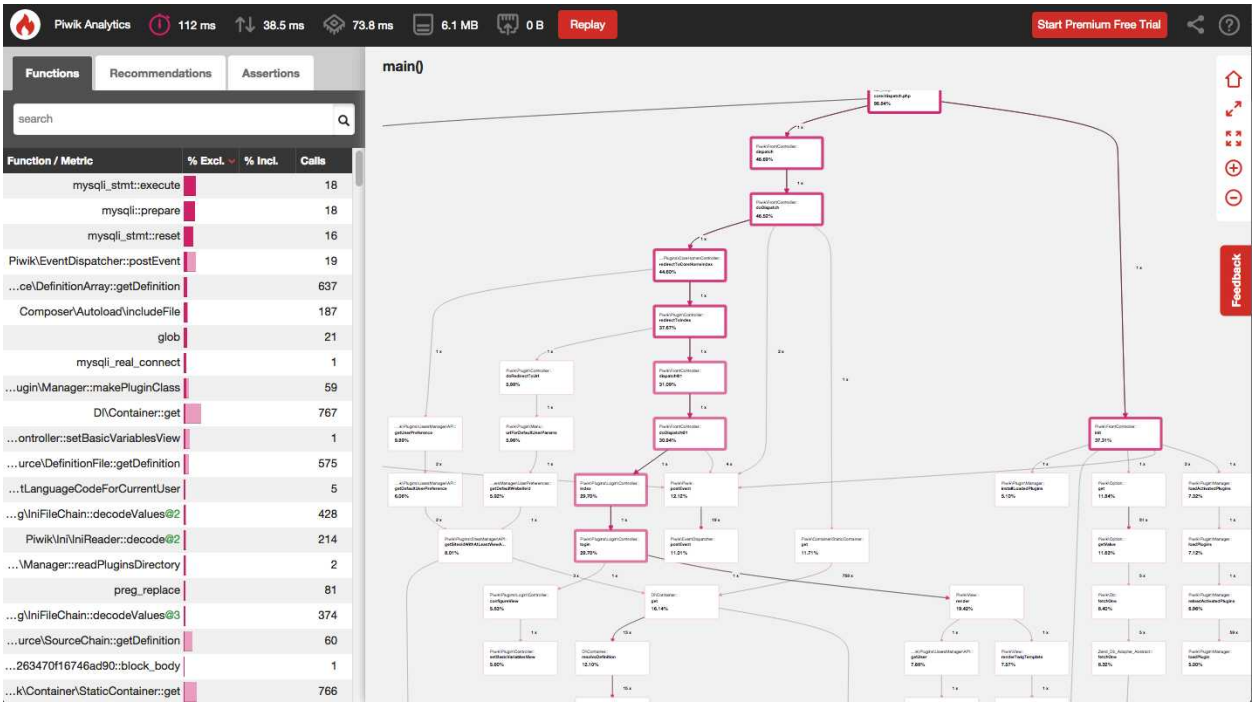
Blackfire.io

Blackfire.io[34] offers a great profiling tool. It only runs when you deliberately cause the profiling to run and not all the time as New Relic does. To my mind this is preferable and I appreciate the ability to run multiple samples of a request.

I think one of the killer features of Blackfire is the way you can organise the call graph. You can target one of several metrics and Blackfire will highlight the parts of your code that were most active for that metric. It's very easy for you to spot which parts of your code are using more memory or CPU than other parts.

Blackfire also makes it very easy to spot how many times a piece of code was called.

[33] https://www.loggly.com/
[34] https://blackfire.io/



Blackfire call graph

This is the profile of the Piwik analytics package login screen. We're examining the wall time of each code block, where the wall time is the difference in time between the entry and exit of a piece of code. It's different from CPU time which measures the actual amount of time the CPU spent on code in the block.

I think the really great part of Blackfire are the tools it offers to help you do split testing on infrastructure changes. You can easily compare different benchmarks against each other which lets you make intelligent data driven changes to your code and infrastructure.

Xdebug and Kcachegrind

Xdebug[35] is a PHP extension. It provides some pretty formatting for var_dump() and error messages and is also able to generate profiling logs. Kcachegrind[36] is a tool to visualize the profiling output that Xdebug creates. There is a Mac port called Qcachegrind and you can be install it with Homebrew.

Xdebug and Kcachegrind are an open source combo that let you generate call graphs and analyze where your code is spending time running. You can do the same thing a little easier with Blackfire but a side benefit to Xdebug is that it can plug into your IDE for interactive debugging.

You should not run Xdebug on production servers because it has performance implications

[35] https://xdebug.org/

[36] https://kcachegrind.github.io/html/Home.html

Installing Xdebug is simple and it should be part of your standard development environment.

```
sudo apt-get install php5-dev
pecl install xdebug
```

To enable profiling you need to edit php.ini and set the `xdebug.profiler_enable` setting to 1.

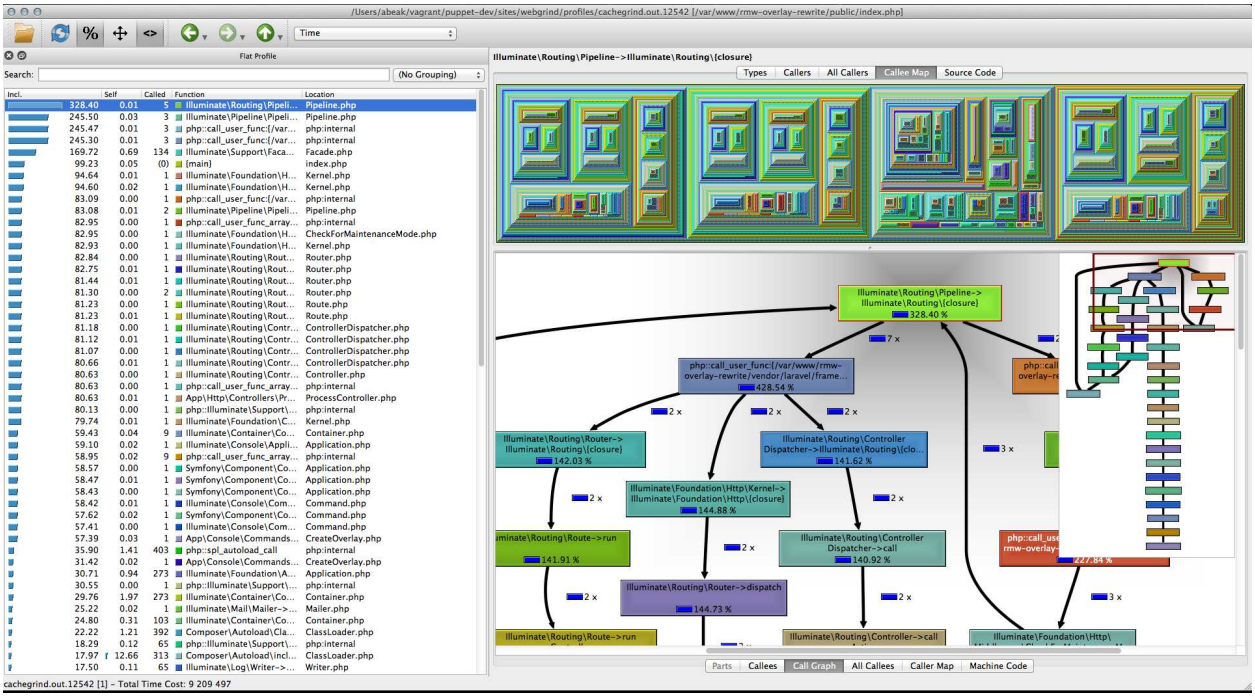
Xdebug will start writing profile information to the directory specified in the xdebug.profiler_-output_dir directory.

Profile information can build up very quickly so remember to turn off debugging once you've gathered the information you need.

You can use the `xdebug.profiler_enable_trigger=1` setting in your php.ini file. This lets you turn profiling on by passing a GET or POST variable called XDEBUG_PROFILE with a value of 1.

When your run has finished, turn off profiling and then open up the file in Kcachegrind.

You should see something like this screenshot of Qcachegrind running on my Mac:



I turn off cycle detection because I find it easier to track through the flow without it. You can find that as a view option in the top menu. If you're using Mac you'll need to install graphviz with brew in order to see the callgraph.

Although the screen is bewildering at first you'll be able to start making sense of where your code is spending a lot of time.

I prefer the Blackfire free edition because it can be used on a production server without any performance implication. It also produces a call graph that can be sorted on various metrics.

Strace

Strace[37] is a Linux command that will let you analyze your code as it is running on the server. It's great for identifying I/O bottlenecks or services that have "gone away".

You can find a list of your PHP processes by running `ps`, like this:

```
$ ps ax | grep php
92: 1224 ? Ss     0:52 php-fpm: master process (/etc/php5/fpm/php-fpm.conf)
99: 5866 ? S      0:03 php-fpm: pool www
```

Then you attach and start listening to one of the workers with `sudo strace -r -p 5866`.

Whenever this process makes a system call strace will output a log of this to your console. The -r option causes strace to print a relative timestamp upon entry to each system call. This records the time difference between the beginning of successive system calls.

