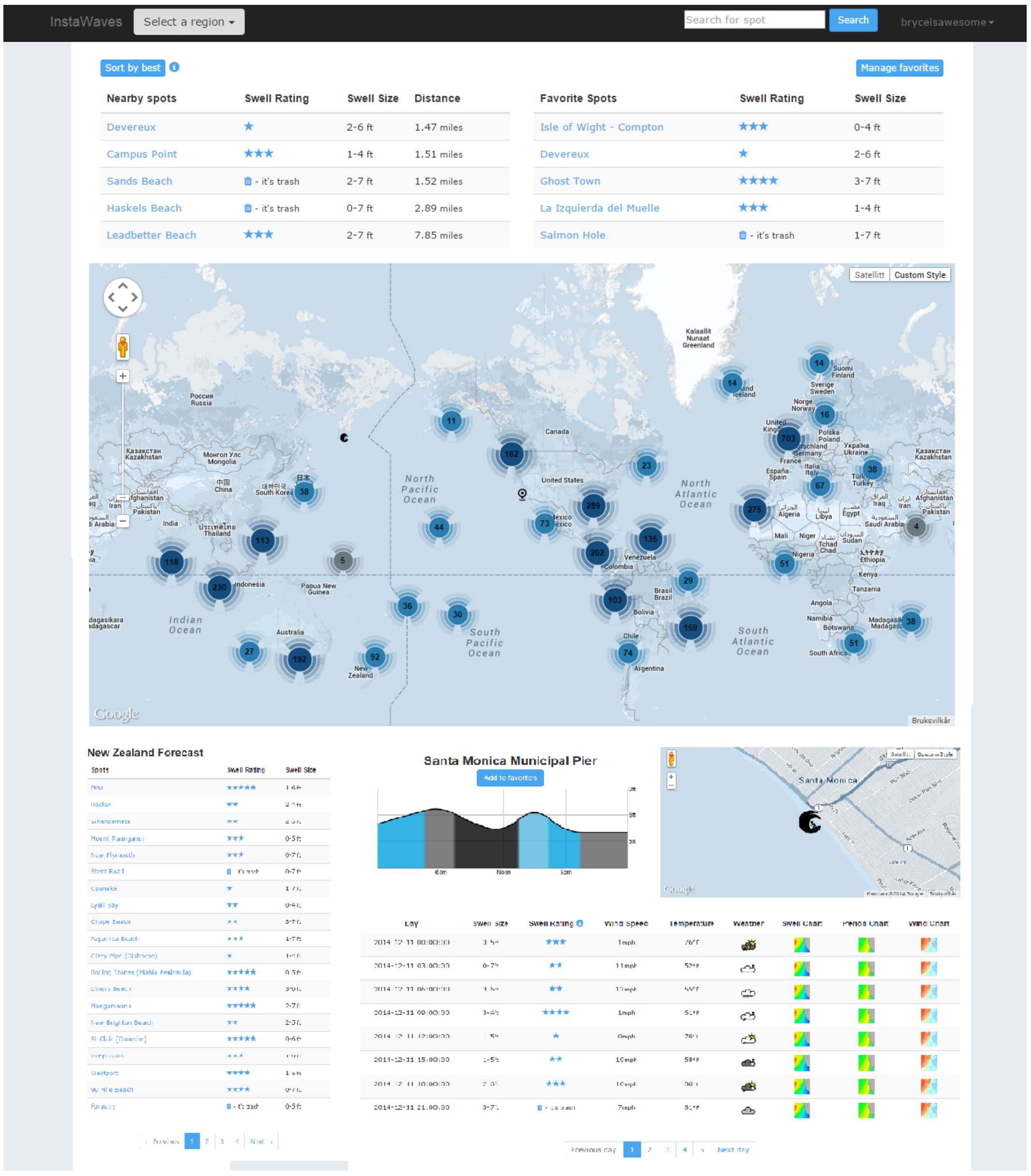
InstaWaves
Surf Forecasts Made Easy



Nearby spots	Swell Rating	Swell Size	Distance
Devereux	★	2-6 ft	1.47 miles
Campus Point	★★★	1-4 ft	1.51 miles
Sands Beach	- it's trash	2-7 ft	1.52 miles
Haskels Beach	- it's trash	0-7 ft	2.89 miles
Leadbetter Beach	★★★	2-7 ft	7.85 miles

Favorite Spots	Swell Rating	Swell Size
Isle of Wight - Compton	★★★	0-4 ft
Devereux	★	2-6 ft
Ghost Town	★★★★	3-7 ft
La Izquierda del Muelle	★★★	1-4 ft
Salmon Hole	- it's trash	1-7 ft

New Zealand Forecast

Spots	Swell Rating	Swell Size
Piha	★★★★★	1-6 ft
Raglan	★★	2-4 ft
Whangamata	★★	2-5 ft
Mount Maunganui	★★★	0-5 ft
New Plymouth	★★★	0-7 ft
Stent Road	- it's trash	0-7 ft
Opunake	★	1-7 ft
Lyall Bay	★★	0-4 ft
Ohope Beach	★★	3-7 ft
Papamoa Beach	★★★	1-7 ft
Gizzy Pipe (Gisborne)	★	1-4 ft
Rolling Stones (Mahia Peninsula)	★★★★★	0-5 ft
[illegible] Beach	★★★★	3-6 ft
Mangamaunu	★★★★★	2-7 ft
New Brighton Beach	★★	2-5 ft
St Clair (Dunedin)	★★★★★	0-6 ft
Greymouth	★★★	[illegible]
Westport	★★★★	1-5 ft
90 Mile Beach	★★★★	0-7 ft
Forestry	- it's trash	0-5 ft

Santa Monica Municipal Pier

Day	Swell Size	Swell Rating	Wind Speed	Temperature	Weather	Swell Chart	Period Chart	Wind Chart
2014-12-11 00:00:00	3-5ft	★★★	1mph	76°F				
2014-12-11 03:00:00	0-7ft	★★	11mph	52°F				
2014-12-11 06:00:00	3-5ft	★★	12mph	55°F				
2014-12-11 09:00:00	3-4ft	★★★★	1mph	51°F				
2014-12-11 12:00:00	1-5ft	★	0mph	76°F				
2014-12-11 15:00:00	1-5ft	★★	10mph	58°F				
2014-12-11 18:00:00	2-6ft	★★★	10mph	86°F				
2014-12-11 21:00:00	3-7ft	- it's trash	7mph	81°F				

Samson Svendsen
Simen Aakhus
Anders Kristiansen
Eivind Kristoffersen

Contents

1 Introduction

InstaWaves is a web application that lets users view wave information for surfing spots from thousands of locations around the world. Wave information includes size and rating for the swell, wind speed, air temperature and more. The application also includes graphics illustrating some of these conditions, in addition to maps showing all surfing spot, making it easy for users to find new surfing spots.

All of our data is gathered by making queries to the Magicseaweed API, which also is a web application providing surf data. InstaWaves differs from Magicseaweed and similar services in the way that our focus is on letting users find the best and nearest surf spots, thereby making it both easy and fast to decide where and when to go surfing. This functionality is located on the frontpage, and is available for both registered and non-registered users, thus allowing all users to find the best surf spots in a matter of seconds. Because our focus is on letting the user find a good surf spot as fast as possible, we tried to make the design as clean and streamlined as possible.

If a user registers for our application, surfing spots can be added to the user's favorites list. The user can then conveniently view their top five favorite spots on the frontpage of the application, shortening the time it takes to find surf forecast for their desired spots. The user can of course make changes to this list, by deleting any of the favorited surfing spots.

Instawaves is implemented in Ruby-on-Rails (Ruby 1.9.3 and Rails 4.1.4). The backend data store is a RDBMS, in our case MySQL. Since the purpose of Scalable Internet Services is to build and deploy a scalable web service, this report will therefore discuss the deployment, performance and scalability of InstaWaves.

2 Development

At the start of the course we were encouraged to follow an agile approach to developing. This included creating stories defining some type of chore or feature (stories) that needed implementation, and working on a portion of these in one-week sprints, using PivotalTracker. We used Pivotal Tracker for story handling and velocity tracking, which made it clear to see what was being worked on and what needed to be worked on.

We also utilised pair programming as a tool for improving our productivity as well as the quality of our code. In addition, it is a great tool for learning, seeing as this is a "learn by doing" course. None of the members of the group had much experience with TDD and we decided not to make use of it when developing InstaWaves. This was because we didn't feel it would be worth spending hours learning to create unit tests, and that those hours were better spent elsewhere.

For the external repository we used Github
(github.com/scalableinternetservices/BaconWindshield) combined with Git as revision control for handling our local repositories.

