CS253 Unit 6: How to Serve Millions - Scaling, caching, optimizations

Contents

Introduction

At this point in the course you know how to build almost anything. You know how to store data, create users, logins, cookies and similar. Most web applications are essentially different arrangements of these technologies. What we haven't talked about is how to run your applications at a large scale.

So far, we've just written 'toy' applications catering for just one or two users. If we want to start writing apps for thousands, or perhaps even millions, of users then obviously we'll need to think about things a little differently. That is what we are going to cover in this unit. This process is called 'scaling'.

When we talk about scaling, this may mean running your application on multiple machines, or storing huge amounts of data, or consuming large amounts of bandwidths. There are a whole range of resources that we need to think about. In this unit we'll discuss different strategies for handling these types of issues.

In particular, we will learn about caching, including the uses of caching, why you might want to cache, and specific caching implementations, like Memcache. Caching is hugely important to any successful web application. The homework will involve adding caching to your existing blog.

Quiz: Why Scale

- So that we can serve more requests concurrently.
- So we can store more data.
- So that we're more resistant to failure.
- So we can serve requests faster.

Quiz: What To Scale

- Bandwidth.
- Computers (memory, CPU).
- Power.
- Storage.

Techniques For Scaling

Let's think about some techniques for scaling.

- Optimise code.
  - cost of development time
- Add additional computers.
  - cost of hardware
  - cost of maintenance
- Upgrade computers
  - more memory
  - more disk space
  - faster CPUs
- Cache complex operations.

The first thing you should think about is optimising your code. If you have a choice between buying a second machine or optimising your code so that your app works twice as fast then this is something you should consider. There are still costs involved, whichever option you eventually select. It takes effort to optimise code, so there is the cost of development time. This has to be set against the cost of additional machines (including additional maintenance costs).

Essentially, optimising code is about programming better. This comes with experience and in time you'll be able to write better and tighter code.

Upgrading the hardware to provide more memory, more disk space or faster CPUs is often a good option, but it's one that may not be available to you if you are using a shared platform where you don't have control of the actual machines. Every couple of years, machines get substantially faster and cheaper (Moore's Law) and this can often be one of the easiest and cheapest ways to scale without the risk of breaking your website.

Another technique is to cache complex operations. We will be spending a lot of time on caching in this lecture.

Caching

Caching refers to storing the results of an operation so that future request return faster.

Basically, if you have an operation that might be slow to run, say a database query, or rendering some HTML, you store the results when you run it so that you don't have to do the computation a second time. This way, you just need to reference the previous result.

So when should we cache the results?

- When the computation is slow.
- When the computation will run multiple times.
- When the output of the computation is the same for a particular input.
- When your hosting provider charges for db access.

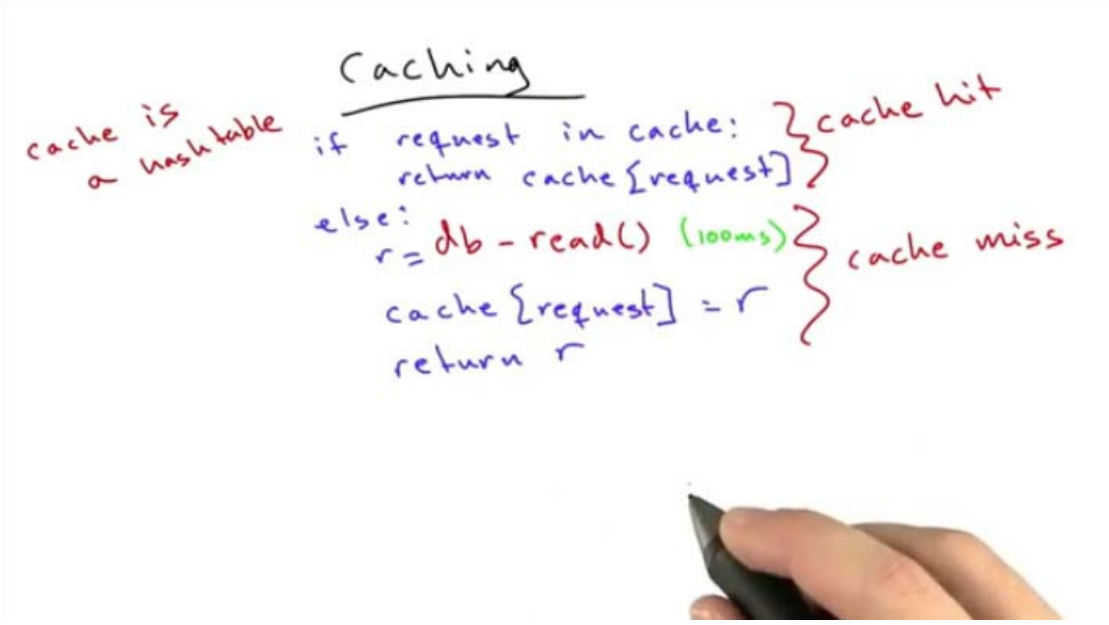
Google App Engine allows you a fixed number of free reads and writes to the datastore in a particular day. If you exceed this limit, you have to pay for it. So even if your website doesn't get a huge amount of traffic, caching requests so they don't have to hit the database over-and-over is a great way to save some money!

Let's think about how we could implement caching.

Image we have a function called db_read() that reads from the database. The function is slow. It takes 100ms to run the query. This means you can only do 10 requests per second. If you're trying to serve thousands of requests which all need to run db_read() you're database is going to get pummelled. So how would we cache db_read()?

The cache is basically a large mapping of keys to values, just like a hash table. In this case, the request is the key to the cache, so the pseudo code for hashing our function db_read() will look something like this:

```
if request in cache:
    return cache[request]
else:
    r = db_read()
    cache[request] = r
    return r
```



So, instead of calling db_read() on every request, the first thing we do is to check whether that request is in the cache. If it is, we return the cached value. This is called a cache hit. Only if the request is not in the cache do we run our query, db_read(), and this is called a cache miss. What we do on a cache miss is to store the result of the operation in the cache and then return the result. Now, in future this request will just bounce off the cache.

Now you can obtain substantial performance improvements just by wrapping this simple algorithm around your slow pieces of code. In the case of a hash table, depending on the size of the hash table we would expect to achieve a considerable improvement in speed. We might expect to retrieve the result in less than 1ms. Now, obviously, if the hash table is huge, and you're caching lots of things, you will need to take the performance characteristics of your hash tables into account, but you can still gain substantial performance improvements.

Quiz: Caching

Improve the cached_computation() function so that it caches results after computing them for the first time so subsequent calls are faster.

Scaling ASCIIChan

Let's look at how we might improve ASCIIChan.