You'll get a long list of calls that looks something like this:

```
 0.000155 write(4, "\1\6\0\1\1\354\4\0X-Powered-By: PHP/5.5.9-"..., 512) = 512
 0.000107 write(4, "<!DOCTYPE html>\n<html lang=\"en\">"..., 65528) = 65528
 0.000560 write(4, "\1\6\0\0013\260\0\0", 8) = 8
 0.000039 write(4, "nbsp;</td>\n\t\t\t\t\t</tr>\n\t\t\t\t\t\n\t\t\t\t"..., 13232\
) = 13232
 0.001001 write(4, "\1\6\0\1\0\5\3\0tml>\n\0\0\0\1\3\0\1\0\10\0\0\0\0\0\0.9-",\
 32) = 32
 0.002368 chdir("/")                    = 0
 0.000052 clock_gettime(CLOCK_MONOTONIC, {2346075, 937591337}) = 0
 0.000028 times({tms_utime=325, tms_stime=31, tms_cutime=0, tms_cstime=0}) = 195\
2564512
 0.000097 umask(022)                    = 022
 0.000054 fcntl(3, F_SETLK, {type=F_UNLCK, whence=SEEK_SET, start=0, len=0}) = 0
```

Although this might be an information overload it will help you to find places where your code is waiting for a service. You'll be able to tell if your code is hanging, and which calls are taking significant amounts of time.

To me strace is much more of a tool to use when things are going badly wrong and you need to work out what is causing your app to hang. It's not at all useful to monitor your app or identify behaviour patterns.

[37] http://linux.die.net/man/1/strace

YSlow and PageSpeed

Yslow[38] was written by Yahoo and is the first application of its type that I remember using. I'm sure there were other tools, but YSlow was the first to give specific advice on improving your page load speed.

PageSpeed[39] by Google is a new entrant and is also a great tool. I really like PageSpeed because of the ability to do mobile and desktop testing.

These tools will let you analyze your page loading time and will give you specific advice on restructuring your code.

[38] http://yslow.org/

[39] https://developers.google.com/speed/pagespeed/?csw=1

Siege

Siege is the best piece of free software I've found to stress test your code. It has a number of great features, including the ability to randomly select pages from a list you give it. You can download it for free from its homepage[40].

I've used Siege to benchmark an application to demonstrate the effect of performance optimizations. I've even used it to demonstrate the fact that Nginx was able to withstand a much higher number of simultaneous connections than Apache.

Configuring Siege

Siege is packaged for Debian and Ubuntu and you shouldn't have any difficulty building it for your distro.

Once you've got it up and running use the `siege.config` file to create a .siegerc file in your home directory.

You can run `siege -C` to view the values in the config file

I usually leave Siege with default options and use the command line switches as required.

Generating a list of URLs

Siege lets you specify a list of urls that it will choose from when making a request. If you don't want to manually choose the list, then it is possible to examine your access log and automatically generate a list of common urls.

I generally use the nginx 'combined' format for my logs so this command works admirably:

```
awk -F\" '{print $2}' access.log | awk '{print "http://test.com"$2}' | sort | un\
iq | sort -r > siegelist.txt
```

Here access.log is my combined format nginx access log for the site and http://test.com is my domain. I then direct the output of this command into the siege url file.

Obviously using your log will include all of the static content files like images and CSS, but its useful when benchmarking your server to hit these files too. If you're wanting to only profile your PHP code you will need to use a pattern like this for awk:

[40] https://www.joedog.org/siege-home/

```
awk -F\" '($2 ~ "php"){print $2}' access.log ....
```

So now you have a list of url's that have been recently accessed on your server and you're ready to run Siege.

You'll be adding two parameters to tell Siege to use your file:

```
siege -i -f siegelist.txt
```

The -i parameter tells Siege to use random urls from the file, and -f lets you specify the location and name of the file.

Setting up POST requests

You are able to use the URL file to specify POST requests.

```
http://carbon.dev/login POST user=admin&password=supersecret
```

Running Siege

The two options I fiddle with most are the ones which affect how many simultaneous connections to make and for how long I want to keep the tests running.

You can also influence the delay in seconds between requests but for the most part I find the default of 3 seconds to be just fine.

There are essentially two ways to limit how long siege runs for:

You can specify -t with a time parameter to run the test for a set period of time. Or you can specify -r with the number of repetitions for siege to complete for each concurrent user.

Alternatively, you can specify a number of repetitions for Siege to run for each of the concurrent transactions.

If you do not specify a limit for Siege it will run until you press CTRL-C.

Parameter	Used for	Example
-c <int>	Number of concurrent connections	-c 100
-r <int>	Number of repetitions for each connection	-r 10
-t	How long to run the test for	-t5m
-i	Pick random URLs to visit	
-f	Specify the file name to use as the url list	-f siegelist.txt
-q	Quiet mode - suppress output	

Once you kick off siege you'll see something like the snippet below unless you use quiet mode. Siege will show information about what is going on, but for the most part the statistics at the end of the run are what you're looking for.

```
$ siege http://carbon.dev -c 5 -t20s
** SIEGE 3.1.0
** Preparing 5 concurrent users for battle.
The server is now under siege...
HTTP/1.1 200   1.65 secs:    3130 bytes ==> GET  /
HTTP/1.1 200   1.65 secs:    3127 bytes ==> GET  /
HTTP/1.1 200   1.66 secs:    3127 bytes ==> GET  /
HTTP/1.1 200   1.66 secs:    3127 bytes ==> GET  /
....
```

Once Siege has run for the time limit (or you press CTRL-C) you'll get a summary screen like this one:

```
Lifting the server siege...      done.

Transactions:                        86 hits
Availability:                    100.00 %
Elapsed time:                     19.78 secs
Data transferred:                0.26 MB
Response time:                       0.68 secs
Transaction rate:                4.35 trans/sec
Throughput:                       0.01 MB/sec
Concurrency:                      2.97
Successful transactions:          86
Failed transactions:                 0
Longest transaction:              1.70
Shortest transaction:             0.17
```

Interpreting results

Testing for a specified amount of time

When I've fixed the time that the test runs then the number of successful transactions is a key metric for me.

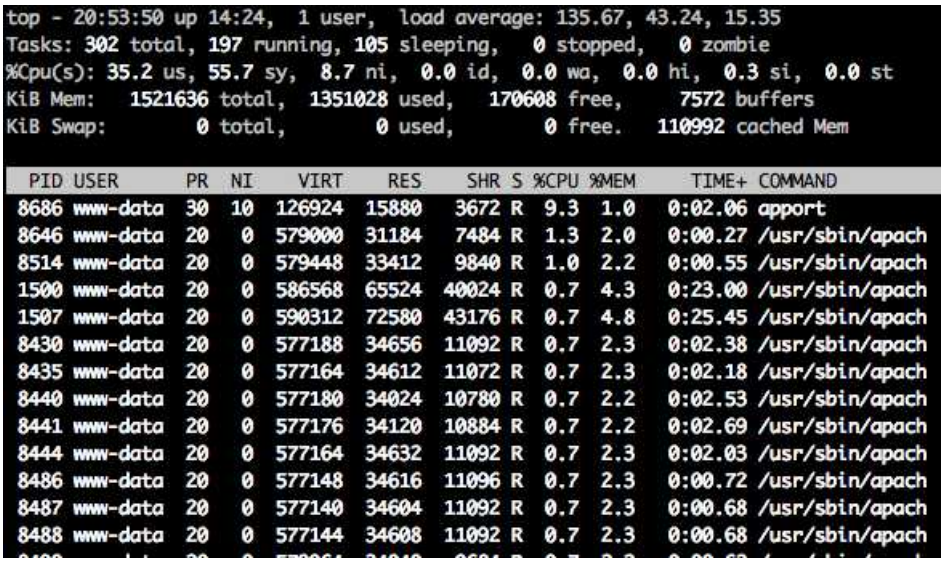
If I'm tuning the server or my application and there is an appreciable change in the number of transactions that can be performed in a given time then I can obviously start to infer the effect of my tuning.

Availability is a great metric to watch because you'll be able to estimate the traffic volume that causes your server to start to fail.

Concurrency is the average number of processes that were running at the same time. It's calculated as total transactions divided by elapsed time.

Concurrency in this environment is caused by requests still being processed when Siege sends a new one. If my concurrency is building up then my requests are not finishing quickly enough to be cleared from the web server. This builds up the load on the server.

If I have high concurrency and 100% availability then it seems that the server is able to handle the simultaneous requests. As soon as concurrency is high and the availability drops then I know that I need to either improve my application performance or scale up my server.



An Apache server on a VM dies

In the screen shot I'm using the `top` command on my Ubuntu VM which is running Apache. I've thrown 100 simultaneous connections at it and the VM is flooded. Take a look at that load average!

Testing a given number of transactions

You can run Siege with the `-r <repetitions> -c <concurrent connections>` parameters to set a number of transactions for it to complete. For each concurrent connection Siege will perform the number of repetitions you specify.

You can see that the total number of transactions will be `repetitions * connections`.

By measuring the elapsed time you can see how quickly your server is processing the transactions you're throwing at it. If the elapsed time of your tests is going down then that's a good thing and your tuning is working.