3 Application Architecture

Through the use of scaffolding in Rails, InstaWaves was set up to have a model-view-controller (MVC) architecture. This gave us a clear framework of how the application data flows and how the routing should be. As an example, when any of our views get a request, the MVC architecture ensures that the routing finds the correct controller, which in turn interacts with its respective ActiveRecord model that contains the application data, and makes queries to the database. The controller then invokes the view, which is rendered in the browser.

InstaWaves has five models and the relation between them looks like this:

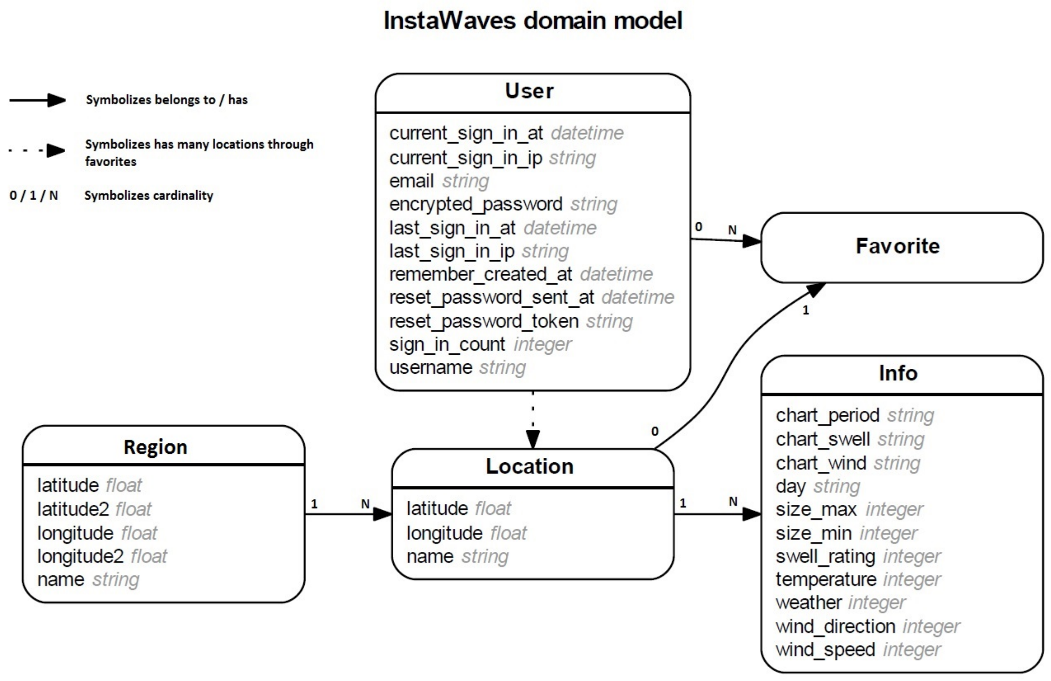


Figure 1: Class relations

4 Scalability Testing and Optimization

4.1 Critical User Paths

In order to test the scalability of InstaWaves, we first had to define the application's critical user paths. For our application we identified a path that includes seven get- and post requests that are all quite vital for our application,and tests were required for each and every one of them. The path starts off by visiting the front page. The next step is visiting the sign-up page for new users, and this is followed by an actual sign-up (which automatically signs the user in). Next up the user will try to find a location. This is done by visiting an arbitrary region that lists all the regions locations. The user will then visit an arbitrary location, which shows detailed wave information. At this point the user makes use of the favorite feature, by adding the said location to the user's list of favorites. The user then returns to the frontpage and the path is concluded when he/she makes use of the manage favorites functionality on the front page, which redirects the user to the user's list of favorites.

This path simulates a lot of people signing up for our applications at the same time. As shown later in this report we can see that user creation is actually one of the most time consuming operations of our application. Under normal load we would never see this many user creations happening at the same time. Some people would already be logged in, others would only need to log in and some users would not sign up at all. This path can therefore be seen as a kind of worst case scenario for testing purposes.

Our read only version of the critical path will be more closely related to average use of our application, and should also produce better response times in our tests.

4.2 Funkload As a Testing Tool

For testing the performance of our application we used an instance on EC2 with Funkload installed. This way the data transfer times between the Funkload instance and app instance will be minimized. To start with we configured the Funkload instance size to be m1.xlarge. We did this to prevent any bottlenecking occurring because of Funkload. However, all the tests when scaling both horizontally and vertically yielded the same results, i.e. we were not able to see any improvements at all, no matter how many concurrent users were simulated. We think that the problem might be that the requests are being queued up on nginx. A possible explanation for this might be that our passenger pool size is smaller than the nginx queue size. We tried to look at the possibility for changing this, as suggested by the instructors of the class, but we were not able to remedy this problem.

However, we were able to walk around it by changing our Funkload instance to m1.medium. Since it only has one CPU we think that the requests are not fast enough to be queued up on nginx. The only problem was that we were limited to testing a maximum of 100 concurrent users because the CPU usage of the Funkload instance neared 100%.

4.3 Initial Performance Testing

Our initial testing was performed on a single m1.medium instance with 3356 locations, each with 24 wave objects, distributed between 93 regions. This is a fairly large dataset, which initially lead to really slow test results. It should be mentioned that when we performed testing on our

EC2 instances, we used a randomly generated dataset for the wave information. This was due to the fact that seeding the database with actual information requires a lot of requests to the Magicseaweed API. This lead to troubles with timeouts when creating stacks, because seeding the database would take up to one hour.

The testing revealed some significant performance hiccups. Perhaps most notably, the search function slowed down to a halt when getting a lot of matches. We performed a search for “a” which resulted in 2697 results being shown. According to our development log the page used 63287 ms to complete the request, where 57969 of them were spent in ActiveRecord.

The initial Funkload testing showed that other pages were also in need of improvements. Below are the slowest average response times during the best cycle with 2 concurrent users: Getting a region page: avg response time: 8.58s Getting root url: avg response time: 6.59s

A response time this slow is generally unacceptable, and our application would seem rather sluggish. In general, the response times are a bit better, averaging at around 2 seconds. This is still not satisfactory, but it seems that the three biggest bottlenecks are the searching function, viewing a region page and viewing the root url. The slow response of the root url would be detrimental to our app, as this is the most frequently used page, and is what all users will be presented with when first accessing our site.

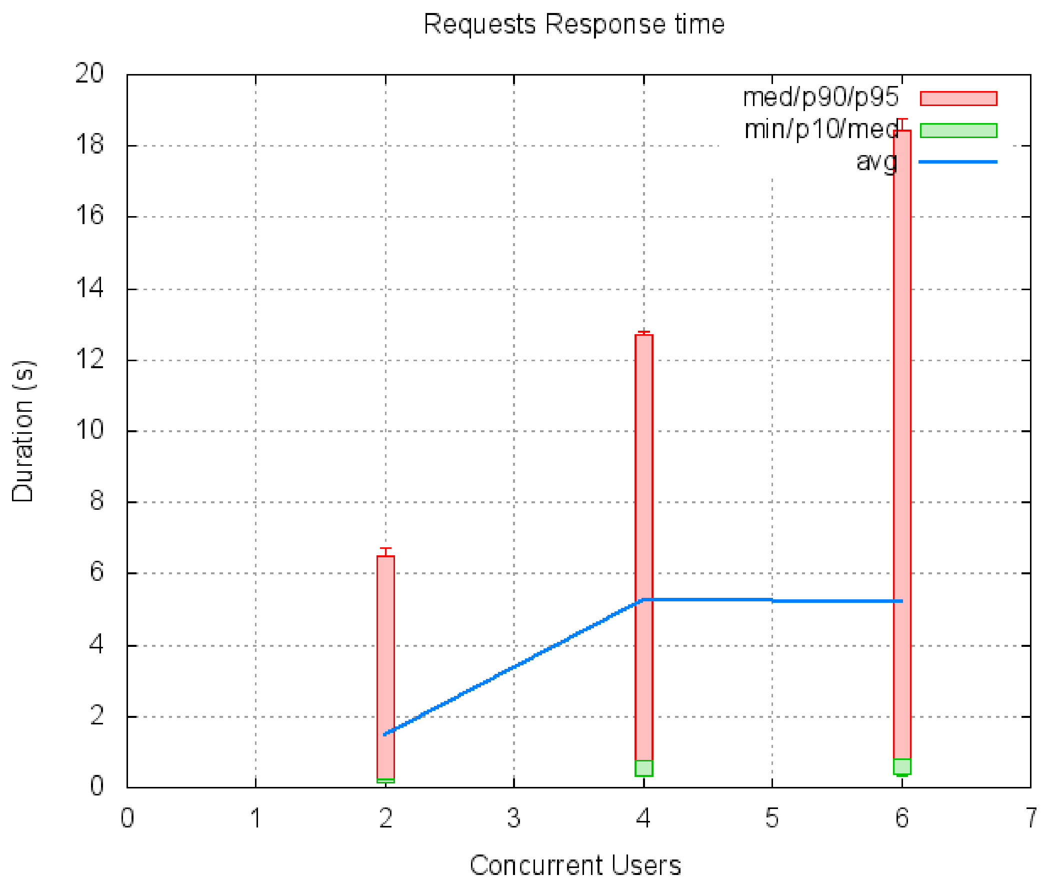


Figure 2: Graph showing rps of our initial testing.