When a user makes a request to ASCIIChan we have to carry out a number of operations:

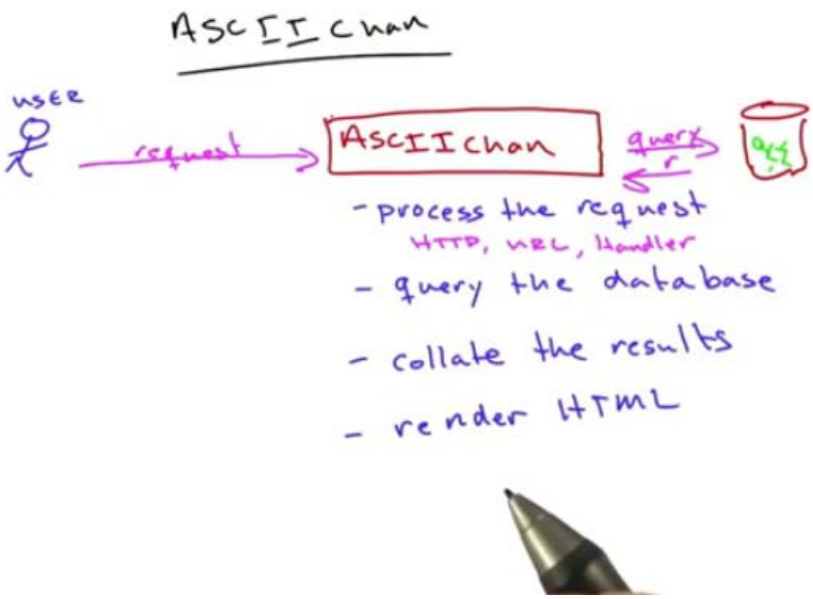
- Process the request
  - parse the HTTP
  - parse the URL
  - map the URL to the handler
- Query the database
- Collate the results
- Render HTML

The first operation takes very little time, but can still be significant if you're going to be processing large numbers of requests.

The next operation, querying the database, is much more substantial. Depending on the complexity of the query it may take a lot of time, and it may also not be free.

Collating the results also takes time. We may have do some sorting, or prune some results because the are spam. We need to convert the results into Python objects and a range of other maintenance tasks.

Rendering the HTML can take a lot of time. If there is a lot of HTML, as on Reddit for example, the time involved in rendering the HTML can be non-trivial and may require some optimisation.



Quiz: Scaling ASCIIChan

Which of these is the best place to start when we are looking to improve our website?

- Process the request
- Query the database
- Collate the results
- Render HTML

Optimizing Queries

We have already seen some of the main techniques for scaling:

- Optimise code.
- Add additional computers.
- Upgrade computers
- Cache complex operations.

When we talk about databases, we're not talking about optimising code as such. Rather we are looking to ensure that we have the appropriate indexes, making sure the query is "sane", making sure that the query is simple and that you're only querying for things that you actually need.

We don't actually need to do this for ASCIIChan, because the query itself is very simple, and because Google App Engine makes the indexes for us. However, this is always the first thing that you should check. If the index that Google App Engine creates isn't optimal for the query you're running then you may need to build something by hand.

In essence, the first step is to limit the work that the database has to do in the worst case.

Adding more machines or upgrading the machines isn't something that we really have control of in this case since App Engine takes care of all of that for us. If we weren't using App Engine then these are techniques that we would need to consider.

This leaves us with the option of caching the query to improve the performance of our application.

In ASCIIChan, the front page only changes when somebody submits a new piece of ASCII art. This makes it an ideal candidate for caching the results of the database query.

The get() function of our mainpage handler simply calls the render_front() function:

```
def get(self):
        return self.render_front()
```

The render_front() function looks like this:

```
        def render_front(self, title="", art="", error=""):
                arts = db.GqlQuery("SELECT * "
                                   "FROM Art "
                                   "WHERE ANCESTOR IS :1 "
                                   "ORDER BY created DESC "
                                   "LIMIT 10",
                                   art_key)
                arts = list(arts)

                img_url = None
                points = filter(None,  (a.author_loc for a in arts))
                if points:
                        img_url = gmap_img(points)

                self.render("front.html", title = title, art = art, error = error, arts = arts, img_url =
img_url)
```

This runs the datastore query, written in GQL, which looks up the ten most recent pieces of art:

```
SELECT * FROM Art WHERE ANCESTOR IS art_key ORDER BY created DESC LIMIT 10
```

Since most users visiting ASCIIChan only view the font page, the front page won't change very often. So we don't actually need to run the query every time someone visits. What we want to do is to cache the query.

Let's take the code that runs the query out of the function render_front() and put it into its own function which we will call top_arts():

```
        def top_arts():
                arts = db.GqlQuery("SELECT * "
                                   "FROM Art "
                                   "WHERE ANCESTOR IS :1"
                                   "ORDER BY created DESC "
                                   "LIMIT 10",
                                   art_key)
                arts = list(arts)
                return arts
```

Now, there is a technique that we can use when developing code like this that will show us when we are actually running the query. We will add the line:

```
logging.error("DB QUERY")
```

at the beginning of our function. This will print out the string “DB QUERY” in our Error Console. Now, normally you should probably use logging.debug() for this purpose, but this makes the demonstration a little easier. We also need to import the logging module.

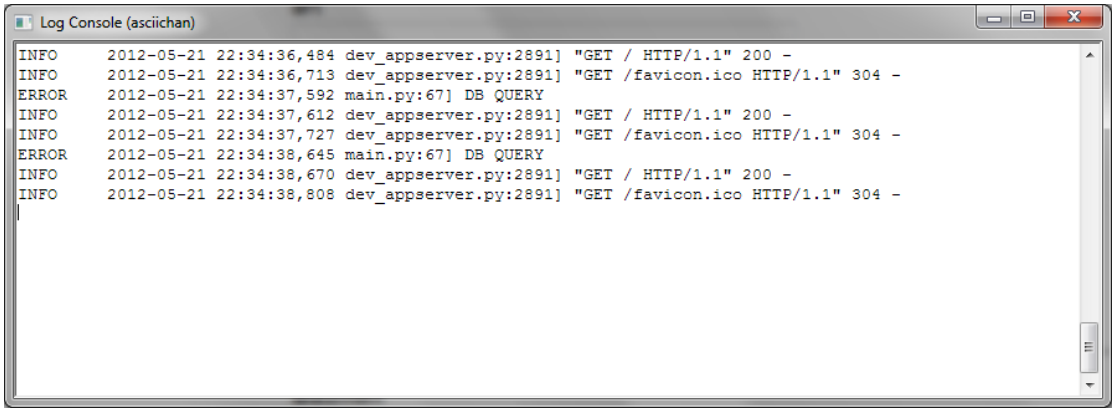
Now, when we refresh the page in the browser, we see the following in the log console:

```
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 -
```

So we are printing ERROR and DB Query, which is the string that we added, and which will be printed every time the db query runs. Then the browser fetched ‘/’, which is the actual request to ASCIIChan and also requested favicon.ico (which we haven’t created).