Having a higher concurrency means that your server is processing more requests at the same time. If this results in a faster elapsed time then that's great, but if your tests take longer to run with higher concurrency then you need to change your tuning strategy.

Profiling MySQL

We'll be looking at tuning MySQL later in this book but while we're still considering how to gather metrics to see the effect of tuning lets see how we can measure MySQL performance.

Benchmarks are artificial and so might not match up to real world load and performance patterns. We'll try to look at how we can best simulate real world behaviour but mustn't lose track of the context in which we generate our numbers.

Benchmarks are an approximation to the real world, and are only directly comparable with runs of the same benchmark. We can only compare like with like so we must control the environment between successive runs of a benchmark to make sure we are only changing the elements we want to test.

We want our benchmark to be as close as possible to the real world and so we should try to use a snapshot of live data to benchmark on. There are often logistical problems that interfere with this, for example here in the EU we are very careful not to use personal details of real people for benchmarking.

When we profile MySQL we can try to do so in isolation or as part of the application performance as a whole. There are benefits to testing MySQL within the context of the application, such as including effects of network traffic and other components. Profiling MySQL by itself is useful when optimising individual queries, indices, or schemas.

Database Metrics

Lets look at some of the metrics that we will want to measure. This will help us decide what areas we want to tune, understand tradeoff decisions, and translate the effect of our database tuning to our application performance.

Throughput

Throughput is a measure of the number of transactions that your database processes in a unit of time. It's easily understood and is a classic benchmark of performance. Most commonly you'll see graphs of transactions per second.

Response Time or Latency

This measures the total time that a task takes to execute. This metric would be benchmarked by running the task repeatedly and then deriving statistics from the results. This reduces the chance

that variability between runs will skew your interpretation of the data. A good way to analyze this information is to group it into percentiles. For example knowing that 98% of the time the task took 8 milliseconds or less is useful information.

Scalability

Scalability itself is a metric. A scalable system is one where adding hardware results in a proportionate increase in performance. Ideally doubling your hardware capacity results in doubling your performance, but this is seldom the case and most systems do not scale linearly.

Measuring the scalability ratio of your stack helps you to plan capacity. You'll know where the benefit of throwing more metal at the problem starts to become more expensive than its worth.

Benchmarking strategies

It is important to plan your benchmarking tests in order to be able to derive meaning from their results and to make them repeatable.

A benchmark that bears no resemblance to your application is not going to help us improve the application performance. We can tune MySQL to improve our benchmark but there is no guarantee this will improve our users experience with our app.

We have to examine the pattern of usage of our application and try to develop a benchmark that emulates them as closely as possible.

Concurrency and User behaviour

As with web servers concurrency is a measure of the number of tasks being simultaneously executed.

We want to understand the effect on throughput of simultaneous connections performing tasks on the server. Rather than trying to achieve high concurrency we should use concurrency as a control variable in the environment we are benchmarking.

If we have an application that we expect to be making 100 connections to the database during normal operation then our benchmark should reflect this. Benchmarking and tuning with a lower (or higher) concurrency may lead to optimisations that do little to help our application.

The rate at which users interact with the database is not typically uniform. There are load spikes and periods where they spend time reading what's on their screen where the load drops. Your concurrency should reflect these natural ebbs and flows.

Similarly user behaviour probably doesn't repeat the same query thousands of times in a loop. Your cache is likely to make repeating the query must more efficient in benchmarking than it would be to run in live.

Caching

Caches in a system that has been running for a while are "warm" in that they are populated with useful data. A cache that is cold will result in cache misses and your database will need to perform queries. If you're wanting to benchmark your database under normal running conditions you'll need to develop a strategy to make sure that any caches are warmed up.

Choosing input data

Ideally we want to have a good representation of live data, both in terms of volume and content. If your live database is going to be 100GB then benchmarking against a small 1GB slice might not be accurate.

Similarly if your data used in benchmarking is random and doesn't reflect the composition of your live database you may not accurately capture the performance characteristics of your stack against live. Real data often has pieces that are more in demand than others.

Running time

Your benchmark should run for long enough to warm up your caches and properly cover enough of your normal live patterns of usage so as to accurately model your production system. When you look at the output you should be able to see that your metrics have "settled down" into a pattern.

Profiling tools

mysqlslap

mysqlslap[41] is released as part of MySQL and can be used to profile MySQL.

I downloaded the demonstration employee database[42] from MySQL and ran the command:

```
mysqlslap --user=root --password --host=localhost --concurrency=25 --iterations=10 --
create-schema=employees --query="SELECT * FROM dept_emp;" --verbose
```

After a short while my little VM came back with the response:

[41] https://dev.mysql.com/doc/refman/5.7/en/mysqlslap.html

[42] https://launchpadlibrarian.net/24493586/employees_db-full-1.0.6.tar.bz2

```
Benchmark
Average number of seconds to run all queries: 4.427 seconds
Minimum number of seconds to run all queries: 4.351 seconds
Maximum number of seconds to run all queries: 4.557 seconds
Number of clients running queries: 25
Average number of queries per client: 1
```

You can tell mysqlslap to auto-generate a standard schema. This is useful for benchmarking system performance. It's also possible to specify an input file consisting of a number of queries, which lets you more accurately model typical database usage.

SysBench

According to its project page on Github, SysBench[43] is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters that are important for a system running a database under intensive load.

On Ubuntu it can be installed by running:

```
apt-get install sysbench
```

SysBench performs a number of benchmarks:

- fileio - File I/O test
- cpu - CPU performance test
- memory - Memory functions speed test
- threads - Threads subsystem performance test
- mutex - Mutex performance test
- oltp - OLTP test

The advantage of SysBench as a benchmarking tool is that it goes out of its way to perform the tests in a manner similar to the actual usage pattern of MySQL. The results from SysBench are supposed to be directly applicable to how suitable the system is as a MySQL server, and not just a general benchmark of performance which other tools provide.

OLTP testing

Using SysBench to test OLTP involves three steps:

1. setting up a table for the test using the *prepare* command,
2. running the test, and
3. dropping the test table with the *cleanup* command.

The test results will look something like this:

[43] https://github.com/akopytov/sysbench

```
OLTP test statistics:
    queries performed:
        read:                            621712
        write:                           0
        other:                           88816
        total:                           710528
    transactions:                        44408  (740.05 per sec.)
    deadlocks:                           0      (0.00 per sec.)
    read/write requests:                 621712 (10360.74 per sec.)
    other operations:                    88816  (1480.11 per sec.)

Test execution summary:
    total time:                          60.0065s
    total number of events:              44408
    total time taken by event execution: 479.8635
    per-request statistics:
         min:                                  0.69ms
         avg:                                 10.81ms
         max:                          18446744072196.55ms
         approx.  95 percentile:              31.72ms

Threads fairness:
    events (avg/stddev):           5551.0000/53.27
    execution time (avg/stddev):   59.9829/0.00
```

You can specify the size of the test table, the number of concurrent visitors, and how many threads to use. All of the options are well documented in the command man page.

FileIO testing

The man page for sysbench has an example of how to run a benchmark to measure FileIO:

```
sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rn\
drw prepare
sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rn\
drw run
sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rn\
drw cleanup
```

The file-test-mode specifies different I/O operations:

Flag	Order	Mode
seqwr	Sequential	Write
seqrewe	Sequential	Rewrite
seqrd	Sequential	Read
rndwr	Random	Write
rndrw	Random	Read / Write
rndrd	Random	Read

The output will include a lot of information, including the following snippet which gives us our throughput (122.67Mb/sec) and the number of requests per second (7850.72)

```
Operations performed:  0 Read, 196608 Write, 128 Other = 196736 Total
Read 0b  Written 3Gb  Total transferred 3Gb  (122.67Mb/sec)
 7850.72 Requests/sec executed
```

Those are good metrics to use to measure the capacity of the machine to work as a database server.

CPU benchmarking

CPU benchmarking is performed by calculating prime numbers up to the value specified in the –cpu-max-primes option. SysBench calculates them using 64bit integers.

Each thread executes the requests concurrently until either the total number of requests or the total execution time exceed the limits specified with the common command line options.

sysbench –test=cpu –num-threads=20 –cpu-max-prime=20000 run

By default SysBench will use 1 thread.

Servers

"When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem." - *Edsger W. Dijkstra*

Replacing Apache with Nginx

"Apache is like Microsoft Word. It has a million options but you only need six. NGINX does those six things, and it does five of them 50 times faster than Apache." - *Chris Lea*

The chief difference between Apache and Nginx is their process model. Apache requires a separate process for each connection it is serving. It is common for clients to make several connections to the server at once which places the need for Apache to run lots of processes. This consumes a lot of memory. Nginx is an event based process which means that each process is able to handle several connections.