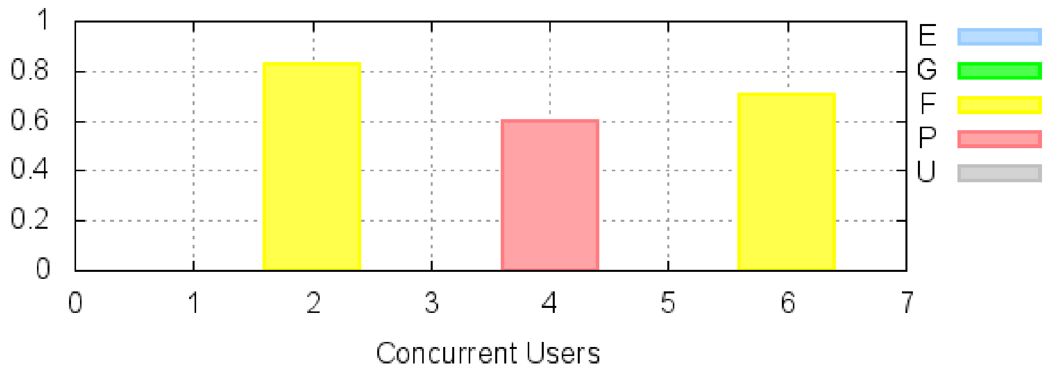


Figure 3: Apdex of initial testing.

The Funkload tests showed that the application broke down when having more than 4 concurrent users.

5 Improving the Application

5.1 Query Improvements

After our initial Funkload testing, and by examining the logs, we noticed that a lot of time was spent in ActiveRecord. The following sections describes how we chose to handle this issue.

5.1.1 Removing N + 1 Queries

When we were looking into improving the queries of our application, we found that a lot of them were subject to the N + 1 query problem. This occurs when you have one-to-many relationships in the database, and you try to nest queries. We experienced this problem when we tried to iterate through a list of locations, and for each location iterate through a list of info objects:

```
@locations.each do |location|
        #print some location information
        location.infos.each do |wave|
                #print some wave information
```

One location is picked, and for that location we iterate through N info objects, this results in more queries then we actually need, which in turn leads to a performance hiccup. To solve this problem we used a technique called eager loading. In this case this was simply done by adding .includes(:infos) to the location query:

```
@locations.includes(:infos).each do |location|
```

This performs one big query by utilizing “join”, instead of many small ones, thereby increasing performance.

5.1.2 Adding Indices

We noticed that none of our models took advantage of database indexes. By having an index on key columns in the database we can significantly reduce the amount of time spent doing reads of our database. The tradeoff is marginally slower writes, but since our application is mostly based on reads this should not affect the application negatively in any significant way. We implemented indexing on all the foreign keys, as well as location names, and saw drastic improvements in query times throughout the entire application.

5.1.3 Results of Improved Queries

All numbers are taken from the production log of our application.

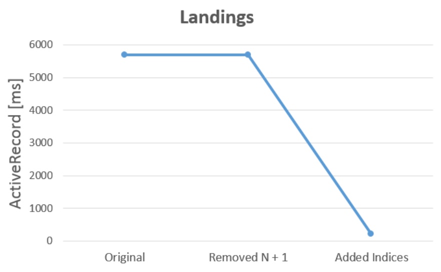


Figure 4: Time spent in ActiveRecord when displaying index page

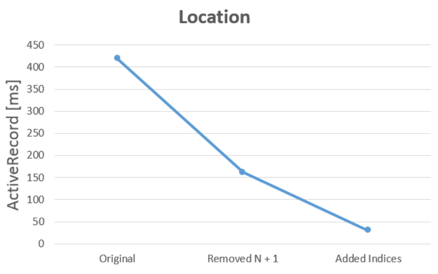


Figure 5: Time spent in ActiveRecord when displaying location page

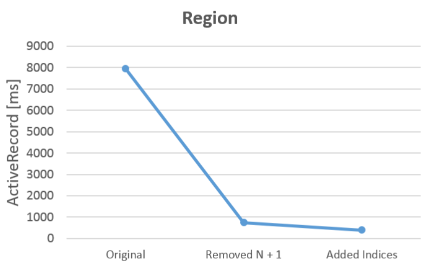


Figure 6: Time spent in ActiveRecord when displaying region page

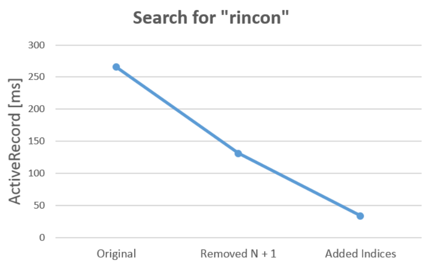


Figure 7: Time spent in ActiveRecord when searching for ”rincon”

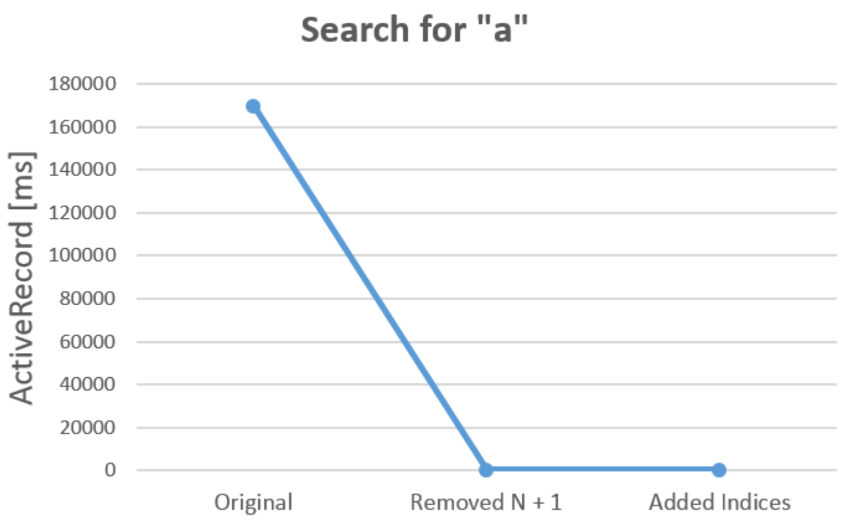


Figure 8: Time spent in ActiveRecord when searching for ”a”

Removing the N + 1 queries proved useful when querying a large dataset. This can especially be seen in broad searches, with lots of matches. We want our application to be robust, so it is important that weird and uncommon searches are fast, even though they won't happen very often.

Adding indices proved to be a significant improvement throughout the entire application, by improving the read times.

Overall improvement as a result of improved queries:

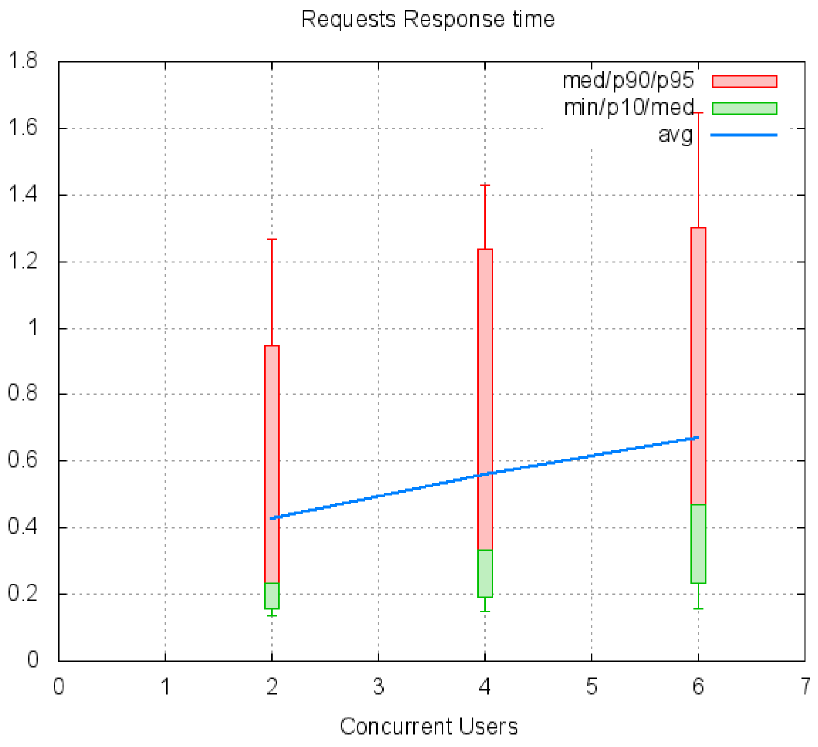


Figure 9: Overall improvement in responsetime due to fixed querries.