Each time we reload the page in the browser these lines will be repeated:

```
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 -
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 -
```



Now we will add some caching to the db query using the same algorithm we saw earlier.

```
CACHE = {}
def top_arts():
    key = 'top'
    if key in CACHE:
        arts = CACHE[key]
    else:
        logging.error("DB QUERY")
        arts = db.GqlQuery("SELECT * "
                           "FROM Art "
                           "WHERE ANCESTOR IS :1 "
                           "ORDER BY created DESC "
                           "LIMIT 10",
                           art_key)
        arts = list(arts)
        CACHE[key] = arts
    return arts
```

We have added the dictionary CACHE. We have to have a key to cache, and we are going to store it in a variable and call the key 'top'. This is how we're going to reference the result of our query in our cache. Then we just added the rest of the caching algorithm.

Now if we refresh the browser, and check the log console, we see that the db query ran and our message DB QUERY is displayed.

```
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
```

But, if we refresh the browser again, we see that the page loaded, but that DB QUERY isn't displayed. i.e. the query results were retrieved from the cache:

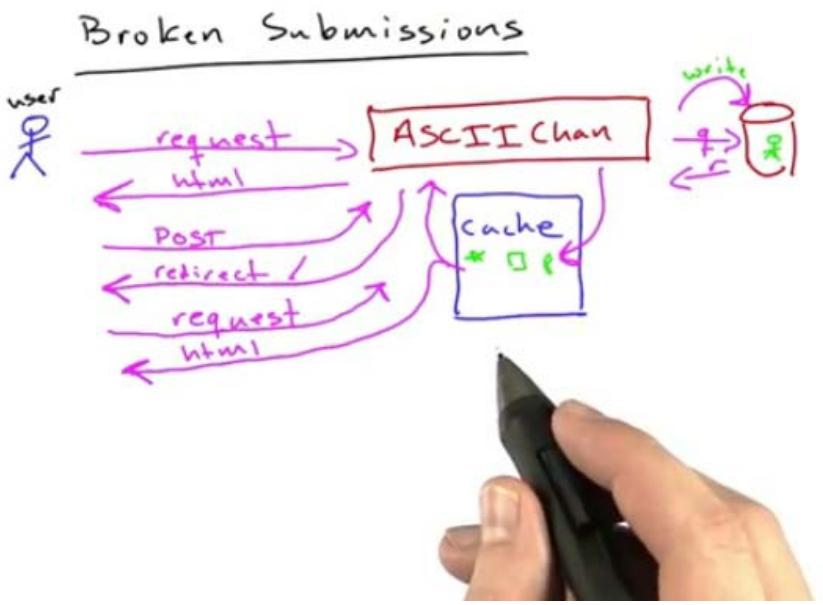
```
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
ERROR    2012-05-21 22:34:38,645 main.py:67] DB QUERY
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
INFO     2012-05-21 22:34:38,670 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-21 22:34:38,808 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
404 –
```

No matter how many times we refresh the page, DB QUERY will not appear in the console because the query results are now held in the cache.

Broken Submissions

Unfortunately, there is a problem. If we now add a new piece of art to our site it won't appear on the front page. It appears that we have broken submissions for ASCIIChan! Let's think about what just happened.

We made a request to ASCIIChan. This caused a query to the database which returned a set of results. ASCIIChan then stored these results in the cache and used the cached results to return the response to the browser. Now, when we submitted a new piece of art to ASCIIChan, the post() function wrote that piece of art to the database and redirected us back to the front page. The get() function found the data in the cache and so didn't check the database and returned the results from the cache – without the new artwork.



This is a problem. Although submissions are still working in the sense that they are being stored on the database but, because of the cache that we just added, all requests are being bounced off the cache which isn't being updated. It appears as though submissions are broken. The fact that the results in the cache aren't being updated is known as a stale cache.

Quiz: Broken Submissions

How can we fix our stale cache problem?

Choose all that apply.

- Improve the cache to automatically expire things after some time.
- After submitting, clear the cache.
- After submitting, update the cache.
- Don't cache. Find a different approach.

Cache Clearing

Now we can improve our caching technique. What we will do is to modify our post() function so that when we submit a new piece of art to ASCIIChan, it will write that piece of art to the database and then clear the cache before redirecting us back to the homepage.

Python includes a function, clear() which can be used to clear dictionaries, so all we need to do is to add this to our post() function:

```
def post(self):
        title = self.request.get("title")
        art = self.request.get("art")

        if title and art:
                p = Art(parent=art_key, title = title, art = art)
                #lookup the user's coordinates from their IP
                coords = get_coords(self.request.remote_addr)
                #if we have coordinates, add them to the art
                if coords:
                        p.coords = coords

                p.put()
                CACHE.clear()

                self.redirect("/")
        else:
                error = "we need both a title and some artwork!"
                self.render_front(error = error, title = title, art =art)
```

Note: you should always use clear() to empty a dictionary rather than setting the dictionary empty directly, i.e. you should use:

d.clear()

not

d = {}

The reason for this is that setting the dictionary empty directly can be the source of some subtle bugs.

So now we are only doing as many database queries as we actually need to do. Now there is no need for every visitor to hit the database unless they are making a new submission to ASCIIChan.

The name of the game when you're scaling websites is to only do the minimum number of database reads that you need to.

Cache Stampede

We'd like to introduce you to a new concept: the cache stampede.

Now that we have modified our cache algorithm, when a user visits ASCIIChan their request causes a query to the database which will return a set of results. these are then stored in the cache and are used in response to subsequent requests from users. When a user submits a new piece of art, the post() function writes that piece of art to the database, clears the cache and redirects the user back to the front page.

The problem comes when there are many users. Say one user posts a new piece of art which is written to the database and then the cache is cleared. If a large number of requests come into the site at the same time, while the cache is empty, they will all attempt to read from the database (with exactly the same query) at the same time. This is called a cache stampede. Now, a query which may only have taken a few milliseconds might take much longer, or even never return at all, because all of the queries are blocking each other.

- **Cache Stampede - When multiple cache misses create too much load on the database.**

Quiz: Cache Stampede

How can we avoid a cache stampede?

- Replicate the db to handle more reads.
- Only allow one web request at a time.
- Only allow one db request at a time.
- Don't clear the cache, but instead overwrite it with new data.