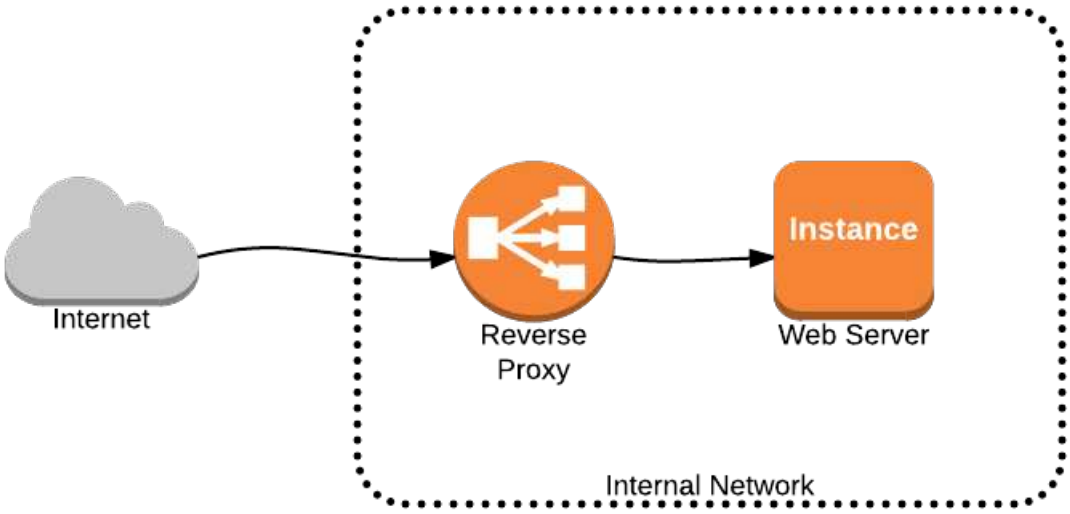
Nginx is not always a direct replacement for Apache, but when I've tried to use it as a proxy for Apache I've always ended up wondering why I'm configuring two web servers when Nginx is perfectly capable itself.

If Apache is all you know and you're reluctant to try out a new server then go check out the Nginx wiki[44] and take my word for it - Nginx is really easy to configure and the performance gains are very real.

Nginx as a Reverse Proxy

Placing Nginx in front of Apache helps because Nginx is better able to cope with highly concurrent requests. When it's deployed like this it acts as a reverse proxy. It accepts incoming requests and passes them on to the Apache web server.

[44] https://www.nginx.com/resources/wiki/



My path to Nginx was actually by way of Varnish[45] which is a really great piece of software and alleviated a great deal of load from my Apache server. This made the site quicker to load and more reliable in general.

One problem that I had was that Varnish does not cache content with cookies in it. I was not able to make changes to the application to get it to work with edge side includes which would have allowed Varnish to work around the cookies.

So the main gain I ended up getting was from serving static content from the Varnish reverse proxy instead of Apache.

This was a pretty big gain because Varnish could handle the large volume of connections that had been causing the site to fall over. However I was not actually getting the full benefit of Varnish's caching ability and it was acting more like a reverse proxy than a cache in the great majority of my requests.

My first experiment with Nginx was as a reverse proxy. I replaced Varnish with Nginx and used Nginx to serve static content while passing PHP traffic on to Apache. This still reduced the load on my Apache server and was easier to set up than the Varnish/Apache stack.

Nginx worked great as a reverse proxy. It was able to respond to multiple connections to get static content like images, stylesheets, and javascript. It passed on PHP requests to Apache and my site was a great deal more stable.

One place where Varnish did make a substantial difference was in serving a Wordpress site. The site had many thousands of articles and was really (very) sluggish to load. Placing Varnish onto the server and installing a Wordpress plugin to make use of it shaved 6 seconds (6000ms) from the load

[45] https://www.varnish-cache.org/

time on average. A site that takes that long to load is just about useless, so Varnish really saved the day.

I found that Varnish is truly great for caching this sort of pseudo-static content.

Nginx with PHP-FPM

It's very simple to configure Nginx to pass PHP requests on to the PHP5-FPM service instead of the Apache server.

I did experiment with using Hiphop[46] instead of PHP but abandoned Hiphop because it did not have native Postgres support. Hiphop was substantially faster than native PHP. For some real world tasks (building a PDF format calendar from source images and a database of special dates) my runtime was cut from 30 seconds to 9 seconds.

This is how I remember the actual numbers

One of the advantages to Hiphop was that it had a built in bytecode cache. Enabling Opcache in PHP-FPM also leads to significant gains and is just a matter of changing one config setting.

Use the most recent stable release of PHP as there are normally performance advantages to upgrading (be careful about upgrading to PHP7 though).

Ultimately although HHVM was consistently marginally faster than PHP-FPM I found that it was not yet 100% compatible and I rather reluctantly had to settle on using PHP-FPM.

Setting up Nginx with PHP-FPM

Setting Nginx up to use PHP-FPM is very simple and there are any number of guides available on the Internet. I maintain a Gist[47] that I usually use as a starting point when setting up a new server.

I'm going to show how to do it on Ubuntu 14.04 LTS. I'll start with a vanilla Ubuntu instance and set it up from scratch.

The commands to install the software are:

[46] http://hhvm.com/

[47] https://git.io/v299s

```
sudo su
apt-get update
apt-get -y install nginx
apt-get -y install php5-fpm php5-mysql php5-curl php5-memcached
wget https://git.io/v29Sb -O /etc/nginx/sites-available/default
service nginx reload
```

At this point if you navigate to the IP address of your server you should see the nginx welcome page. We've also installed PHP and some of the modules we might need. Obviously you should tailor this to your taste.

You probably want to download a copy of nginx_modsite.sh[48]. I place it into /usr/local/bin/nginx_modsite and make it executable.

We're using the config file from my Gist which is a quick shortcut to setting up a site. I also keep a copy of an HTTPS configuration handy on Github Gists[49] that has the basic security settings set up. Note, however, that I am forced to maintain support for IE8. Even though Microsoft has formally disavowed it some of my clients are banks and they are slow to move away from software they trust. Because I need to support IE8 I have enabled TLS 1.0, which you should probably disable in your config.

Opcache

Now we're going to configure PHP-FPM and enable opcache. Open `/etc/php5/fpm/php.ini` in your favourite editor and search for 'opcache'. The first hit should be the start of the Opcache settings.

We're going to make the following changes:

```
opcache.enable=1
opcache.validate_timestamps=0
max_accelerated_files=5000
opcache.fast_shutdown=0
opcache.memory_consumption=128
```

The first setting is obviously going to enable Opcache, it is disabled by default.

We set `validate_timestamps` to zero in production settings. This tells Opcache never to check if a file has changed. If it has a cached version of the files bytecode then Opcache will use that, regardless of how old the cache is. When you deploy your code you need to tell PHP5-FPM to reload with `service php5-fpm reload` in order to invalidate the cache.

[48] https://git.io/v299z
[49] https://git.io/v299t

On your development machine you probably want to disable it entirely or have Opcache regularly checking files for changes. It's going to make debugging more difficult if after changing some code your application still uses older versions from cache.

The value for `max_accelerated_files` should be set to something more than the number of PHP files in your project. This value is the maximum number of PHP files that your cache will store.

Opcache does not have an eviction strategy and will perform a cache restart when it reaches the maximum number of accelerated files. An Opcache restart means that the cache is emptied and every script is recompiled. This has obvious performance implications, and can create a thundering herd problem[50].

You can count the number of PHP files in your project using the Bash command `find project/ -iname *.php|wc -l` The `memory_consumption` setting is something you should tweak on your setup. Have a look at how much free memory your server has and if you can afford to increase it from 128 megabytes then do so. Just make sure you don't deny memory to your other services!

If Opcache reaches the memory limit it will attempt to restart the cache. You should profile your application with the opcache get status()[51] function to find a memory limit that prevents Opcache from restarting.

We're actually leaving the `fast_shutdown` value at the default of zero to disable it but I wanted to explicitly mention it. This option assists with the deconstruction of objects and would offer a speed improvement, but there is a fairly long standing open issue on the Opcache Github project[52] relating to using it.

Wasted Memory

When PHP-FPM is set to validate timestamps it will recompile a script if it has changed. The memory space that it used to occupy in the cache is not freed up, and is considered "wasted" by Opcache.

When the percentage of the maximum memory that is wasted reaches the threshold specified in the `max_wasted_percentage` setting then Opcache will trigger a full restart.

Opcache also performs restarts when it reaches the memory limit or when it reaches the maximum number of files. However it only does so if the wasted memory in Opcache exceeds the `max_wasted_percentage`.

In our production setting we're never validating our file timestamps which means that our wasted percentage should not increase. This simplifies the tuning process somewhat.

[50] https://en.wikipedia.org/wiki/Thundering_herd_problem

[51] https://secure.php.net/manual/en/function.opcache-get-status.php

[52] https://github.com/zendtech/ZendOptimizerPlus/issues/146

It does, however, imply that Opcache won't try to restart when it reaches the memory limit or maximum number of files. By monitoring your application with opcache get status()[53] you will be able to tune those values to reduce the number of cache restarts.

Configuring Nginx

Out of the box Nginx is a fast and stable web server. Of course the default settings are designed to work generally and so we can tune a few settings for your hardware to optimise it.

Configure worker connections

Because of its event driven architecture Nginx worker processes can each handle thousands of connections. Nginx spawns a single master process which controls a number of worker processes.

The maximum number of concurrent requests your server can handle is the number of worker processes multiple by the number of connections each.

A good option to look at is to match the number of worker processes to match the number of cores your server has. You can see how many processor cores your server has by running the command `cat /proc/cpuinfo |grep processor | wc -l`. Set your worker processes to the output of that command and see the effect by using Siege.