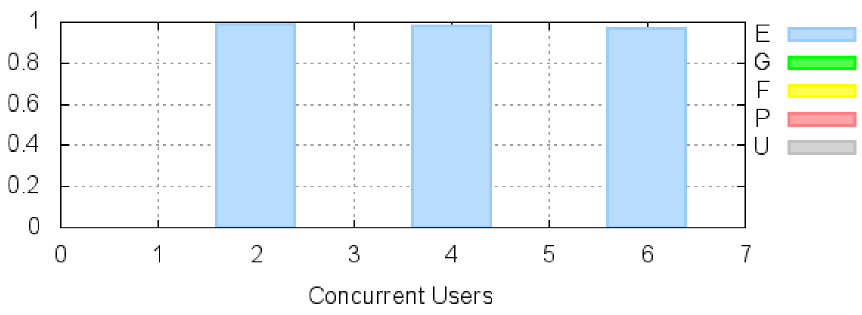


Figure 10: Apdex of improved queries.

5.2 Pagination

After fixing up the queries we see that the time spent in ActiveRecord has been significantly reduced. The rendering however, is still pretty slow. This is due to the large amounts of information being displayed on screen. By utilizing pagination we can limit the amount of data shown on screen at any given time, and thereby reducing the rendering times of the different views. In order to divide the content into pages, the pagination method uses COUNT, which is can be a slow operation. In our locations/show view we already know how many objects are going to be shown on each page, so by hardcoding the total_entries, we can prevent the COUNT from happening, shaving off a few ms.

From the production log we saw a significant reduction in render times, in addition to a reduction in query time:

	With Pagination	Without Pagination
View a Region	Completed 200 OK in 1153ms (Views: 758.2ms — ActiveRecord: 393.2ms)	Completed 200 OK in 335ms (Views: 195.0ms — ActiveRecord: 136.7ms)
Search for ”a”	Completed 200 OK in 7942ms (Views: 7551.9ms — ActiveRecord: 388.5ms)	Completed 200 OK in 200ms (Views: 157.8ms — ActiveRecord: 40.8ms)
View a Location	Completed 200 OK in 150ms (Views: 97.1ms — ActiveRecord: 30.3ms)	Completed 200 OK in 137ms (Views: 99.1ms — ActiveRecord: 26.9ms)

As we can see from this table, the greatest improvements were made when performing a search yielding a large result

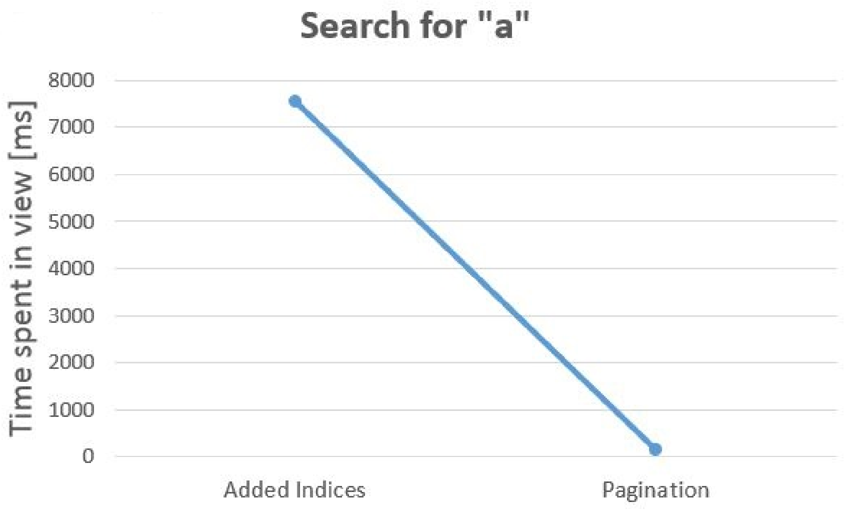


Figure 11: Improvements on search by using pagination

5.3 Caching

In a web application there is a lot of content that is similar or identical for a lot of users. Since a lot of users share the same data we can prevent database queries, as well as speed up rendering times by reusing information across multiple users. This can be done by caching. Caching is already implemented in rails ,so by just adding a few lines of code, the efficiency of any app can increase significantly.

5.3.1 Fragment Caching

Caching in rails can be done in several ways. Caching an entire page or action can be done in rails, but it does not work very well with a highly dynamic application. That is why we decided to add some fragment caching instead.

By caching we can also prevent our controller from performing expensive actions or queries, that only depend on the cached values. By preventing these actions we can save a lot of time.

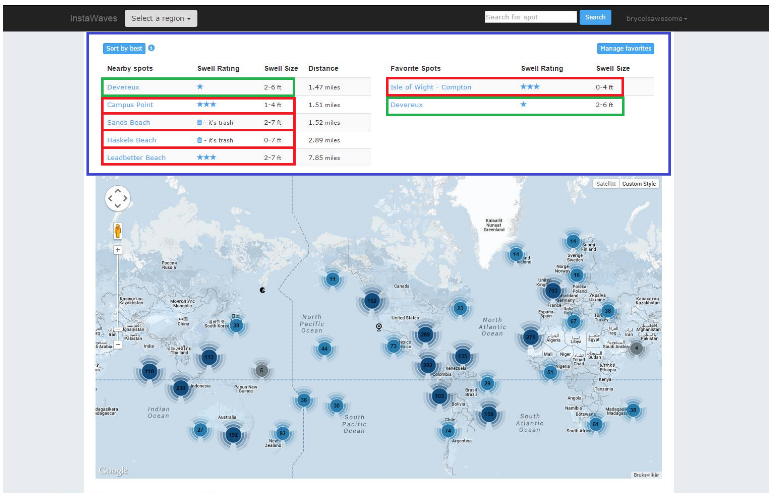
By looking at the logs of our application it became apparent to us that the navbar was rendering on every single page request, and to make the region dropdown, it was doing a lot of redundant querying. Since this dropdown is the same for each page, and is infrequently updated we decided that this was a good place to start our caching.

By caching the dropdown the amount of time spent rendering the navbar went from an average around 100 ms to an average of 2 ms. Since the navbar is rendered over and over, this should help a great deal with overall speed of the application.

5.3.2 Russian Doll Caching

Our frontpage has content that depends on what user is logged in. Different users have different locations and favorites displayed. To scale our app better we utilized some russian doll caching. First we cache the “entire” front page, or what corresponds to the blue box in the picture under. Inside this box, we cache the locations and favorites. This way if the outer cache is invalidated it will be built up by using the inner cache, and the time spent when re-writing a new cache is minimized.

```
cache cache_key_frontpage do
        @locations.each |location| do
                cache |wave| do
                        #render location information
                end
         end
 end
```



Nearby spots	Swell Rating	Swell Size	Distance
Devereux	★	2-6 ft	1.47 miles
Campus Point	★★★	1-4 ft	1.51 miles
Sands Beach	🗑 - it's trash	2-7 ft	1.52 miles
Haskels Beach	🗑 - it's trash	0-7 ft	2.89 miles
Leadbetter Beach	★★★	2-7 ft	7.85 miles

Favorite Spots	Swell Rating	Swell Size
Isle of Wight - Compton	★★★	0-4 ft
Devereux	★	2-6 ft

Figure 12: Example showing caching structure of our frontpage

We also reuse the cache fragments on the frontpage. If someone has a spot in their favorites list, that also appears in the nearbys the cached fragment will be reused. This is shown by the green boxes on the picture above. This will save some memory in the cache store.

5.3.3 HTTP caching

HTTP caching is made very easy in rails. Rails automatically provides an etag from a MD5 digest of the body. This way subsequent request made to a site with the same content responds with a 304 code, instead of the body. This is much more efficient. To avoid generating the response every time you simply need to add a fresh_when tag to the corresponding action, along with an etag. (Not sure if Funkload supports 304 not modified)

5.3.4 Memory Caching

We decided to use memcached in our application, by using the 'dalli' gem as a memcache store. Memcached stores the cache in memory instead of on disk. This way the read and write time of cache fragments will be reduced, and the memory can be shared across different servers. When the memory is filled up, memcache flushes the old caches.