Let's see how we might overwrite the cache, rather than clearing it in ASCIIChan. Firstly, we need to modify our top_arts() function.

```
def top_arts(update = False):
        key = 'top'
        if not update and key in CACHE:
                arts = CACHE[key]
        else:
                logging.error("DB QUERY")
                arts = db.GqlQuery("SELECT * "
                                                "FROM Art "
                                                "WHERE ANCESTOR IS :1 "
                                                "ORDER BY created DESC "
                                                "LIMIT 10",
                                                art_key)
                arts = list(arts)
                CACHE[key] = arts
        return arts
```

What we have done is to add the parameter update to the function and set its default value to False. Now, if update is False then the function should use the cache as before, but if update is True, it should update the cache.

Now we can replace CACHE.clear() in the post() function with top_arts(True):

```
        def post(self):
                title = self.request.get("title")
                art = self.request.get("art")

                if title and art:
                        p = Art(parent=art_key, title = title, art = art)
                        #lookup the user's coordinates from their IP
                        coords = get_coords(self.request.remote_addr)
                        #if we have coordinates, add them to the art
                        if coords:
                                p.coords = coords

                        p.put()
                        #rerun the query and update the cache
                        top_arts(True)

                        self.redirect("/")
                else:
                        error = "we need both a title and some artwork!"
                        self.render_front(error = error, title = title, art =art)
```

Caching Techniques

Let's have a brief review of caching techniques.

The first technique was not to cache at all. This meant that every time the page was loaded there was a db read, but no db read when art was submitted.

Next we had what might be called naïve caching, using the basic cache algorithm. This only did a db read in the event of a cache miss, and still didn't do a db read when art was submitted, but suffered from a major bug in the form of a stale cache.

To avoid the problem of the stale cache we started clearing the cache. Again, this only did a db read in the event of a cache miss and no db read when art was submitted, but might be susceptible to a cache stampede if we are scaling the application.

We improved the technique to refresh the cache. Now we only do a db read on a page view when the cache is empty (i.e. when the app is turned on), and we do one db read each time a new piece of art is submitted to the database. In effect, a normal user browsing the site never touches the database. This is a really good property to have:

> Simple users should never touch the database.

This will improve the user experience because their requests will be handled faster. This will also keep your load down since you can have a great many of these users and because their requests are bounced off the cache you don't have to do much work to serve them.

Now, there is a fourth approach that we haven't looked at yet, and which is the most aggressive of all these techniques. We might refer to this technique as updating the cache (distinct from refreshing the cache). This can allow us to achieve the state where we never do a db read on a simple page view, and we don't do any database reads on submission either. We will look at how to implement this technique in the next section.

The techniques are summarised in the table below:

Approach	**db read/ page-view**	**db read/ submit**	**Bugs**
No caching	Every	None	
Naïve caching	Cache miss	None	Yes
Clear cache	Cache miss	None	
Refresh cache	Rarely	1	
Update cache	0	0	

More caching algorithms can be found here.

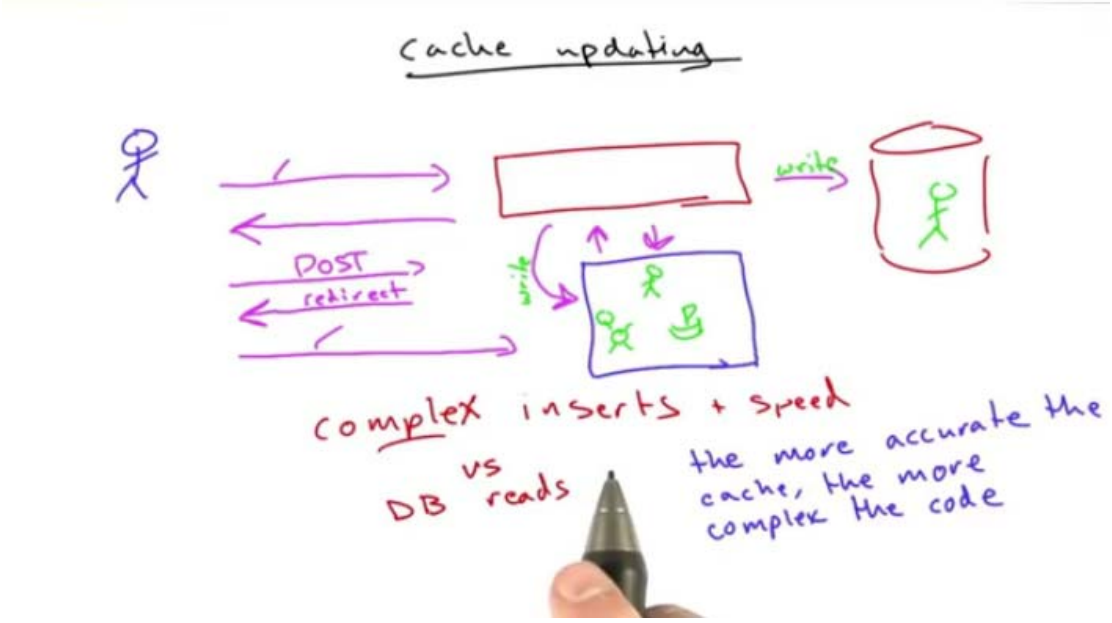
Cache Updating

With cache updating, a user viewing a page on the site is served from the cache in the same way we have seen before. However, when a user submits a new piece of art to ASCIIChan, the post() function will write the art to the database *and simultaneously write it to the cache as well*. Now this is a little more complex than the techniques we have seen so far, but it means that we never have to do a db read.

Using this technique means that the only time we ever have to do a database read is when we start up the site, and we may actually set up a program to do this for us so that no user ever has to do a database read. This is the approach taken by Reddit. Every page that you can look at is stored in its own cache. When you post a link or update a vote they then update the appropriate caches.

This effectively introduces the trade-off between complex inserts (and improved speed) versus database reads. In general, the more accurate the cache, the more complex the code will need to be.

The additional complexity is probably not justified for ASCIIChan right now. The site simply isn't at that scale. However, large sites like Reddit gain huge benefits from using this technique.