If you're using Linux kernel 2.6+ you should use epoll connection processing[54].

Lastly you can increase the maximum number of connections each worker can handle. You don't want to set it too high, but the default values are a bit low. I've normally set it at 2048 and haven't noticed substantial benefits in increasing it past there.

The top of your nginx.conf file should look something like this:

```
1  user www-data;
2  worker_processes 2;
3  pid /run/nginx.pid;
4
5  events {
6          worker_connections 2048;
7          multi_accept on;
8          use epoll;
9  }
```

Configure static file caching

The server block we're using for our site in the config above tells the browser to cache certain files:

[53] https://secure.php.net/manual/en/function.opcache-get-status.php
[54] http://nginx.org/en/docs/events.html

```
location ~* \.(js|css|png|jpg|jpeg|gif|ico)$ {
    expires max;
    log_not_found off;
}
```

This lets the client know that it should cache the files for as long as possible.

Nginx also has the ability to cache meta information about files to help it avoid hitting the disk. This is configured with the `open_file_cache` setting and helps Nginx cache information like file sizes and 404 problems.

Gzip

Compressing your content is a tradeoff between consuming processor cycles and reducing the bytes sent over the wire.

If you're going to enable gzip then you should make sure that you're only enabling it for files that are likely to be compressed. Set a minimum size for files and also ensure that you're only compressing text format files.

```
gzip on;
gzip_vary on;
gzip_comp_level 6;
gzip_min_length 10240;
gzip_proxied expired no-cache no-store private auth;
gzip_types text/plain text/css text/xml text/javascript application/x-javascript\
 application/xml;
gzip_disable "MSIE [1-6]\.";
```

The compression level ranges from 1 (low) to 9 (high). Using a high compression level consumes more processor cycles and should result in a better compression ratio.

The `gzip_min_length` option is the minimum number of bytes that a file must be before it's compressed.

Configuring PHP5-FPM

PHP5-FPM works by maintaining a pool of processes which are used to serve requests. It supports a number of ways to configure how these processes are spawned through the config file(s) found in the `/etc/php5/fpm/pool.d` directory. Obviously the location of the files may be different on your distribution of Linux.

It is possible to have more than pool running on the server and if you are running multiple sites on the same server it is advisable to use a dedicated pool per site. Not only does this provide a measure of additional security but it also means that your Opcache memory is not shared by all of your sites.

Out of the box PHP5-FPM will run a single pool of processes that dynamically grows and shrinks according to the demands of your application. The settings in the pool configuration file that control this are:

Setting	Controls
pm	Choose how the process manager will control the number of child processes. Possible values: static, ondemand, dynamic. This option is mandatory.
pm.max_children	The number of child processes to be created when pm is set to static and the maximum number of child processes to be created when pm is set to dynamic. This option is mandatory.
pm.start_servers	The number of child processes created on startup. Used only when pm is set to dynamic.
pm.min_spare_servers	The desired minimum number of idle server processes. Used only when pm is set to dynamic. Also mandatory in this case.
pm.max_spare_servers	The desired maximum number of idle server processes. Used only when pm is set to dynamic. Also mandatory in this case.
pm.max_requests	The number of requests each child process should execute before respawning. This can be useful to work around memory leaks in 3rd party libraries. For endless request processing specify '0'. Equivalent to PHP_FCGI_MAX_REQUESTS. Default value: 0.

Each child process will use up a certain amount of memory and provide the ability to handle connections. We can find out the average amount of memory that your processes are taking up by using the command `ps -ylC php5-fpm --sort:rss`. It will output something like this:

```
$ ps -ylC php5-fpm --sort:rss
S   UID   PID  PPID  C PRI  NI   RSS    SZ WCHAN  TTY          TIME CMD
S     0 22354     1  0  80   0 17540 119471 ep_pol ?       00:00:05 php5-fpm
S    33  5673 22354  0  80   0 36188 120588 skb_re ?       00:00:00 php5-fpm
S    33  5663 22354  0  80   0 38708 121270 skb_re ?       00:00:00 php5-fpm
```

The RSS column shows the "Resident Set Size". This is the amount of memory allocated to the process and that is in RAM right now as opposed to being swapped out. It shows the memory measured in kilobytes, so in my example about we can see that my processes are taking 36-38MB of memory.

Now that we know more or less what a process requires we need to work out how much memory we can allocate to PHP. Take a look at the other services you're running on the machine and how much memory they need, how much RAM you have in total, and then leave a healthy margin of free memory. That should give you an indication of how much extra RAM you can allocate to PHP, and so an idea of how much to increase your max_children value by.

Cookieless Domains in Nginx

Cookies are associated with a domain and are sent by the web server to the browser. Cookies contain pieces of information that the server needs, for example a session identifier or something similar. The server doesn't need this information for every resource request though. It can probably serve something static like an image or piece of Javascript without knowing anything about the cookie.

When it makes a request the browser includes all the cookies for the domain, regardless of whether the cookie is needed for the particular file it is requesting. This means that every request and response has the overhead of adding the cookie to it.

Cookies are attached to the domain so if we want to avoid sending them we need to make our request to a *different domain*, the cookieless domain. Unless your application sets a root domain cookie (or uses an application that does) you can use a subdomain of your normal domain. In other words makes sure that only cookies for www.your-domain.com are set, and not for the top level your-domain.com.

Edit your DNS to add a new CNAME and point it to your server. Lets call the subdomain "static" which is a very common and popular choice. We'll set up the server block in Nginx to include a location block like this:

```
server {
    listen       80;
    listen       [::]:80;
    server_name  static.your-domain.com;

    ....
    location ~* \.(jpg|jpeg|gif|css|png|js|ico|svg|txt|woff|ttf|eot)$ {
        access_log off;
        expires    max;
        add_header Pragma public;
        add_header Cache-Control public;
        fastcgi_hide_header Set-Cookie;
    }
}
```

Using a cookieless domain also allows clients to download static resources in parallel to your pages. You could even consider sharding your static assets over multiple subdomains to allow further parallelization.

Tuning MySQL

I think it's important not to get too bogged down in the thinking that MySQL is the only database to use with PHP. So instead of launching straight into discussing how to tune MySQL I think we should first take a look at some alternatives. You might not need to be tuning your database if there is a better tool for the job at hand.

After we've decided that MySQL definitely is the right tool for the job, we'll look at some of the settings that can affect your performance.

Alternatives to MySQL

In my opinion MariaDB is a better alternative because Oracle's efforts on MySQL seem to be focused more on providing commercial solutions than on the open source server such as their cluster offering[55] while lagging a bit behind on offering new features and patches for MySQL.

If you don't believe me compare the new features in MariaDB that have been released with the features released in MySQL since Oracle bought it. MariaDB offers NoSQL storage engines, just as an example.

MariaDB is a drop-in replacement for MySQL and I've never had any compatibility issues. Their website does have a complete list of the incompatibility issues so you don't have to go into testing it blindfolded.

If you're willing to make a slightly bigger change then I believe PostgreSQL is worth exploring. There is an excellent article on Wikivs[56] that explores the differences, but my experience with Postgres was that it was fast and felt a lot more professional than MySQL. The security model felt a lot tighter and the syntax and column types felt more compatible with MS-SQL and Oracle. I set up a range partition to store UK postcodes and was impressed at how quickly I could query it. I was using the earthdistance plugin to calculate the distance between UK postcodes and found that with careful indexing I was able to get some pretty great performance. I didn't get the chance to set up a Postgres cluster, so I can't comment on it at that sort of scale.

Perhaps an easier way to cluster is to use a service such as one that Amazon offers. Aurora[57] offers support for clusters right out of the box. An advantage of using Amazon is that it offers push button scaling by adding or removing servers to the cluster. It's compatible with MySQL and you don't have to worry about the fiddly bits of making the cluster.

[55] https://www.mysql.com/products/cluster/features.html

[56] https://www.wikivs.com/wiki/MySQL_vs_PostgreSQL

[57] https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Aurora.html

With reserved capacity Amazon pricing isn't nearly as expensive as you'd think after playing around with on-demand instances.

Lastly we should look at NoSQL options. I'm very fond of MongoDB which was ridiculously easy to set up a cluster with. It's not a complete replacement for an ACID compliant relational database but we were using it to guzzle down a firehose of traffic data. It provided some really nice durability features right out of the box and supports sharding natively.

The process of tuning

For me the process of tuning always begins by reading extensively on what the setting does. There is often conflicting information available, but the Percona blog[58] is a pretty reliable and consistent source of information.

Having decided what to change I try and establish how I will measure the effect. Benchmarking is only effective to a point and real life traffic is difficult to emulate properly. Some effects will take longer to be measured than others, so the period of observation needs to be understood.

Be careful of spurious correlations, for example a high cache hit ratio is not correlated to the the size of the cache. Rather it is related to the type of work being performed.

MySQL settings

I'm going to list some of the common tweaks to MySQL and can recommend that if you're going to commit to this platform that you find a manual such as High Performance MySQL[59] published by O'Reilly.