5.3.5 Results From Caching

Before caching (m1.large)

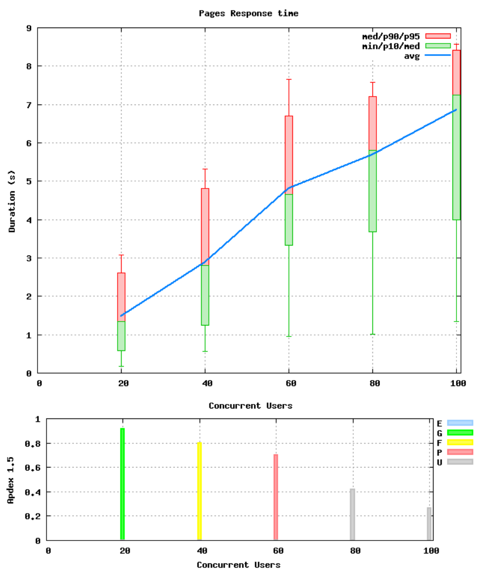


Figure 13: Test results without caching.

After caching (m1.large)

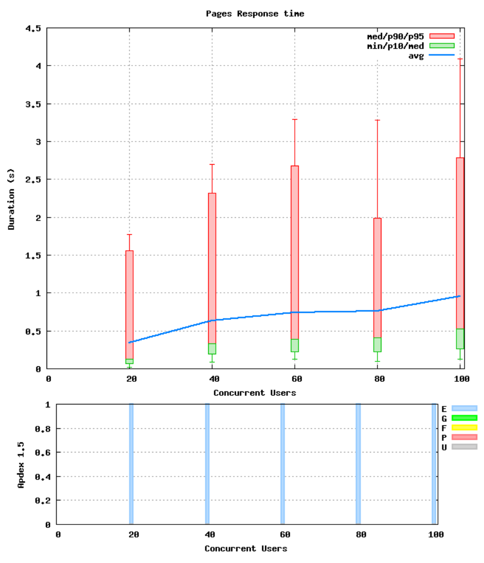


Figure 14: Test results after caching.

As shown by the graph, the average response time went from around 7 seconds at 100 concurrent users, to about 1 second. This is a very significant improvement, which goes to show that caching is a very powerful tool when it comes to scaleing an application. After all the optimization, we can see that the slowest request has changed, and is now the user creation post request.

5.3.6 Read Only Testing

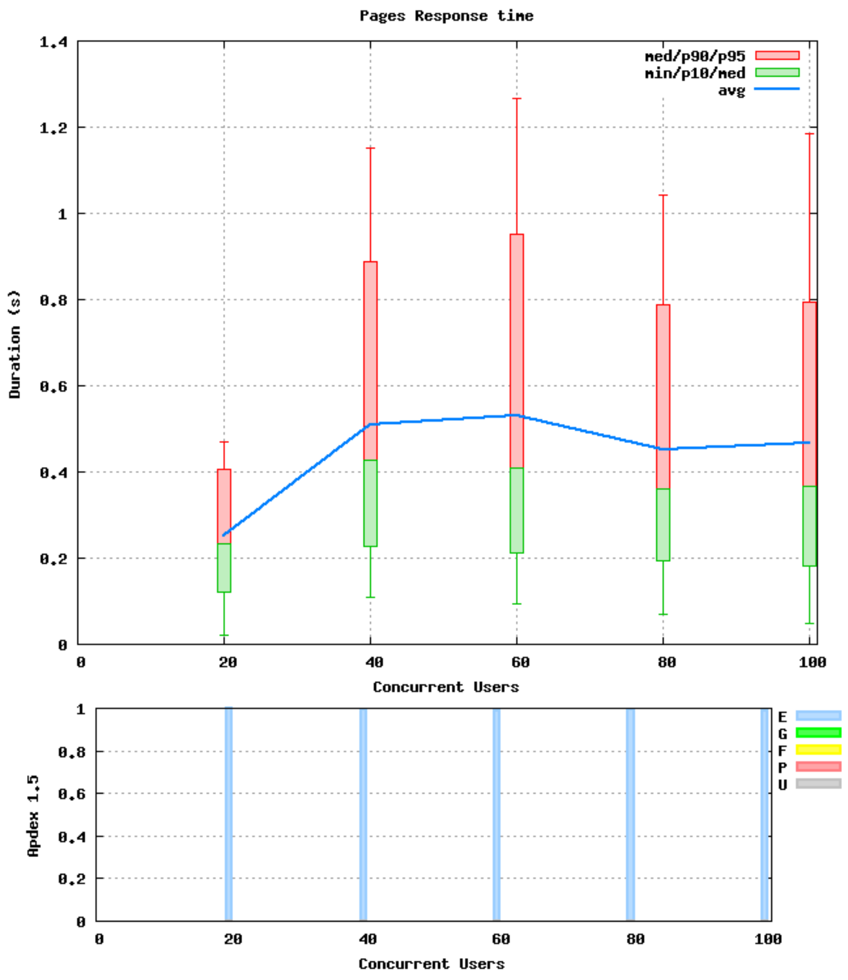


Figure 15: Test results of read only test.

As suspected in section 4.1, the results of the read only tests is an improvement over the write test (referring to figure x.x). We see two reasons for this. Firstly we know that one of the slowest requests of our application is user creation. In our write test every user creates a new account, causing the page requests to slow down.Secondly there is way more cache hits when a user is not logged in, since a lot of the caches are invalidated at log inn. It is, however, important to include results from the read only tests, as they mimic the application under load more accurately than the write test. Whereas the write test can be seen as more of a worst case scenario for the application, the read only test is closer to an average load scenario.

6 Configuring rails, Nginx and Passenger

6.1 Turbolinks

Turbolinks is a gem that make links in the web application work faster. It does this by keeping the current page instance alive and replacing only the body and the title in the head. This way the browser doesn't have to recompile the JavaScript and CSS between each page change.

6.2 Assets Pipeline

The asset pipeline precompiles, concatenates and minifies the assets into one central path. This makes the assets load faster, due to less requests being sent, hence the web application will perform better as well.

6.3 Nginx and passanger

We used Nginx as our HTTP-server. Nginx is a reverse proxy server that works also as a load balancer, HTTP cache and web server. It uses gzip compression and decompression to reduce time spent sending assets. As an application server we used Phusion Passenger. Passenger is a web server and application server for Ruby, Python and Node.js, and is designed to integrate into Apache or Nginx.

7 Optimizing javascripts and responsiveness

7.1 Javascript optimization

Even though the performance of our javascripts is not tested in Funkload, we found it to be a significant part of the total user experience. In the early stages of developing we had a map with several thousand markers, and this halted performance to such an extent that the user experience was heavily degraded. The first thing we did was to add buttons on the map where the user could choose between a region view and an all spots view. This way the user could choose the respective region and only render a subset of all the spots. This was a poor solution as it only partly avoided the problem instead of fixing it.

Our final solution was to strip off some functionality, such as choosing region and buttons, and introduce clustering of the markers. We did this through a grid-based clustering of the markers. This improved the performance of the javascript tremendously. The time it took to render from minimum to maximum zoom was improved from 22798 ms to 5054 ms, which is a factor of 4.5. This is still quite slow but we haven't had time to implement further improvements.

Further improvements can be achieved by creating a custom overlay, so that computations can be performed by the server. One can also use a viewport to only render the markers or clusters currently visible for the user, by getting the bounds of the map.

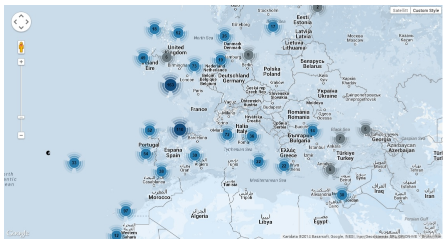


Figure 16: Showing our page when not responsive

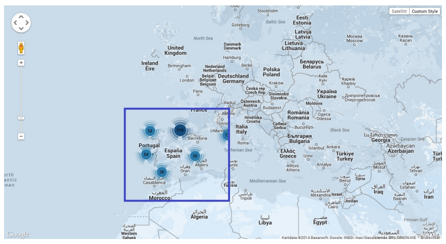


Figure 17: Showing our page when responsive

7.2 Website Responsiveness

One of our goals with the application itself has been to make it responsive so that it looks good on all screen sizes. This has been achieved by the use of Bootstrap and dynamically creation of Javascripts.