App Server Scaling

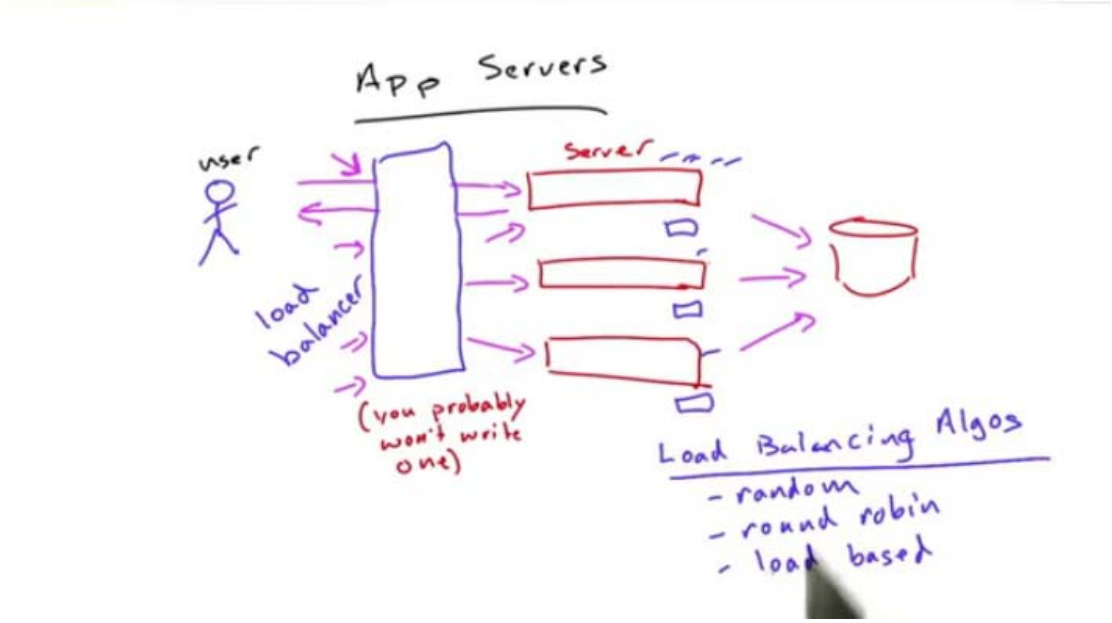
OK. So now we have taken a fair amount of load off our database by using caching to minimise, or even eliminate, database reads. Let's go back to the steps involved in a request to see what else we can do to improve the performance of our site:

- Process the request
- Query the database
- Collate the results
- Render HTML

We have improved the db query, but what can we do to improve the other steps involved in handling a request? We could certainly use caching to render the HTML. But there is another technique that we could use to improve all three other parts of the request, which is adding additional app servers.

Up until now, conceptually we have had a single program, ASCIIChan, running on a single server, that handles all of our requests. If we have so many requests coming in that a single machine can no longer handle the load, what we can do is to add multiple machines to take up some of the load. Each machine will have its own cache, and will probably interact with the database as appropriate. But how do we get requests to multiple machines?

There will be a piece of technology that sits between the users and all of our app servers. This is a physical machine, just like the servers, that is optimised for performing just one task: spreading requests between multiple machines. The load balancer will have a list of all the app servers that are available, and decides which of these the requests from users should be directed to.



All the load balancer does is to take in the request, select a server, and then pass that request along. This is why it can handle so much traffic when the application servers can't. You will probably never have to write a load balancer. If you are using Google

App Engine, or something similar, they do all of this for you, but it is good to know how they work.

There are a number of algorithms that a load balancer can use to decide which server to send a request to:

- Random – select an app server at random.
- Round Robin – requests are directed to app servers in turn.
- Load-based algorithms – allocates based on current app server load.

Quiz: Round Robin

Implement the function get_server(), which returns one element from the list SERVERS in a round-robin fashion on each call.

Caching With Multiple Servers

Why is our dictionary cache problematic with multiple app servers?

Check all that apply.

- It's not! This is a trick question.
- Multiple app servers = multiple caches. How do we keep them in sync?
- Each app server may have to hit the db to update its cache.
- We'll be caching data redundantly.

Memcached

Memcached is essentially a very fast, in-memory cache. It is a free and open source, high-performance, distributed memory object caching system which was developed back in 2003 by a company called Danga Interactive for a system called LiveJournal. This was a very popular site where people maintained their own blogs before apps like FaceBook took over the world.

These days, just about every major website on the Internet, including FaceBook, Twitter, YouTube and Reddit, uses Memcached. It has become an essential piece of software when you are writing web applications. In fact, other than Linux, Memcached is probably the piece of software that most online sites have in common.

Memcached is a process which may be run on its own machine, but which may also be run on the same machines as the app servers. The algorithm is pretty simple and essentially similar to what we have been doing with caching up to now. What is handy about it is that *all* of your app servers can interact with Memcached. Memcached is

fast enough, and can support a great many sockets so that is works well in a multi app server scenario. Also, most Memcached libraries have the built-in ability to use multiple Memcached machines.

Because Memcached is just a key-value store – basically like a giant hash table – you can hash on the keys to decide which server to send your data to. Because it is a cache, it is OK if you occasionally lose data since the authoritative copy is always held on the database. It is very common to run a little Memcached server on each machine and allow each app server to communicate with each of the caches.

So, Memcached is a very simple protocol, and the operation that it does are very simple. Ultimately, we are going to be storing values to keys, where both the keys and values are strings. The operations look something like this:

SET(key, value)

GET(key) → value

DELETE(key)

There are other operations, and there are other parameters, but these are the main basic operations. With just these basic operations, we can implement all of the caching we have been doing in ASCIIChan in a system that will scale a lot better than just having a dictionary. We also have the advantage that, if our process is restarted, we don't start with an empty dictionary.

Memcached homepage.

Memcached on Wikipedia.

Quiz: Implement Memcached

Implement the basic Memcached functions, set, get, delete and flush.

Properties Of Memcached

One of the main properties of Memcached, and one of the things that makes it different from a normal database, it that it stores everything in memory. This is what makes it so fast. Reading from memory is very fast, while reading from disk can be relatively slow. This gives Memcached a couple of properties:

- It is very fast.
- It is not durable. If you re-start Memcached you will lose all your data.
- The amount of data we can store is limited by the amount of memory that machine has.

Quiz: Properties Of Memcached

What happens when you store more data in Memcached than there is memory available?

- Error
- Throw away data that is least frequently used.
- Throw away data that is least recently used.
- Write the extra data to disk.

Memcached And ASCIIChan

So let's replace our dictionary cache in ASCIIChan with Memcache.

Fortunately, there is a version of Memcache built into the App Engine that we use on our local machines. This lets us use it when we are developing apps, and then when we deploy to App Engine, Google has it ready installed so we don't have to deal with it. The first thing we need to do is to import Memcache which is in the Google App Engine API:

```
from google.appengine.api import memcache
```

Now we are ready to replace our dictionary cache with Memcache. The current function of our cache and top_arts() function looks like this:

```
CACHE = {}
def top_arts(update = False):
        key = 'top'
        if not update and key in CACHE:
                arts = CACHE[key]
        else:
                logging.error("DB QUERY")
                arts = db.GqlQuery("SELECT * "
                                                "FROM Art "
                                                "WHERE ANCESTOR IS :1 "
                                                "ORDER BY created DESC "
                                                "LIMIT 10",
                                                art_key)
                arts = list(arts)
                CACHE[key] = arts
        return arts
```

We no longer need the CACHE dictionary, and we can delete it.

