SCALABLE INTERNET SERVICES

EASYGO

December 3, 2015

Group: EEGO
Fengjia Xiong: 204435922
Mingrui Shi: 104517541
Haosheng Zou: 804518325
Zimu Li: 504516709

Contents

1 INTRODUCTION

EasyGo is web application that enables users to search for travel plans and share their own travel plans.The main feature of EasyGo is that users can search for travel plans based on location, number of days or number of people to travel with. Advance to traveling, people can write travel plan for the incoming travel to get fully prepared, and they can share their experience after their amazing adventure to help other people planning for traveling. Changing, deleting and exploring travel plans can be easily done in EasyGo.

EasyGo also has networking feature enabled to let people meet with their friends and make new friends who are also interested in traveling. Moreover, by creating travel groups, users can work on the same travel plan together and share information with each other in private.

EasyGo is implemented in Ruby-on-Rails (Ruby 2.2.1 and Rails 4.2.4) and uses an RDBMS (MySQL) as the backend data store. Since the purpose of Scalable Internet Services is to build and deploy a scalable web service, this report will discuss the deployment, performance and scalability of EasyGo.

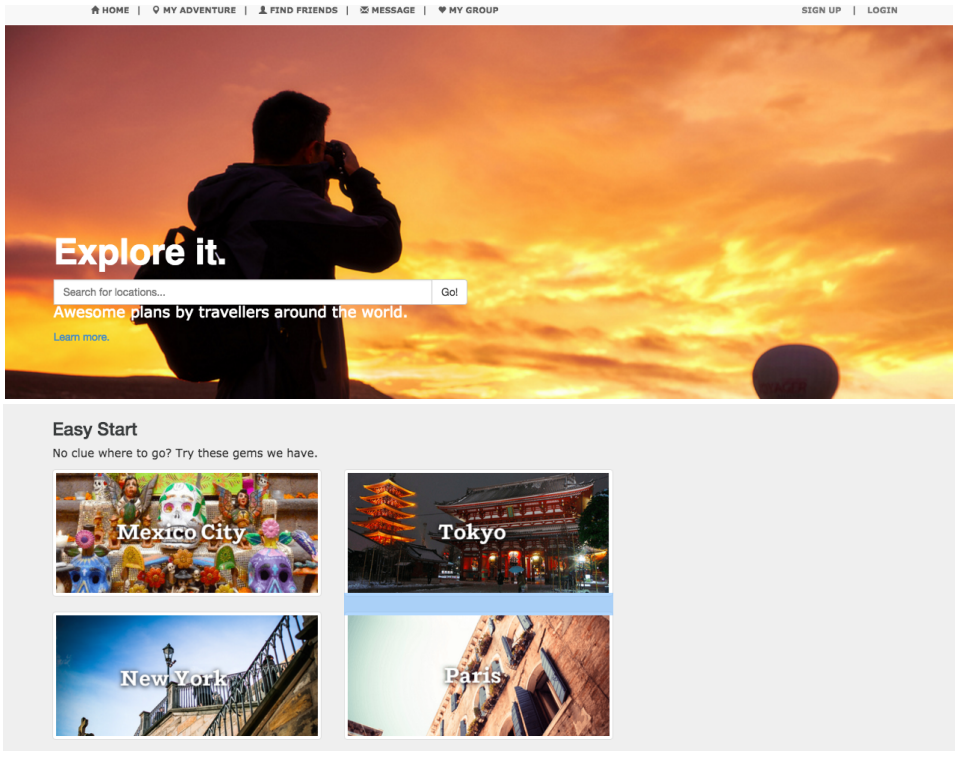


Figure 1: EasyGo Home Page

2 DEVELOPMENT

We are motivated to use Agile method in our application development process. Sure enough, what we followed in our development routine was a light version of Scrum with only the team, using Pivotal Tracker as a story tracker and team collaboration platform and updating sprints every week. We did not quite center our development on test-driven methodology, though, due to the concentration on our restricted features. We strongly believe, however, that TDD plays an important role in larger projects. Most of the time we worked separately towards everyone's own goal, and did adopt pair programming in certain links where we expected and found it really efficient.