Default Engine

InnoDB is the best choice for most tables. It has row level locking which is a big bonus for concurrent workloads. If you're using MyISAM you need to make sure that you have a valid use case for it.

innodb_buffer_pool_size

This setting controls how much memory MySQL will use for caching table data. It's particularly important for read heavy servers (your cluster slaves) and less important for write heavy servers (your cluster master).

[58] https://www.percona.com/blog/

[59] http://amzn.to/1X2lesx

You can set this value to a high percentage of your system memory, assuming your server is devoted to MySQL of course. You don't want to use so much memory that swapping starts happening, but you also don't want to have lots of spare memory on your machine.

Start by looking at your total system memory.

You need to make sure that you provide enough memory for the operating system, for MySQL's other memory requirements, and caching InnoDB log files.

Subtract the amount you need from your total memory to give a rough size of the pool you can set.

Add a small percentage of the pool, say 5%, for the overhead of maintaining the buffer and then round down the value you've arrived at. This should give you the innodb_buffer_pool_size.

A very large buffer pool will take a lot of time to shut down and warm up after rebooting.

max_connections

I'm going to throw in a note about persistent connections here too. Using persistent connections reduces the overhead of connecting to the database every time your script runs. It does come with some problems, such as SET values being sticky or transactions spanning multiple requests if you don't commit them properly.

Overall it does improve performance to use persistent connections, particularly if your database is not on the same machine as your web server. If you keep in mind the need to leave your SQL connection in a consistent state at the end of your application you should consider using persistent connections.

Your max connections setting limits how many simultaneous connections to the database can be made. You'll need at least one database connection per FPM process. Take the `pm.max_children` setting from your FPM pool(s) configuration as a start for calculating how many database connections you will need.

If you're running your application on multiple servers multiply the max_children setting by the number of servers in your web cluster.

Setting a very large value for max_connections will waste memory, rather use something that is as close to your theoretical maximum connections as possible.

query_cache_size

The MySQL cache has a few gotchas and before enabling it you should carefully examine whether it is going to be useful to you.

The first gotcha is that the cache is emptied on a per-table basis for any write operation on the table. So obviously your cache is going to be a lot more useful if your application is relatively light on writes.

Remember that a write to the table invalidates all of the cache for that table. This means you'll have more cache misses, but also that MySQL is performing extra work in emptying out the cache.

You are able to select which tables should be cached (see here[60]) with syntax like this:

```
SELECT SQL_NO_CACHE * FROM volatile_table;
```

or

```
SELECT SQL_CACHE * FROM table_you_mostly_read_from;
```

Just using the cache takes CPU effort for MySQL, and you need to make sure that your ratio of cache hits versus misses is enough to compensate for the work MySQL puts into maintaining the cache.

The next gotcha is that the query cache is a single mutex, or lock. When you have concurrent processes, such as you would expect in a multi-core server, you may find that processes are waiting for the lock to become available in order to access the cache. This becomes more of a problem with higher values for your cache size.

Generally speaking you should leave the cache query disabled (the default state) unless you're sure that your application load is suitable for it. If you do enable it, don't set the value too high as this can aggravate problems with contention.

innodb_io_capacity

I've always used Amazon for any serious database which allows you to easily specify provisioned IOPS. It also manages bursts for you with their credits system.

The default value is very low, and can be increased quite substantially depending on your hardware. If you set it too high then you might find that your system performance degrades as MySQL starves out the OS.

Using the memory storage engine

MySQL offers the MEMORY storage engine[61] which creates tables with contents that are stored in memory.

You obviously do not want to use a memory storage engine for data that you want to persist in the event of your server crashing or failing.

However memory tables are absurdly fast to access and perform brilliantly for lookups. You can populate them by copying data from a persistent table when your server or application starts up.

[60] https://dev.mysql.com/doc/refman/5.7/en/query-cache-in-select.html
[61] https://dev.mysql.com/doc/refman/5.7/en/memory-storage-engine.html

One application I work on is heavily dependent on an "inputs" table which stores static information about clients. We use the data to customize the page for the user and pre-populate their order form. We run the application in campaigns and typically have around a quarter of a million entries in the table. Storing the whole lot in memory uses only 50 megs and makes it lightning fast to get the data we need to render a page.

Content Delivery Networks

A content delivery network is a network of servers that is spread over the world geographically. When a user wants a piece of content they are directed to the server that is closest to them. This reduces latency as they have to traverse less of the network to reach the server. The load of the application is spread between the geographic data centres which improves performance and availability.

Setting up your own CDN is obviously going to be very expensive, but there are a number of providers who offer this as a service. Nowadays it's a lot more affordable to use a CDN than it used to be.

An advantage to using CDN which might not immediately spring out is the fact that you can use them as a cookieless domain. This cuts the need to send and receive cookies with your static assets.

Serving your static assets from another domain also allows the browser to make two simultaneous connections and download resources in parallel. Having a CDN with multiple sub-domains will further improve this.

Using a CDN reduces the load on your application web server. It will no longer have to deal with all of the requests associated with your application that can be handled by an edge server. The CDN provider will worry about load balancing and scaling the edge locations.

Amazon Cloudfront

Amazon Cloudfront is dead simple to set up, integrates with your S3 bucket, is PCI DSS 1 compliant, is included in the Amazon free trial, and can enforce encrypted traffic between the CDN and your origin server. It's backed by the Amazon network which has an extensive global reach of edge locations.

Amazon pricing is competitive, and you can lower your bill by reserving capacity and selecting a smaller set of edge locations to deploy to. For example if you don't have many users in America you don't have to deploy to that region.

Amazon creates a subdomain for your CDN and handles all the routing for you. By default the subdomain is able to accept HTTPS connections and you don't need to configure anything.

Instead of using the CloudFront subdomain you can create CNAME's for your domain and point your CloudFront CDN to those subdomains. If you plan on doing this you'll need to bring your own SNI certificate for your subdomains.

You don't have to host your other services with Amazon to use any one of it's separate services. You can use CloudFront and host your instances anywhere else, for example. I've found, however, that the really exciting part of using Amazon is the synergy you get from joining your products together.

You can try CloudFront for free in your free trial year when you open an AWS account.

Akamai CDN

Akamai is a very big player in the CDN game. It serves 15 to 30 percent of all Web traffic, which is a staggering amount of traffic. Akamai offers a number of additional products which optimise page load speed and help to protect your application from DDOS.

Akamai provides an API that lets you manage your application so you can get much finer grained control than is possible with Amazon CloudFront.

I've never used Akamai and I can't spot a place to sign up for a free trial and play.

Using a CDN

Using a CDN is very simple and requires only minor code changes. You're going to need to be able to quickly change the references pointing to your static assets. Whenever your site loads a static asset it needs to be pointed to the correct domain where that asset it hosted.

It might seem okay to just prefix the asset with the domain, but being able to change the sub-domain means you can change T> CDN vendor more easily.

This means editing your views to include a configuration variable in front of static asset names. There are Github packages already built for Laravel such as Vinelab CDN[62]. If you're not using Laravel you'll need to code up your own version.

The helper just needs to accept an asset name and return the full url where the asset must be loaded from. You can build in the ability to support multiple subdomains so that clients can download assets in parallel.

[62] https://github.com/Vinelab/cdn

Load Balancing

A load balancer distributes a workload amongst a group of servers arranged in a cluster. Instead of having just one machine running your application you can have copies of it running on multiple machines with the load balancer choosing which one to use to serve up a requests.

The process of adding more servers to a cluster is *horizontal scaling* which can be contrasted to *vertical scaling* which is the process of upgrading your computing resource.

The notes I made at the beginning of the book about decoupling architecture come to bear when we're discussing load balancing. Having a decoupled architecture makes it a lot easier to replicate individual pieces of your application. This allows efficiency in scaling because instead of replicating your entire architecture you can replicate just the bits that are bottlenecks.

We've already discussed how to use Memcached or Redis as a session store. By having a centrally shared session storage area it no longer becomes important which server the load balancer passes the request onto since they can all retrieve the session. This makes it easier for the load balancer to evenly distribute the load.

We've also looked at how to set up cookieless domains and we've looked at clustered databases.

Load Balancing Algorithms

There are several different ways for your load balancer to distribute requests between the servers in your farm.

The simplest algorithm is the **Round Robin** method. When using this algorithm the load balancer will rotate sending requests through the list of servers in your farm. This is a fairly naive algorithm as it does not consider anything about the server, such as its response time or how many connections it is already busy with.

A refinement to Round Robin is **Weighted** load balancing. This algorithm lets you assign a weight to servers which the load balancer takes into account. More requests are sent to servers with a higher weight.

The **Least Connections** algorithm favours sending requests to servers that have the least amount of current connections. The aim of this algorithm is to send requests to servers that are less busy than others.

A **Sticky Session** algorithm will send requests from the same client to the same server. This means that one server can handle the session and you don't have to distribute your session store. This is not typically a good algorithm to use, because it can lead to some servers being over-subscribed while others lie fairly dormant.

Load balancers should perform health checks on the servers in the farm and stop sending requests to servers that are not healthy. This isn't so much an algorithm in itself as it is an additional consideration that all the algorithms need to take into account.