Next, in the function top_arts() we need to try to look up arts from Memcache:

```
arts = memcache.get(key)
```

Now, in the official Memcached protocol, both keys and values need to be strings. In the Memcache library that Google provides, the values can be Python datatypes which the library will convert into strings. When you fetch the data back from the cache, the Memcache library will convert it back into your Python datatype. So we can still store our art objects directly into the cache, but the keys have to be strings.

The next thing we are going to do is to modify the if statement in top_arts(). If arts isn't in the cache, the value of arts will be None, so we can say:

```
if arts is None or update:
```

If this is True, the we run the db query.

Lastly, we need to modify the line that updates the cache if we had to run the db query as follows:

```
memcache.set(key, arts)
```

So, the top_arts() function now looks like this:

```
def top_arts(update = False):
        key = 'top'
        arts = memcache.get(key)
        if arts is None or update:
                logging.error("DB QUERY")
                arts = db.GqlQuery("SELECT * "
                                   "FROM Art "
                                   "WHERE ANCESTOR IS :1 "
                                   "ORDER BY created DESC "
                                   "LIMIT 10",
                                   art_key)
                arts = list(arts)
                memcache.set(key, arts)
        return arts
```

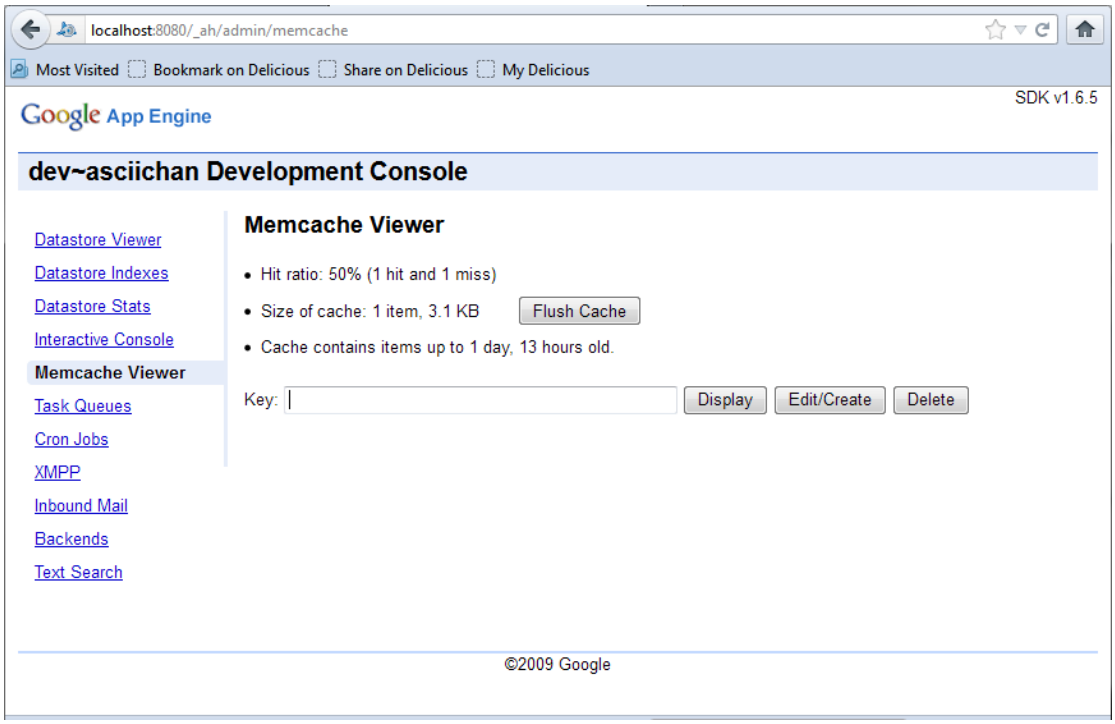
If we now refresh the browser a couple of times to reload ASCIIChan, and then check the log file we see:

```
packages\\setuptools-0.6c11-py2.7.egg-info'
ERROR    2012-05-23 11:22:27,759 main.py:67] DB QUERY
INFO     2012-05-23 11:22:27,986 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-23 11:22:28,157 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
304 –

INFO     2012-05-23 11:25:00,576 dev_appserver.py:2891] "GET / HTTP/1.1" 200 -
INFO     2012-05-23 11:25:01,019 dev_appserver.py:2891] "GET /favicon.ico HTTP/1.1"
304 –
```

When we first reloaded the page we got a db Read, as we would expect since the Memcached cache would be empty, and on the next reload there was no db read as the art was loaded from Memcache.

Now, if you're running App Engine, there is a built-in Admin tool for Memcache. If you are running App engine, you can go to _ah/admin/memcache and you will see the Memcache viewer. This shows you information about the cache and provided tools, including "Flush Cache", which are useful for helping you test the caching in your apps.



Google App Engine documentation for Memcache

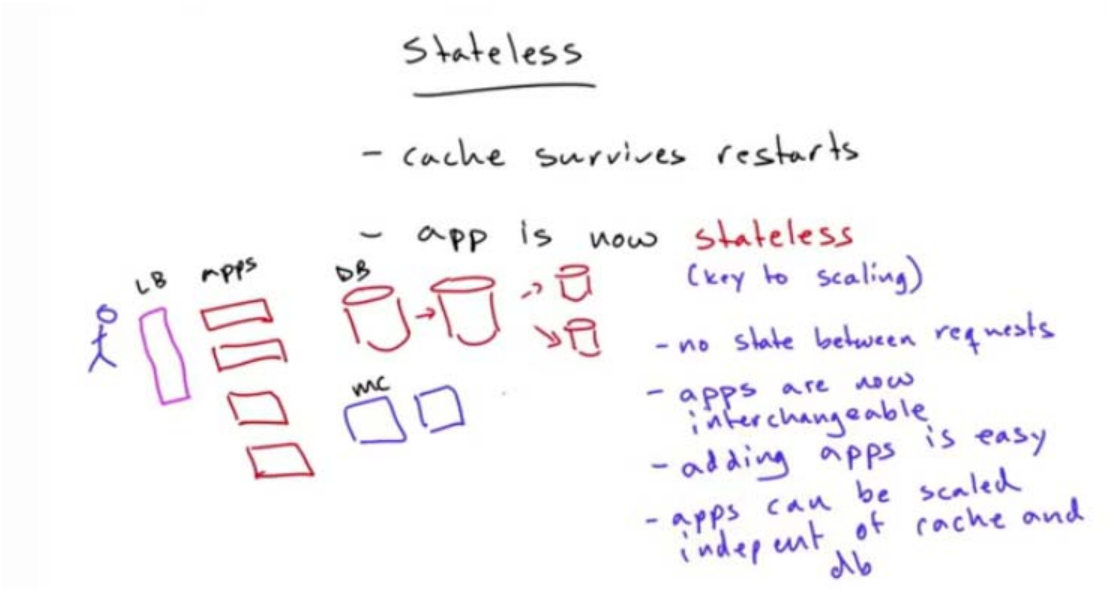
Stateless

Adding Memcached to ASCIIChan has given us two new properties which are really nice.