We utilized Tsung in load testing, which is a significant part of our development. We used the same Tsung script for almost all the tests. Four different sessions with various weights were created, featuring distinct user behaviors. These user behaviors include:

User Behavior
1. Creating a new adventure and repeatedly checking own adventures;
2. Repeatedly searching for all existing adventures;
3. Repeatedly searching for adventures with several specific keywords;
4. Repeatedly searching for random users.

Several behaviors were conducted multiple times in one single session in order to emphasize the effect of our improvements. Besides, we believe there is plausibility in such actions for simulating, for example, cautious users who tend to refresh pages frequently, or certain events where all users hope to get most up-to-date information.

We used Github for our external repository, as well as Git for version control.

3 ARCHITECTURE

3.1 System Flow

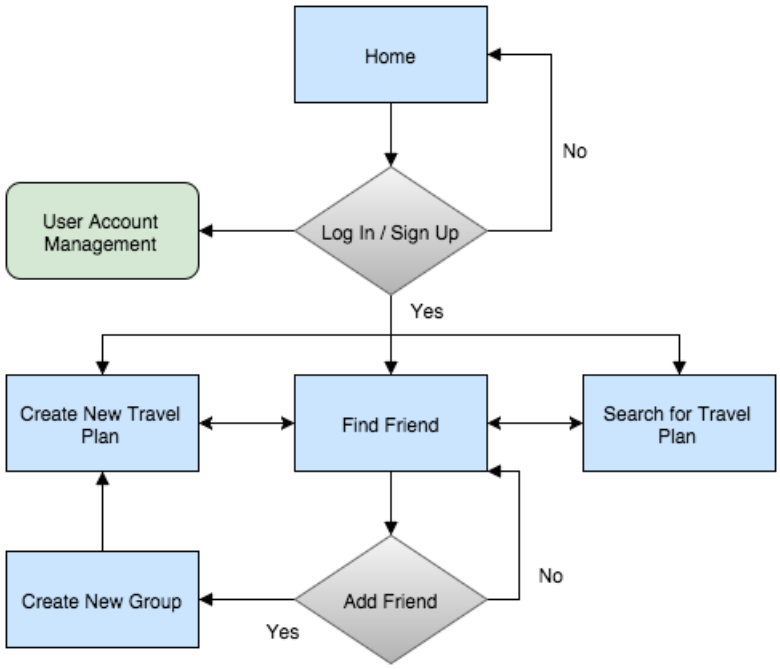


Figure 2: System Diagram

In our website, except for home page, all the other operations require user login. When it is the first time for the user to visit, we provide signup service.

Generally, our website consists of three major services: adventure plans, friends and groups. The most used feature is adventure plans. Users can use *search* bar to get all the existing adventure plans with a matching location name. If there is no available searching results, users can create their own adventure plans.

Another appealing feature is joining groups. Each user can search for other users by name, and send “add friend" requests to them, after which other users will receive notifications of new friend requests. Once these users accept their friend requests, they become friends. Then among all their own friends lists, each user can create new groups, which contain some or all of their friends. The aim of introducing “group" feature is to allow users to create their private adventures, which can only be edited and seen by all of friend members in the group. This can not only cater to those users

that care much about their privacy, but also make it easier for a group to share a common adventure plan.