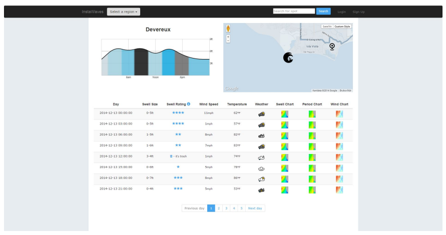


Figure 18: Showing our page when not responsive

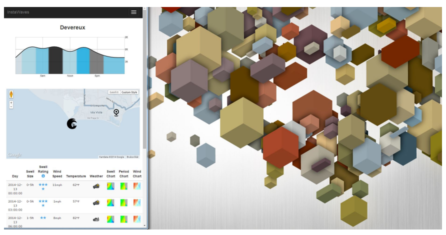


Figure 19: Showing our page when responsive

8 Vertical Scaling

Vertical scaling involves adding more power to our existing machines, and in most cases this means adding more computational power and memory. In order to test the application's vertical scalability we used three different instance types; M1, M3 and C3 . Both M1 and M3 instances provide a good balance between computational power, memory and network resources, but M3 will outperform M1 in most cases. Reasons for this includes better and more consistent performance and SSD-backend storage delivering higher I/O performance. C3 on the other hand is compute optimized and makes use of the highest delivering processes. The model instances are compared in the table below.

Instance type and model	vCPU	Memory (GiB)	Instance Storage (GB)	SSD Storage (GB)	Network performance
M1.medium	1	3.75	1 x 410	None	Moderate
M1.large	2	7.5	2 x 420	None	Moderate
M1.xlarge	4	15	4 x 420	None	High
M3.2xlarge	8	30	None	2 x 80	High
C3.4xlarge	16	30	None	2 x 160	High

Every test was performed with a Funkload stack running on an m1.medium instance. By looking at the average page response time for the critical user path we defined, as illustrated below, we observed a good improvement when scaling the application vertically, as expected.

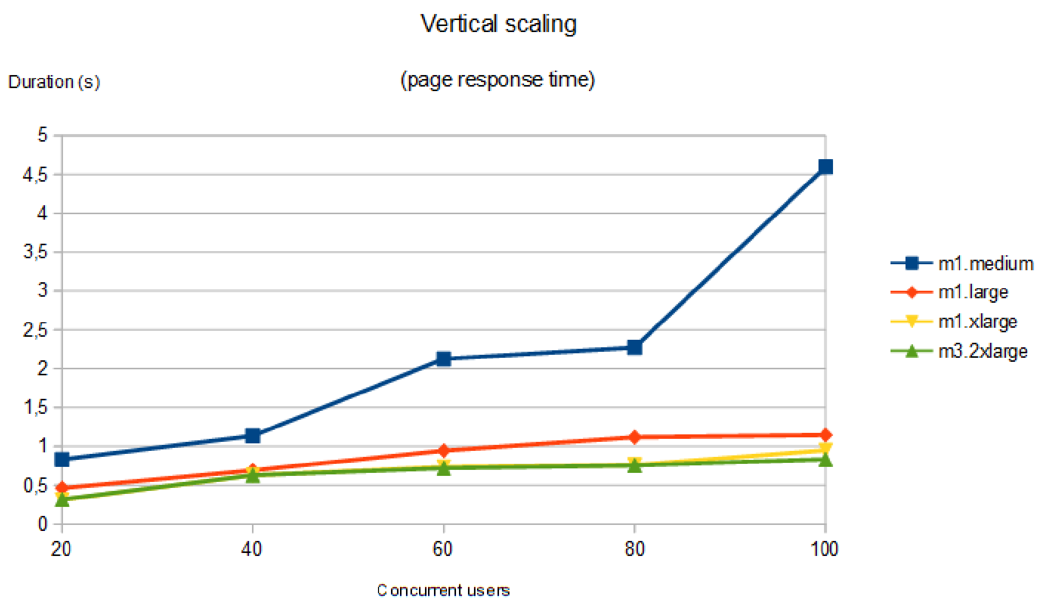


Figure 20: Vertical Scaling

For 100 concurrent users the average page response time decreased from 4616 ms to 1146, meaning a difference of 3470 ms when going from m1.medium to m1.large, which is a huge improvement. A reason for this could be that we are going from a single to two CPUs. Also when continuing to upgrade the instance the results improved a lot when looking at the changes in percentage. An upgrade from m1.large to m1.xlarge decreased the average response time by 220 ms, which corresponds to over 14% lower response time. Going from m1.xlarge to m3.2xlarge reduced the response time by over 12%. So a total of 81% faster response time was met.

The exception to the trend is the upgrade from m3.2xlarge to c3.4xlarge in which we registered no improvements. We believe this is due to the low number of concurrent users. For this reason the results for the stack running on a c3.4xlarge was not included in the graph above, because there were no improvements to illustrate.

Two more specific cases were investigated. The first one is the page response time for the index page, which is a get-request. This is interesting because we believe the frontpage will be most exposed to heavy loading under normal operating conditions. The second case looks at concurrent users signing up, which is interesting to analyze under heavy loading because it is a post-request, and we have also seen that it is the slowest request in our application. Below is the page response time compared for the smallest and the largest instance for both cases.

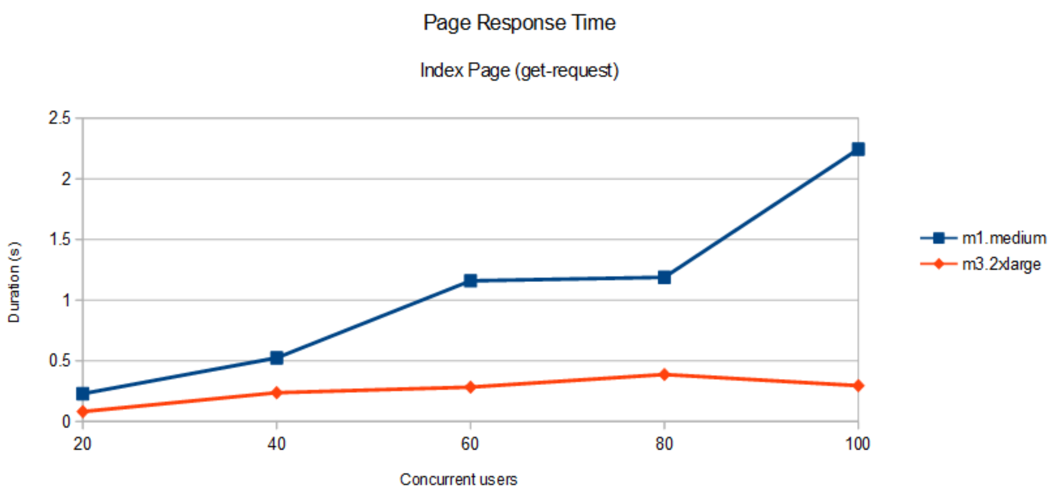


Figure 21: Vertical get request

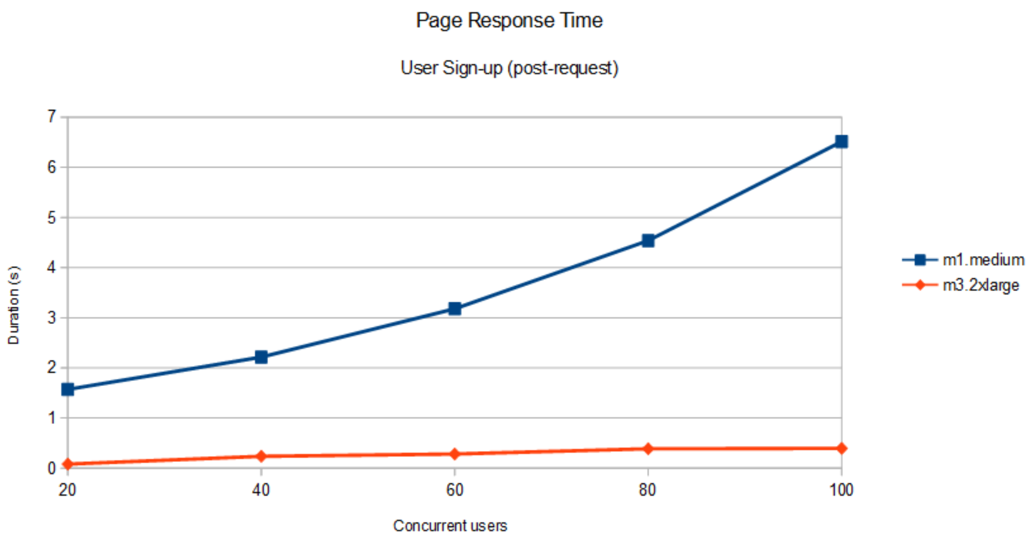


Figure 22: Vertical post request

The page response time difference for the two instances compared increase significantly as the number of users is increased for both requests analysed. When looking at 100 concurrent users for the get-request, the page response time decreased from 2244 ms to 295 ms, which is an improvement of close to 87%. For the post-request the improvement is even more spectacular, and we managed to reduce the page response time from 6516 ms to 395 ms, which is an improvement of almost 94%. For this request it is worth noting that the response time for the m1.large instance was 846 ms, and for m1.xlarge it was 606 ms. From this we observe that a small upgrade in instance made a huge difference, which might be surprising for this low number of concurrent users.