Types of Load Balancing

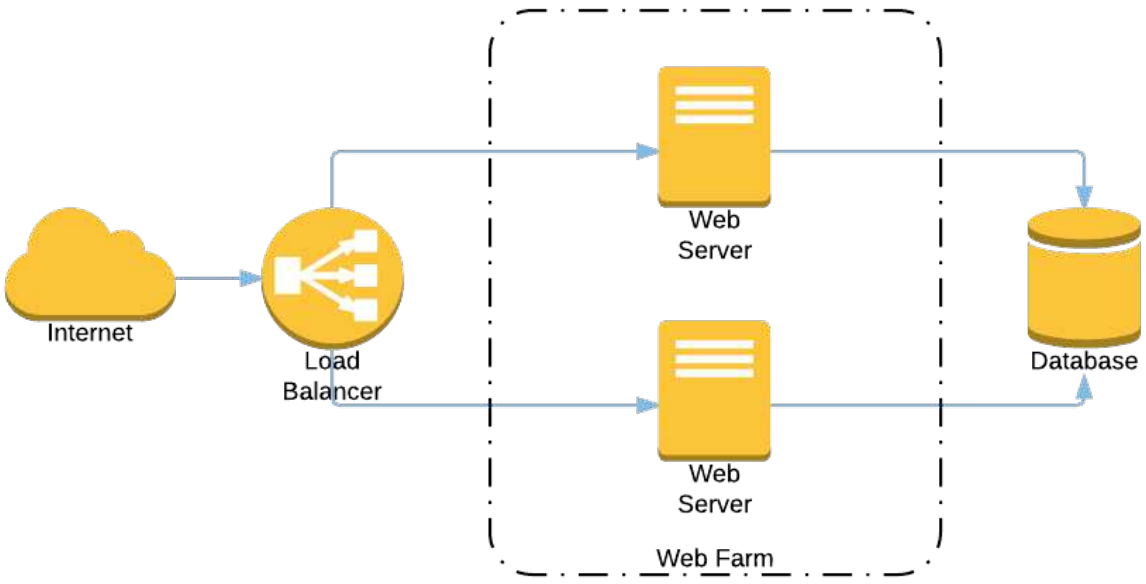
The two types of load balancing that we will discuss each focus on a particular layer of the OSI model[63]. This model splits up network communications into seven layers and the load balancing forms we look at each focuses on one of these layers.

Layer 4 - Transport Layer Load balancing

This form of load balancing focuses on the IP address and port requested. The Load Balancer will respond to requests to your site (which are mapped to your IP address and port 80) and forward them to one of the servers in your farm.

HTTPS requests on port 443 are also forwarded onto your application servers. Your servers will be responsible for terminating the HTTPS connection which means that they will need the certificate to be installed on them. This reduces the CPU requirement on the load balancer and also ensures that requests from the client are encrypted in transit on your internal network.

Your setup might look something like this diagram:



OSI layer 4 load balancer

[63] https://en.wikipedia.org/wiki/OSI_model

This form of cache requires that each server will serve the same response for each request. This means that the application must be deployed in its entirety to each server and that they must have shared access to the database(s).

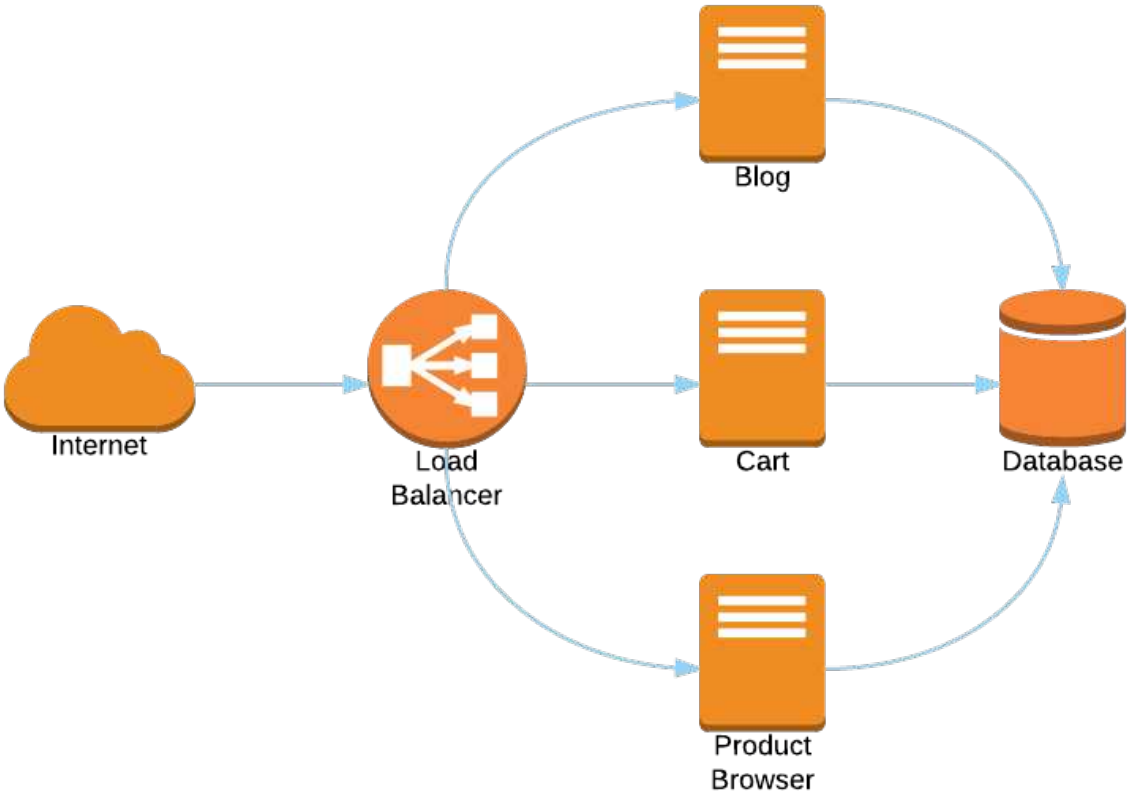
Remember back in the section on microservices we spoke about the scale cube? We can see that layer 4 load balancing is ideally suited for horizontal (X-axis) scaling.

It is not appropriate for Y-axis scaling, which leads us onto the next type of load balancing.

Layer 7 - Application Layer Load balancing

Load balancing in the application layer decides which server to send the request to based on the functionality that is required. This allows us to implement Y-Axis scaling which involves separating out the functionality of the application by server.

In the diagram below I'm showing an example of load balancing with microservices. We're using OSI layer 7 load balancing to split requests to servers based on what functionality is being requested.



OSI layer 7 load balancing

In this example if the user requests the /blog directory then they will be forwarded onto the server that handles blog requests.

Synchronising File Systems

The "hello world" application for Load Balancing in PHP is setting up a Wordpress site. We need to synchronize our file system because when you write a post on your massively scaled Wordpress blog you'll upload an image to go with it. That image must be available to every server in your webfarm otherwise a visitor will only see it if they happen to connect to the same machine you were connected to when you uploaded it.

GlusterFS

Gluster[64] is an open source distributed file system that makes it a lot easier to coordinate your file system.

Gluster uses some jargon which we will need to know:

Term	Meaning
Brick	The brick is the storage filesystem that has been assigned to a volume.
Client	The machine which mounts the volume (this may also be a server).
Server	The machine (virtual or bare metal) which hosts the actual filesystem in which data will be stored.

A brick can consume an entire disk volume or it can be placed inside an existing file system. In this example we're going to place the brick into the filesytem, but you can just as easily use a whole disk for it.

Installing it is easy. You'll need to do this on each server that you want to include in your farm:

```
1 sudo apt-get install software-properties-common
2 sudo add-apt-repository ppa:gluster/glusterfs-3.5
3 sudo apt-get update
4 sudo apt-get install glusterfs-server
```

Set up hostnames in your /etc/hosts file, the top of your hosts file should look something like this:

```
1 127.0.0.1 localhost
2 10.0.0.28 glusternode1
3 10.0.0.29 glusternode2
```

Make sure that you've configured your firewall so that the instances can reach other. On AWS I create a security group that looks like this:

[64] https://www.gluster.org

Type	Protocol	Port Range	Source
Custom TCP Rule	TCP (6)	111	10.0.0.0/24
Custom TCP Rule	TCP (6)	2409	10.0.0.0/24
Custom TCP Rule	TCP (6)	24007-24008	10.0.0.0/24
Custom TCP Rule	TCP (6)	38456-38467	10.0.0.0/24
Custom TCP Rule	TCP (6)	49152-49160	10.0.0.0/24
Custom UDP Rule	UDP (17)	111	10.0.0.0/24

AWS security group rules

Setup a spot on each of the servers where they will store the files it manages.

```
sudo mkdir /data/gluster/wordpress_sync -p
```

Choose a server to be the master (mine is glusternode1) and run this command to link up the second node:

```
gluster peer probe glusternode2
```

If nothing happens (there is quite a long timeout set) and the command exits without a message double check your firewall settings. If you're on Amazon make sure that your ACL for your VPC also allows traffic on the ports above.

On the other server run the command `gluster peer probe glusternode1`.