3.2 Database

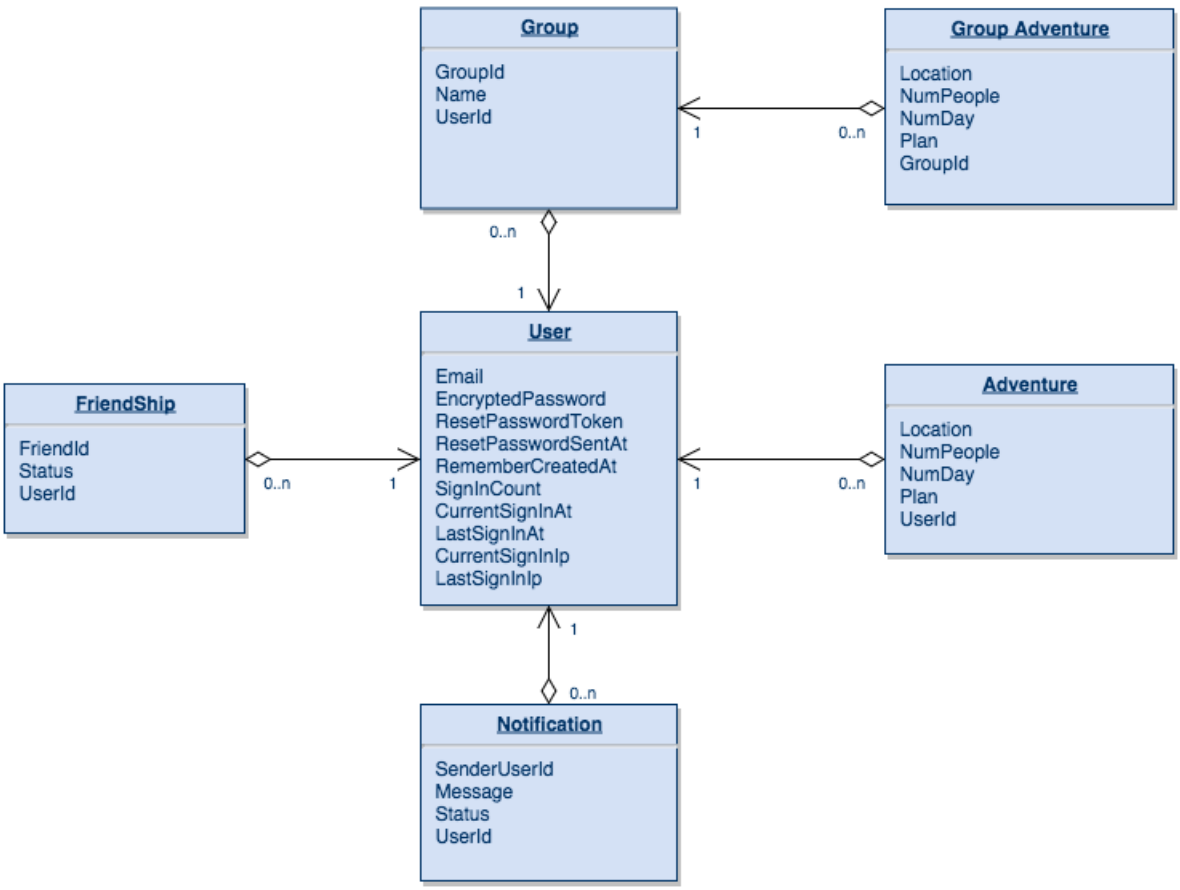


Figure 3: Databse Schema

The figure above shows our database schema. We designed six different tables to implement our adventure, group and friend features. For each user, he can have multiple friends, groups, adventures, and notifications. Each group can have one or more group adventures.

4 SERVER CACHE

4.1 Data Cache

In this part, we use two ways to improve the performance of our website. One approach is to use low-level system cache, and this will eliminate extra repeated database queries. Another way is to use **dalli** to implement memcached.

4.1.1 Eliminating Extra Database Lookups

As one of the most common patterns of improving system's throughput, caching will eliminate database lookups for commonly accessed data. If the server receives the same request multiple times, and has to make the same query to lookup data in the database, it wastes a lot of resources for this redundant job. If we use cache's fetch method and get results directly from previous query stored in the cache, it will save the server great effort.

In the following block, we use fetch method for **Adventure** model.