The smaller instances should be the ones failing first. This is observed as the application fails when running on the smallest instance with 100 concurrent users. The larger instances have no problems operating under such conditions, and therefore we assume that larger instances perform better under very heavy load. Of course we wanted to investigate this further, but as explained in section 4.2 on Funkload, we were not able to perform this kind of testing, even after spending hours upon hours trying to solve the issue.

9 Horizontal Scaling

Horizontal scaling involves adding more machines to the the pool of resources that the application uses. For us to test the horizontal scalability of our application, we had to separate our database from the appservers. We inserted the database on a single m1.large stack. Then we created several Passenger application servers, with an NGINX HTTP-server on top of that. For the app servers we used m1.large EC2 stacks. We added a load balancer to direct traffic into the different application stacks. We also included a memcached server on a separate stack, for caching purposes. Using a memcache instance could be quite advantageous as the different app servers reuse the same cache, which again will lead to more cache hits. If we were to store the caches locally, it would be stored on disk in the respective app server. This way there is no shared cache, which would increase the amount of missed caches, leading to slower performance.

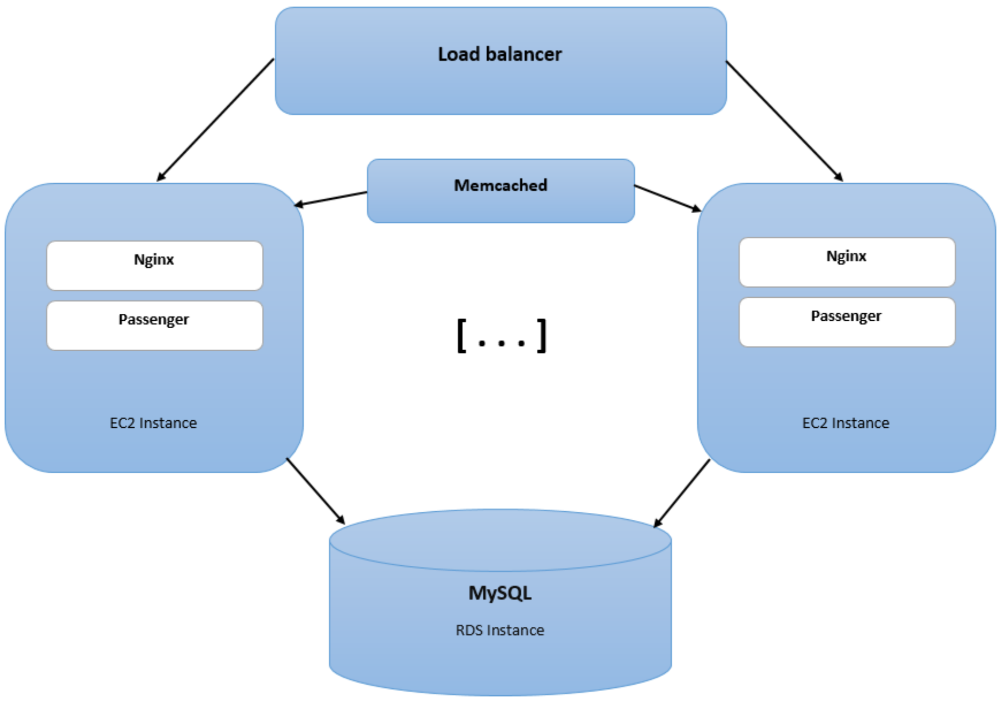


Figure 23: Basic EC2 setup

We created three different setups, and tested them using the critical path discussed in section 4.1. The first test had a single app server, the second one used two app servers while the final test had the maximum number of app servers available, which is eight. By looking at the average page response time for a number of concurrent users ranging from 20 to 100, we determined that the performance was significantly better when using multiple app servers. The page response time was on average reduced from 1181 ms to 806 ms, this is a reduction of almost 32%. We wanted to test the application under heavier load, i.e. more concurrent users, but our

Funkload problems mentioned in section 4.1, limited us to testing only 100 concurrent users. In theory the effects of horizontal scaling should increase as the number of concurrent users increase, and we should be able to handle more load.

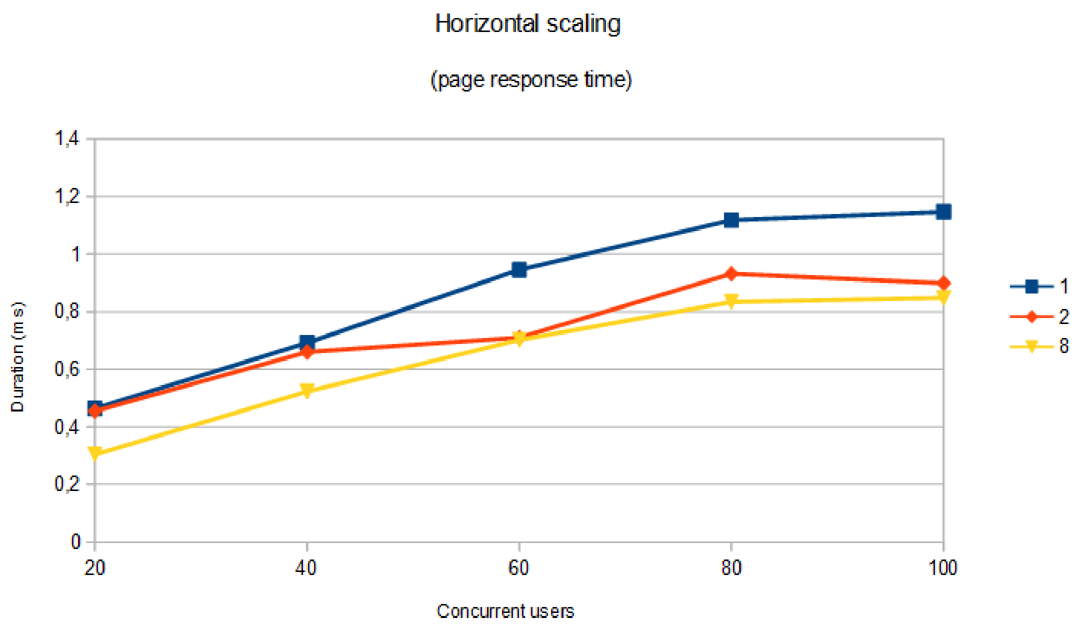


Figure 24: Horizontal - Response time

When looking at the get-request to the front page, the page response time went from 574 ms to 463 ms when increasing the number of app servers from one to eight and simulating 100 concurrent users. This is an improvement of almost 20% which is quite significant. All of our tests, both for get- and post-requests, showed approximately the same improvement. This is much unlike the vertical scaling, where the variation in improvements was much greater, especially when comparing get- and post requests.

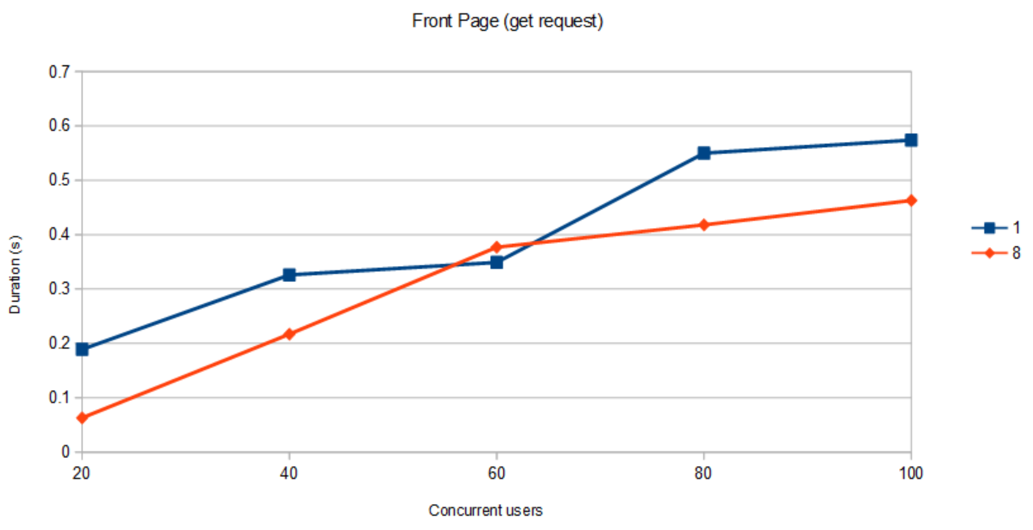
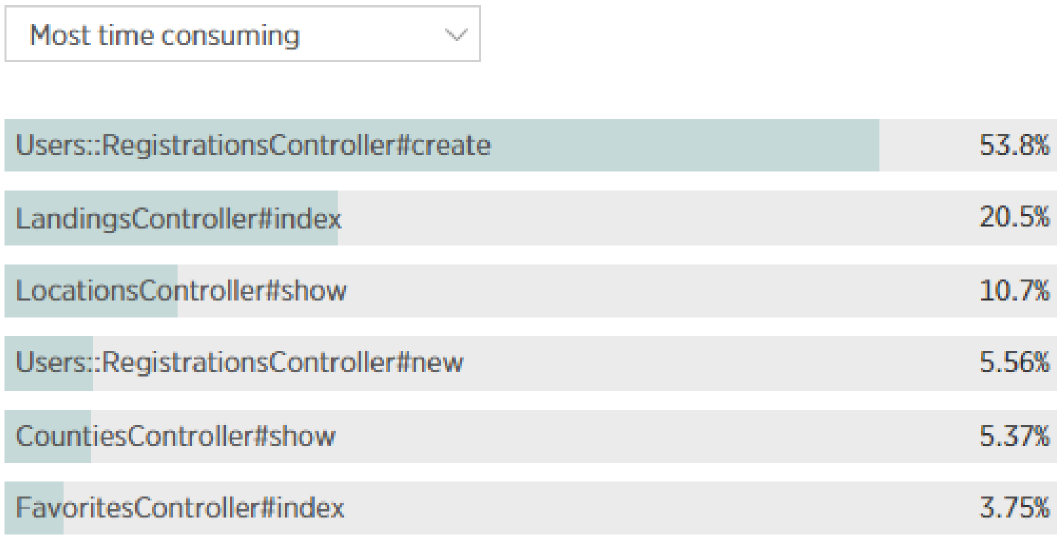