You should very nearly immediately get a success message, when you run the command `gluster peer status` then you should get output like this snippet:

```
peer probe: success.
root@ip-10-0-0-95:/home/ubuntu# gluster peer status
Number of Peers: 1

Hostname: glusternode2
Uuid: b01d9232-eca3-42ca-a802-e723eaa6a14d
State: Peer in Cluster (Connected)
```

Now set up the cluster with the following command:

```
sudo gluster volume create syncword replica 2 transport tcp glusternode1:/data/g\
luster/wordpress_sync glusternode2:/data/gluster/wordpress_sync force
```

Now start up the cluster by running:

```
sudo gluster volume start syncword
```

You should be up and running, you can check by running `gluster volume info`

```
root@ip-10-0-0-95:/home/ubuntu# gluster volume info
Volume Name: volume1
Type: Replicate
Volume ID: acb83f12-c88d-4b63-8b62-1ce2b51329c4
Status: Started
Number of Bricks: 1 x 2 = 2
Transport-type: tcp
Bricks:
Brick1: glusternode1:/mnt/gluster
Brick2: glusternode2:/data/gluster
```

The filesystem is created. We're going to use the servers as a client, just to show that it works. Run the following commands on both nodes:

```
sudo mkdir /mnt/synced_drive -p
sudo mount -t glusterfs glusternode1:/syncword /mnt/synced
```

Create a file on glusternode1 in the /mnt/synced_drive directory and you'll notice that it's also available in the same directory on glusternode2.

Using S3 to sychronize files

S3FS[65] lets you mount your S3 bucket into your filesystem.

The install instructions are on the Github page,

[65] https://github.com/s3fs-fuse/s3fs-fuse

```
sudo apt-get install automake autotools-dev g++ git libcurl4-gnutls-dev libfuse-\
dev libssl-dev libxml2-dev make pkg-config
git clone https://github.com/s3fs-fuse/s3fs-fuse.git
cd s3fs-fuse
./autogen.sh
./configure
make
sudo make install
```

Pop over to S3 and make a bucket. I'm calling mine wordshare.

We need to store credentials on the instance, so create a `~/.passwd-s3fs` file and paste in your details in the format `<AWS Access Key ID>:<AWS Secret Access Key>`. Now we can create a cache directory and mount the bucket into our filesystem:

```
mkdir /tmp/cache
chmod 777 /tmp/cache
mkdir /mnt/s3
s3fs -o allow_other,use_cache=/tmp/cache wordshare /mnt/s3
```

At this point the filesystem should be mounted and you can access the bucket.

Note that we mount with the option allow_other so that we are able to let Nginx read the directory.

Finally we can edit fstab to make sure that when we reboot our bucket is remounted. We use a line in fstab with this format,

```
s3fs#wordshare /mnt/s3 fuse allow_other,use_cache=/tmp/cache 0 0
```

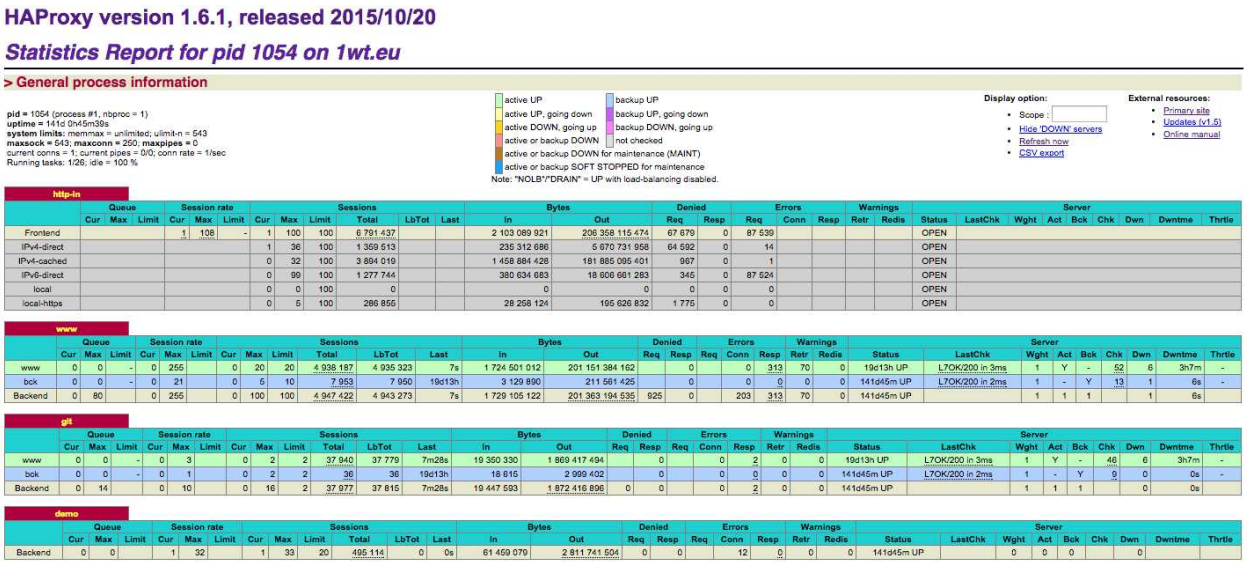
Choosing a load balancer

HAProxy

HAProxy[66] is the go to load balancer in the OSS community. It's a reliable and fast load balancer that offers a set of features that are a de facto standard for load balancing.

HAProxy is used or has been used by some pretty heavy hitters, like Github and Imgur. It's included in most Linux distribution repositories and is widely respected.

[66] http://www.haproxy.org/



HAProxy version 1.6.1, released 2015/10/20

Statistics Report for pid 1054 on 1wt.eu

> General process information

pid = 1054 (process #1, nbproc = 1)
uptime = 141d 0h45m39s
system limits: memmax = unlimited; ulimit-n = 543
maxsock = 543; maxconn = 250; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec
Running tasks: 1/26; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export

External resources:
- Primary site
- Updates (v1.5)
- Online manual

http-in

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	108	-	1	100	100	6 791 437			2 103 089 921	206 358 115 474	67 679	0	87 539					OPEN								
IPv4-direct							1	36	100	1 359 513			235 312 686	5 670 731 958	64 592	0	14					OPEN								
IPv4-cached							0	32	100	3 894 019			1 458 884 428	181 885 095 401	967	0	1					OPEN								
IPv6-direct							0	99	100	1 277 744			380 634 683	18 606 661 263	345	0	87 524					OPEN								
local							0	0	100	0			0	0	0	0	0					OPEN								
local-https							0	5	100	286 855			28 258 124	195 626 832	1 775	0	0					OPEN								

www

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
www	0	0	-	0	255		0	20	20	4 936 187	4 935 323	7s	1 724 501 012	201 151 384 162		0	0	0	313	70	0	19d13h UP	L7OK/200 in 3ms	1	Y	-	52	6	3h7m	-
bck	0	0	-	0	21		0	5	10	7 953	7 950	19d13h	3 129 890	211 561 425		0	0	0	0	0	0	141d45m UP	L7OK/200 in 2ms	1	-	Y	13	1	6s	-
Backend	0	80		0	256		0	100	100	4 947 422	4 943 273	7s	1 729 105 122	201 363 194 535	925	0			203	313	70	0	141d45m UP		1	1	1		1	6s	

git

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
www	0	0	-	0	3		0	2	2	37 940	37 779	7m28s	19 350 330	1 869 417 494		0	0	0	2	0	0	19d13h UP	L7OK/200 in 3ms	1	Y	-	46	6	3h7m	-
bck	0	0	-	0	1		0	2	2	36	36	19d13h	18 615	2 999 402		0	0	0	0	0	0	141d45m UP	L7OK/200 in 2ms	1	-	Y	9	0	0s	-
Backend	0	14		0	10		0	16	2	37 977	37 815	7m28s	19 447 593	1 872 416 896	0	0		0	2	0	0	141d45m UP		1	1	1		0	0s	

demo

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Backend	0	0		1	32		1	33	20	495 114	0	0s	61 469 079	2 811 741 504	0	0		12	0	0	0	141d45m UP		0	0	0		0		

Demo of statistics screen from http://demo.haproxy.org

It supports both layer 4 and layer 7 load balancing.

Amazon Elastic Load Balancer

To me the killer feature of the Amazon Elastic Load Balancer is the ability to combine it with auto-scaling groups. You can easily set up metrics for your webfarm and spin up instances as required. Instances can be torn down when demand drops, and you can build in some "stickiness" to avoid spikes causing you to spin servers up and down all the time.

The Amazon Elastic Load Balancer supports layer 4 and layer 7 load balancing. Amazon will take care of the requirements to terminate SSL on the load balancer, so you won't need to worry about vertically scaling your load balancer. Connections between the load balancer and your backend servers can be encrypted separately.

Varnish

Varnish has a built in load balancer but it's strength is really as a reverse proxy cache. You are able to put Varnish behind HAProxy which is a much more sensible plan than using Varnish to load balance traffic.

If you use Varnish to cache entire pages then you won't even need to hit your application to serve up pages. CMS systems like Wordpress can be dramatically improved by Varnish.

On the other hand I've had issues with Varnish and cookies, and there was not budget to refactor the application to make use of ESI.

Nginx

Nginx can also serve as a load balancer, but it only supports layer 7 load balancing. Nginx supports caching and is nearly as fast as HAProxy. I haven't used it as a load balancer and am not likely to.