Low Level Cache for "Adventure"

```
1  class Adventure < ActiveRecord::Base
2    belongs_to :user
3    def self.search_by_location(location)
4      @adventures ||= Rails.cache.fetch("adventure_#{location}",
           expires_in: 1.minute) do
5        @adventures = Adventure.where("location LIKE ?","%#{location}%")
6      end
7      @adventures
8    end
9
10 ...
11 end
```

We can see from the block above that instead of making queries to the database, we add another fetch method *search_by_location* in the **Adventure** model. The first time a user hits the adventure search based on location, the server will process line 6 and generate query for data lookup. Also because this is a new query and no available cache can be used, the server will populate the cache for

this lookup miss. Then for all the later adventure lookup requests with the same request parameters, as we already have the response results in cache, the server will directly fetch data from cached results, greatly reducing the database burden.

4.1.2 Memcached

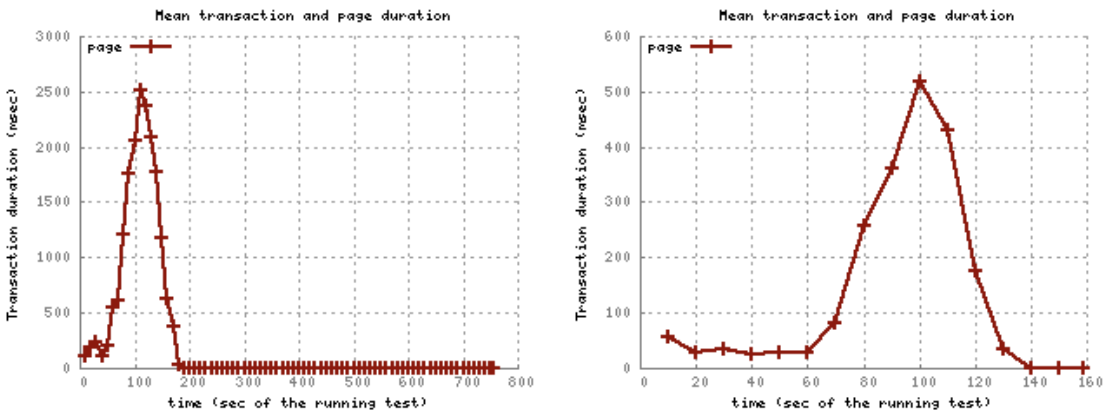
With fetching cached results, we can, to some extent, improve the system's performance. This cache mechanism is handled by Rails itself. However, this cache mechanism is not efficient enough. In order to further improve the cache efficiency and speed, we introduce **Dalli** to our system.

Dalli is a high performance pure Ruby client for accessing memcached servers. We bundle the 'dalli' gem and then change the default cache storage to 'dalli_store'.

In this case, 'dalli' stores the cache in another machine, which reduces the read and write time for cache fetch and population. When the memory is filled up, the old cache would be flushed by memcached.

4.1.3 Data Cache Load Test Result

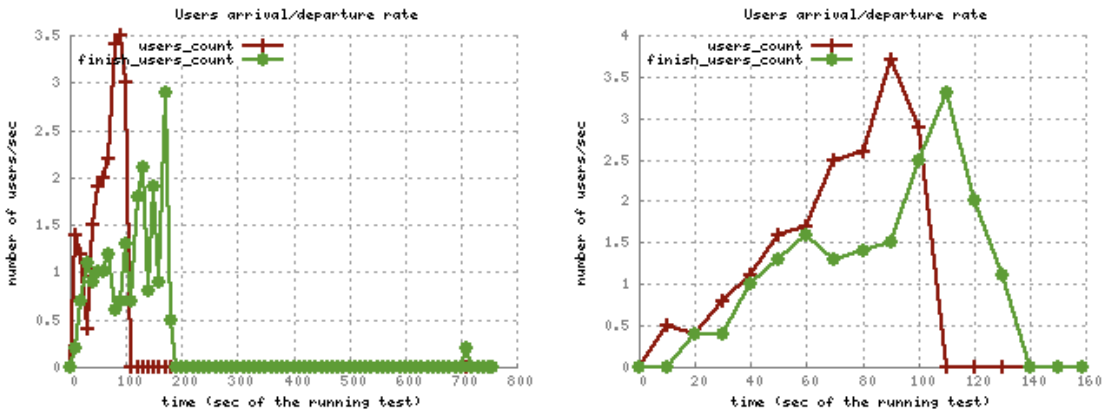
In this part, we use load test to evaluate the performance of data cache. For both cases, we use AWS EC2 **m3.large** instance.