Figure 25: Horizontal - Frontpage get requests

10 New Relic

In addition to set Funkload to test our application, we used a service called New Relic, by installing their gem, and signing up for a free account. From the reports generated by New Relic, we can see that the most time consuming operation is creating a new account. We also got some metrics on the scalability of our application.

Figure 27 below shows that the response time of our application does not increase as the number of requests get bigger.



Users::RegistrationsController#create	53.8%
LandingsController#index	20.5%
LocationsController#show	10.7%
Users::RegistrationsController#new	5.56%
CountiesController#show	5.37%
FavoritesController#index	3.75%

Figure 26: Load distribution

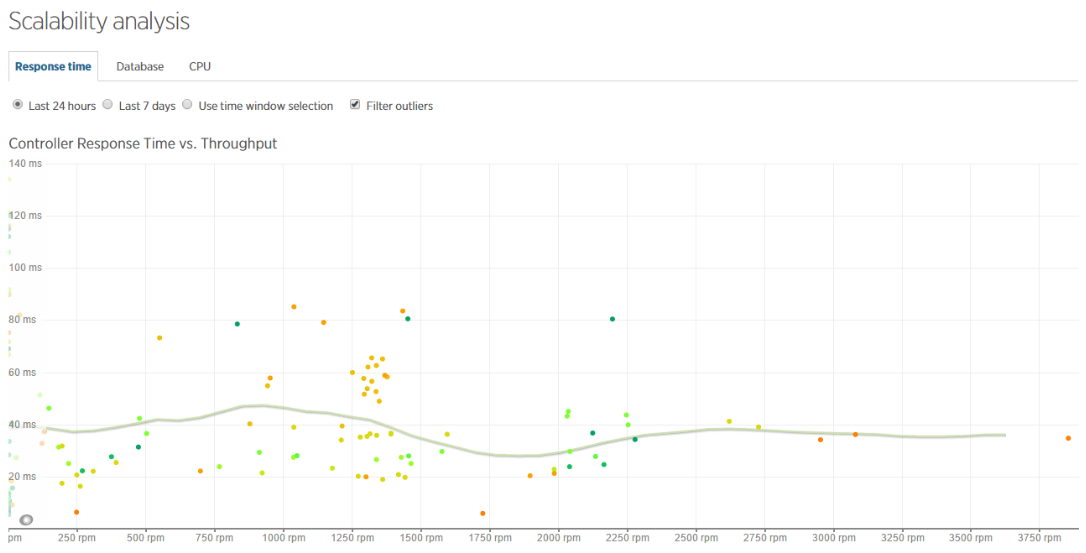


Figure 27: Controller Response Time vs. Throughput

11 Further Development

11.1 Evaluate Geocoder

Although we have done a lot of optimization of our application, there is still parts that can be tweaked or optimized further. This is especially true in regards to the nearbys array we get, that contains a list of all the nearest spots. The way this list is obtained is through a gem called Geocoder. Geocoder uses a client's IP address to determine the location, and this lookup is done via the REST API of Telize, which allows for unlimited geolocation requests (and is thereby perfect when load testing). Geocoder also comes with several methods that operates on the location arrays, i.e. nearby(d) which returns the spots within the distance, d. The issue arising is that one cannot use the nearby scope within another scope that uses includes because the SELECT clause generated by nearby will overwrite it. There are workarounds for this issue, like for instance using .joins with the :select option, but we are so far unsuccessful in making a query like this that works as intended. As a consequence, the result is an n+1 query, and should be improved accordingly. Another workaround is to make the methods that calculates the nearest spots ourselves, but our time is better spent on improving the application elsewhere.

11.2 Optimization via Configuration

When we reach a point where we have "exhausted" scalability both vertically and horizontally, we can look into fine tuning configurations of MySQL, Nginx and Passenger. We could look into some of the following things:

- Nginx and Passenger optimizations:

The first thing to look into when it comes to Nginx's and Passenger's performance is the number of workers, handling both processes and connections. The general rule of thumb here is to make sure that the number of cores we have available match the number of processes, and that the connections match the droplet size of nginx. Further we could configure buffer sizes and access logging. We could also experiment with responsive compression and disabling of Nagle's algorithm. Disabling Nagle's algorithm could elevate the speed by enabling free flow of data through the network. This can be good for sending frequent small bursts of data.

- MySQL cache warming

  The performance of our application should increase if the caches are warmed up. This means that the cache already contains most of the cache fragments, so they won't have to be written when a user enters the page. This is especially applicable to the wave data of our application. Since the wave data should update at the very least every 24 hours, we should look into reheating the caches after said update. If done during a period with little load, like during the night, users don't have to notice any latency due to the warming of caches.

11.3 Availability

As a web service grows, failures are bound to occur one place or another, no matter how fast the web service may be. For InstaWaves, these failures can be identified to happen in three places, namely in the web server, the application server or the database. Obviously, by creating several web and application servers, the website won't shut down if one server fails. However, all of our setups only had one database instance. If that instance fails, there is no backup. This will cause the application to shut down, and could possibly lead to loss of data.The obvious solution in our case is to have multiple redundant db instances. This will increase availability at the cost of performance, as data needs to be replicated, and we need to ensure database integrity. An alternative to pure redundancy is database sharding. By using sharding, we could partition our database by dividing our dataset in a clean way, through clustering data by regions of the world. The users who then signs up could be tracked by their geolocation, and added to the matching database, thereby splitting out similar data among several RDS instances. On the other hand, making such a partition will decrease the scalability by some factor, so we would have to find a tradeoff that gives a satisfactory balance between availability and scalability.

When adding several databases, these should be located in different regions and/or different continents. This is done so that all data wouldn't be destroyed in case of a large scale disaster or power failure.

12 Learning points

- Load testing takes a large amount of time.
- Agile development (combined with Pivotal Tracker) is a great model for developing and coordinating the workflow of a project.
- Allocation of time. Use more time in advance to consider what's important and not, so that we avoid spending a lot of time on non-critical components of the application.
- Piazza and lab hours proved very helpful whenever we were stuck on problems. Learning from experienced people is always very effective.
- Pairing is helpful, and was vital at several stages throughout the project.

13 Summary and Conclusion

Through the course of Scalable Internet Services we have learned how to build, deploy and scale a web application. This includes doing everything from making the initial RoR application and learning the different languages and frameworks, to deploying the application in the Amazon cloud, and performing load testing on the instances.

We are all relatively new to web development, so the learning curve has been steep, but the learning outcome has been huge, and rewarding - not only in terms of scalability, but also how "everything" around an internet service is connected.

From our optimization and testing we can also conclude that the application itself benefited greatly from improving the way we communicate with the database. Query optimizations through adding indices and utilizing eager loading, actually goes a long way for making the application faster and creating a better user experience. Caching has also proved to be a valuable tool in decreasing page response, by reusing fragments across different users, preventing heavy computations. All these techniques are relatively cheap, and can significantly improve the scaling of a website.

If improving queries and caching isn't quite enough, both horizontal and vertical scaling can give a significant improvement of the applications response times, and provide an easy way to scale an application, given that the financial costs are not a problem.