- The cache will now survive restarts (of the app).
- The app is now stateless.

Statelessness is the key to scaling. What it means is that the app doesn't store any state information between two requests. A number of properties stem from the fact that the app is now stateless:

- There is no state between requests
- Apps are now interchangeable.
- Adding/removing apps is easy
- Apps can be scaled independently of the cache and database.



Now, when your app is constrained in different ways, perhaps by database reads, caching, or maybe by processor requirements, any part of the system can be scaled independently of all the others. This is a large part of what scaling is about.

Advanced Cache Updates

Consider the following problem:

Multiple users submit to the app at the same time, update the cache at the same time and overwrite each others data.

What are the possible implications of this?

Imagine the scenario where we have two users, User A and User B, each being served by a different app server, and both trying to submit data to the database. The database currently holds two pieces of data, element1 and element2. User A submits element3 to the database at the same time as User B submits element4.

So far, there's no problem. The database is designed to handle this situation and it will now contain the four elements: element1, element2, element3 and element4. Now we come to update the cache, and this is where the problem manifests.

If each app server manipulates the cache directly (without communicating with other app servers) the first app server will attempt to update the cache to store element1, element2, and element3 at the same time as the second app server tries to update the cache with element1, element2, and element4. One app server will overwrite the other, and whichever data makes it to the cache will be wrong!

If we were using our earlier approach where we read from the db after data was submitted and then updated the cache from the db read we would still have a problem. The first app server will insert element3 and then read the database, getting the result

element1, element2, element3. The second app server inserts element 4, reads the db and gets element1, element2, element3, element4. Now there is ***no way to guarantee which app server will write to the cache first***. Therefore we cannot guarantee that the contents of the cache will be correct after the update.

If we redirect the user to the front page to do the update that way, the odds of the 'wrong' data being stored in the cache are even greater because of the additional delays introduced be the need to communicate with the user.

Let's look at a possible solution to this problem.

CAS

Memcache has a couple of ways of dealing with the problems we saw in the last section. The first of these is called CAS, which stands for Compare and Set. CAS adds two commands to the Memcache protocol:

gets(key) → value, unique

cas(key, value, unique) → True/False

gets() is an alternative to the get() function, and cas() an alternative to set(). unique is like a hash of the value which is unique to the specific value in Memcache.

The way that the cas() function works is that, if the unique matches the unique that you get out of Memcache it is OK to overwrite the value and return True. If you don't pass a unique, or the key isn't in the cache, or the unique has changed, then the cas() function won't set the value and will return False.

Now you have two commands that you can use to prevent two people from overwriting the same key in the cache.

The code that an app server would use to update the cache might look something like this:

```
val, unique = memcache.gets(key)
r = memcache.cas(key, newval, unique)
while r = False:
    unique, val = memcache.gets(key)
    r = memcache.cas(key, newval, unique)
```

The loop gets the new unique out of Memcache and tries to update val to newval. When r is true you know that you have successfully updated the value.

Quiz: Implementing CAS

Implement the gets() and cas() functions for our simple simulation of Memcached.

For gets(), return a tuple of (value, h), where h is hash of the value. A simple hash you can use here is hash(repr(val)).

For cas(), set key = value and return True if cas_unique matches the hash of the value already in the cache. If cas_unique does not match the hash of the value in the cache, don't set anything and return False.

Quiz: Separating Services

Why do we separate our services?

Choose all that apply.

- So they can be scaled independently.
- To increase fault tolerance.
- So two very different processes aren't competing for resources.
- So they can be updated independently.

Additional Pieces

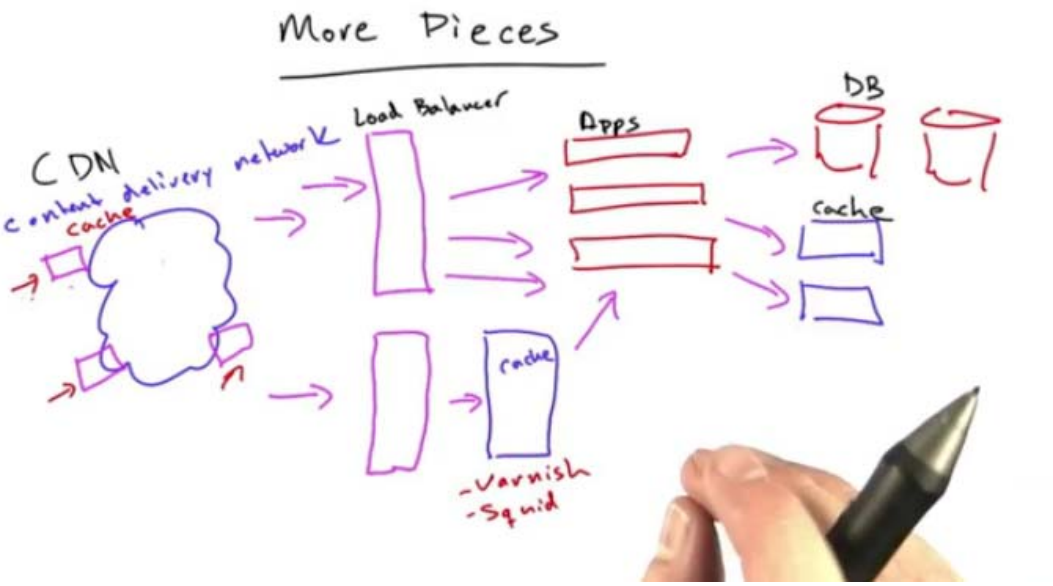
Let's close this lecture by looking at a few final pieces that would be needed for a major website, but that we haven't really talked about yet.

Firstly, what happens if we get really big, and we're receiving so much traffic that a single load balancer can no longer handle it. We can add a second load balancer, but what load balances the load balancers? What you would generally do in tis situation is something called DNS Round Robin.

DNS is the system that converts a domain name into an IP address. It is cached across the Internet with DNS machines all around the world. What you would do is, instead of mapping your site to a single IP address, you would map it to several IP addresses, one for each load balancer. This can allow you to spread potentially huge amounts of traffic across multiple load balancers.