(a) No Cache **(b)** Data Cache

Figure 4: Mean Transaction and Page Duration: Data Cache

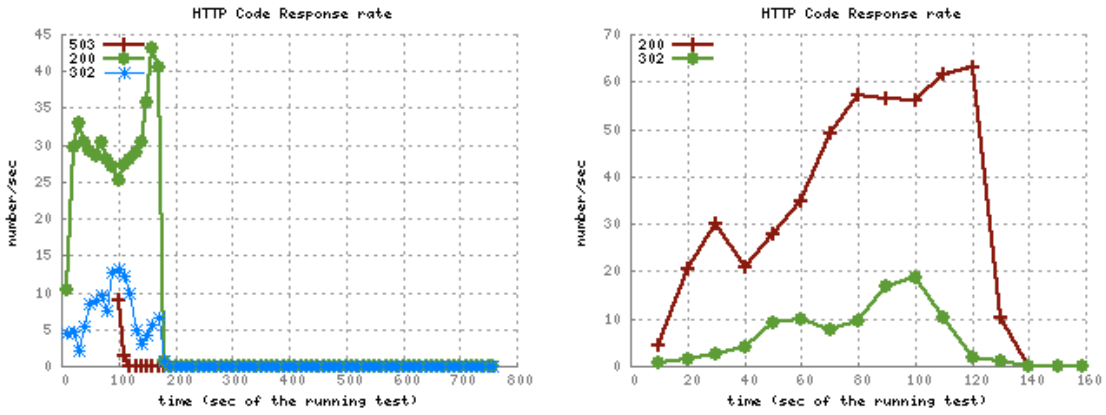
In the figure above, we can easily see that with low-level data cache and **dalli** gem, the mean transaction time and page duration are greatly reduced from 2500ms to around 500ms.



(a) No Cache **(b)** Data Cache

Figure 5: User Number: Data Cache

We can also see the user arrival and departure rate from the above figure. In the system with no cache, the last user that finishes all the requests in the session leaves at around $t = 200s$. In contrast, with cache, the server can easily fetch the query results stored in memory, and immediately respond to users.



(a) No Cache **(b)** Data Cache

Figure 6: Http Response Code: Data Cache

HTTP Return Code

No Cache

Code	Highest Rate	Total number
200	43 / sec	5071
302	13.2 / sec	1245
503	8.9 / sec	103

Data Cache

Code	Highest Rate	Total number
200	63 / sec	4914
302	18.7 / sec	941

As our load test assigns different weights to different session operations, every user that arrives will choose a session based on the probability distribution of session weight. So it is very common that the sessions vary among different tests. Hence, there is a small fluctuation on the total number of each type of return codes in these two instances. Nevertheless, this trivial difference will not have too much influence on the accuracy of the tests.

Due to the heavy load, after around $t = 100s$, the server without data cache mechanism can no longer respond to all the users' requests, and respond code 503 to the users. The server is down after $t = 100s$. However, the server has relatively much lower pressure, and is able to handle all the incoming users' requests. The highest rate of returned 200 code is increased in the more robust server.

4.2 Fragment Caching

4.2.1 Static Page Caching

In our application, we have several pages or fragments are static during the entire user session. For example, our navigation bar is rendered on every single page request, so we can cache it in order to prevent redundant rendering.

Another example, the static home page of our application contains images and may be requested at a large rate. By caching the home page, we can get significant improvement on server performance when serving a large number of users.

In a word, we can perform fragment caching on all the static or seldomly changed fragments to reduce redundant requests by directly reading from cache.