Another thing that you may have is another cache just for HTML. This cache may intercept requests before they even reach that app servers. The HTML cache may hold HTML and images and so forth that are not going to change for a particular user or group of users (e.g. users who are not logged in). Some examples of the technology available to provide HTML caching are Varnish (used at Reddit and Hipmunk) and Squid, although many companies may well develop their own bespoke system to exactly match their requirements.

Another thing that you may use is a CDN or Content Delivery Network. These are 3rd-party companies that you pay to cache your content all across the internet. They have machines all over the world that can intercept your DNS requests and, if they have the page that your customer has requested in their cache, respond without passing the request on to your servers.



So the name of the game is caching, and the real question is at what level do you cache? The issue is how much content is actually different on each request and how much is the same. Content that is the same can be cached, and ideally, you will push that content further away from you, and closer to your customers to achieve higher speeds.

ASCIIChan 2

The final version of ASCIIChan 2 now looks like this:

```
import os
import re
import sys
import urllib2
import random
import logging
from xml.dom import minidom
from string import letters

import webapp2
import jinja2
from google.appengine.api import memcache
from google.appengine.ext import db

template_dir = os.path.join(os.path.dirname(__file__), 'templates')
jinja_env = jinja2.Environment(loader = jinja2.FileSystemLoader(template_dir), autoescape=True)

art_key = db.Key.from_path('ASCIIChan', 'arts')

def console(s):
	sys.stderr.write('%s\n' % s)

IP_URL = "http://api.hostip.info/?ip="
def get_coords(ip):
	ip = "4.2.2.2"
	url = IP_URL + ip
	content = None
	try:
		content = urllib2.urlopen(url).read()
	except URLError:
		return

	if content:
		d = minidom.parseString(content)
		coords = d.getElementsByTagName("gml:coordinates")
		if coords and coords[0].childNodes[0].nodeValue:
			lon, lat = coords[0].childNodes[0].nodeValue.split(',')
			return db.GeoPt(lat, lon)

class Handler(webapp2.RequestHandler):
	def write(self, *a, **kw):
		self.response.out.write(*a, **kw)

	def render_str(self, template, **params):
		t = jinja_env.get_template(template)
		return t.render(params)

	def render(self, template, **kw):
		self.write(self.render_str(template, **kw))

GMAPS_URL = "http://maps.googleapis.com/maps/api/staticmap?size=380x263&sensor=false&"
def gmap_img(points):
	markers = '&'.join('markers=%s,%s' % (p.lat, p.lon) for p in points)
	return GMAPS_URL + markers
```

```
class Art(db.Model):
	title = db.StringProperty(required = True)
	art = db.TextProperty(required = True)
	created = db.DateTimeProperty(auto_now_add = True)
	coords = db.GeoPtProperty( )


def top_arts(update = False):
	key = 'top'
	arts = memcache.get(key)
	if arts is None or update:
		logging.error("DB QUERY")
		arts = db.GqlQuery("SELECT * "
					"FROM Art "
					"WHERE ANCESTOR IS :1 "
					"ORDER BY created DESC "
					"LIMIT 10",
					art_key)
		arts = list(arts)
		memcache.set(key, arts)
	return arts

class MainPage(Handler):
	def render_front(self, title="", art="", error=""):
		arts = top_arts()

		img_url = None
		points = filter(None, (a.coords for a in arts))
		if points:
			img_url = gmap_img(points)

		#display the image URL
		self.render("front.html", title = title, art = art, error = error, arts = arts, img_url =
img_url)

	def get(self):
		self.render_front()

	def post(self):
		title = self.request.get("title")
		art = self.request.get("art")

		if title and art:
			p = Art(parent=art_key, title = title, art = art)
			#lookup the user's coordinates from their IP
			coords = get_coords(self.request.remote_addr)
			#if we have coordinates, add them to the art
			if coords:
				p.coords = coords
			p.put()
			#rerun the query and update the cache
			top_arts(True)

			self.redirect("/")
		else:
			error = "we need both a title and some artwork!"
			self.render_front(error = error, title = title, art =art)

app = webapp2.WSGIApplication([('/', MainPage)],
		 debug=True)
```

Answers

Quiz: Why Scale

- **So that we can serve more requests concurrently.**
- **So we can store more data.**
- **So that we're more resistant to failure.**
- **So we can serve requests faster.**

Quiz: What To Scale

- **Bandwidth.**
- **Computers (memory, CPU).**
- **Power.**
- **Storage.**

Quiz: Caching

```
cache = {}
def cached_computation(a, b):
    key = (a, b)
    if key in cache:
        c = cache[key]
    else:
        c = complex_computation(a, b)
        cache[key] = c
    return c
```

Quiz: Scaling ASCIIChan

- Process the request
- **Query the database**
- Collate the results
- Render HTML

Quiz: Broken Submissions

- Improve the cache to automatically expire things after some time.
- **After submitting, clear the cache.**
- **After submitting, update the cache.**
- Don't cache. Find a different approach.

Quiz: Cache Stampede

- Replicate the db to handle more reads.
- Only allow one web request at a time.
- Only allow one db request at a time.
- **Don't clear the cache, but instead overwrite it with new data.**

Quiz: Round Robin

```
n = -1
def get_server():
    global n
    n += 1
    return SERVERS[n % len(SERVERS)]
```

Caching With Multiple Servers

- It's not! This is a trick question.
- **Multiple app servers = multiple caches. How do we keep them in sync?**
- **Each app server may have to hit the db to update its cache.**
- We'll be caching data redundantly.

Quiz: Implement Memcached

```
CACHE = {}

#return True after setting the data
def set(key, value):
    CACHE[key] = value
    return True

#return the value for key
def get(key):
    return CACHE.get(key)

#delete key from the cache
def delete(key):
    if key in CACHE:
        del CACHE[key]

#clear the entire cache
def flush():
    CACHE.clear()
```

Quiz: Properties Of Memcached

- Error
- Throw away data that is least frequently used.
- **Throw away data that is least recently used.**
- Write the extra data to disk.

Quiz: Implementing CAS

```
CACHE = {}

def set(key, value):
    CACHE[key] = value
    return True

def get(key):
    return CACHE.get(key)

def delete(key):
    if key in CACHE:
        del CACHE[key]

def flush():
    CACHE.clear()

def gets(key):
    val = CACHE.get(key)
    if val:
        unique = hash(repr(val))
        return val, unique

def cas(key, value, cas_unique):
    r = gets(key)
    if r:
        val, unique = r
        if unique == cas_unique:
            return set(key, value)
        else:
            return False
```

Quiz: Separating Services

- **So they can be scaled independently.**
- **To increase fault tolerance.**
- **So two very different processes aren't competing for resources.**
- **So they can be updated independently.**