4.2.2 Russian Doll Caching

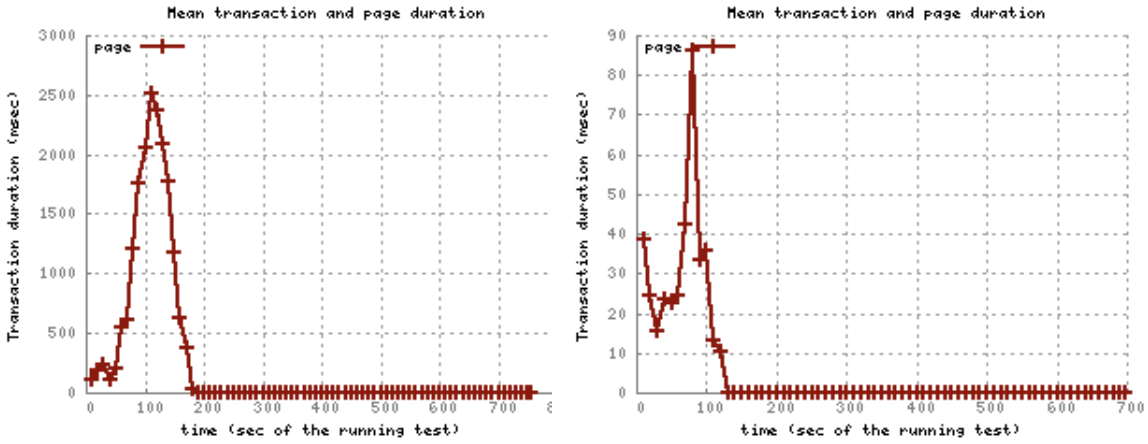
In addition to static pages or fragments, our application also have fragments or pages, which may be relatively frequent to be changed, such as “Search Adventure" feature searching through keywords and “My Adventure" feature corresponding to each single user. However, in these pages or fragments, most parts remain the same, which means only a small part may be changed for each time. To be more specific, searching through keywords could be cached by using search parameter as the cache key while adventures associated with each user could be cached by using current_user id as the cache key, which are the outer loop fragment caching. Moreover, in each table, since only a small part is changed for each time, we implement inner caching for every entry of the table to prevent redundant requests for the entire table.

Russian Doll Caching

```
1 <% cache(cache_key_for_adventure_table) do %>
2         <% @adventures.each do |adventure| %>
3                 <% cache(cache_key_for_adventure_row(adventure)) do %>
4                         # rendering each entry of the table
5                 <%end%>
6         <%end%>
7 <%end%>
```

After implementing fragment caching for “Search Adventure" and “My Adventure", the fragments could be reused through caching when users search adventures for the same location or access their own adventures. Follow the same rule, other features could also benefit from fragment caching, resulting in significant improvement on the server-side performance.

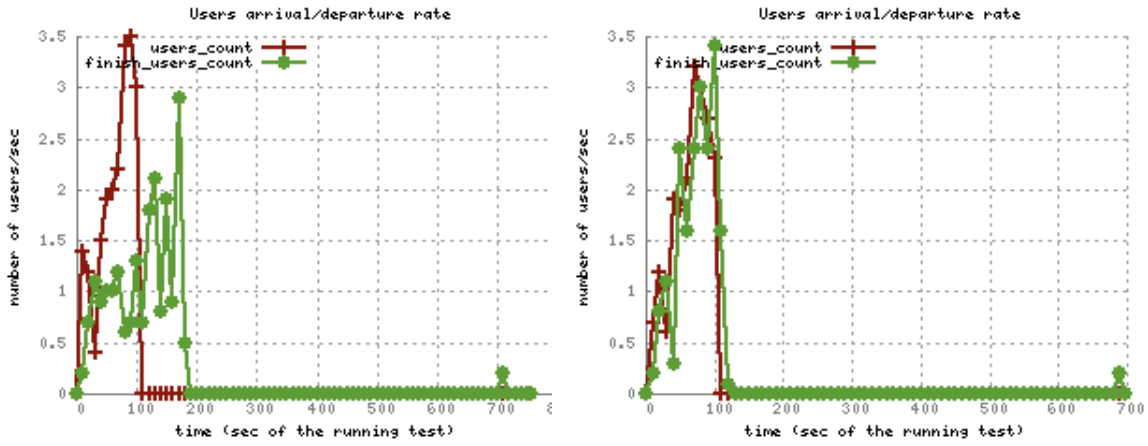
The following results show evaluation of performance with server-side fragment caching.



(a) No Cache **(b)** Server Cache

Figure 7: Mean Transaction and Page Duration: Server Cache

In the figure above, we can easily see that by using server-side fragment caching, the mean transaction time and page duration are dramatically reduced from 2500ms to 85ms. To certain degree, this is because our tsung contains a lot of repeating actions that hit the same page or search for the same content, but we could see that the performance improved significantly.

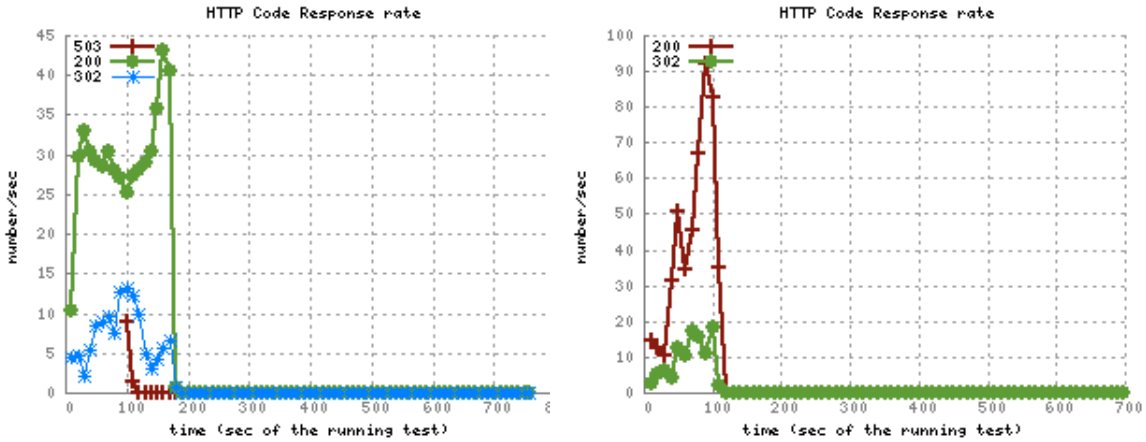


(a) No Cache **(b)** Server Cache

Figure 8: User Number: Server Cache

The above figure shows the users arrival and departure rate during the load testing. With

server-side fragment caching, user can quickly finish the entire session and depart while no caching results in an about 80ms time gap between arrival and departure for each user.



(a) No Cache **(b)** Server Cache

Figure 9: HTTP Response Code: Server Cache

HTTP Return Code

No Cache

Code	Highest Rate	Total number
200	43 / sec	5071
302	13.2 / sec	1245
503	8.9 / sec	103

Server Cache

Code	Highest Rate	Total number
200	92.3 / sec	4783
302	18.2 / sec	1056

We can also evaluate the server-side caching performance from the HTTP-CODE rate perspective. Without caching, users start to get 503 around 100s while no 503s occur with server-side fragment caching.

Concluding from evaluations above, server-side fragment caching is extremely useful when you have a large number of users searching for the same content or hitting the same page. For fragments or pages changing frequently, server-side fragment caching may not necessarily needed because writing fragment to the server consumes additional time. However, in general, implementing fragment caching results in an improvement on application performance.

5 CLIENT CACHE

5.1 Web Caching

Control Web Caching

```
1   def search
2     location = params[:location]
3     @adventures = Adventure.where("location LIKE ?","%#{location}%")
4     expires_in 1.minutes, :public => true
5   end
```

In line 4, we use web caching for users to cache their adventure search requests. When we hit the same search request more than twice, we will receive “200 OK from cache" in the http response header.

The public setting indicates that the response may be cached by any cache or proxy and should never be used in conjunction with data served up for a particular end user.

5.2 ETags

The ETags scheme, where E stands for entity, allows us to avoid making any request at all if nothing has changed on the server since the last time a particular resource was requested. Properly implemented ETags scheme can serve as a great improvement for web services that have especially high traffic load.

ETags

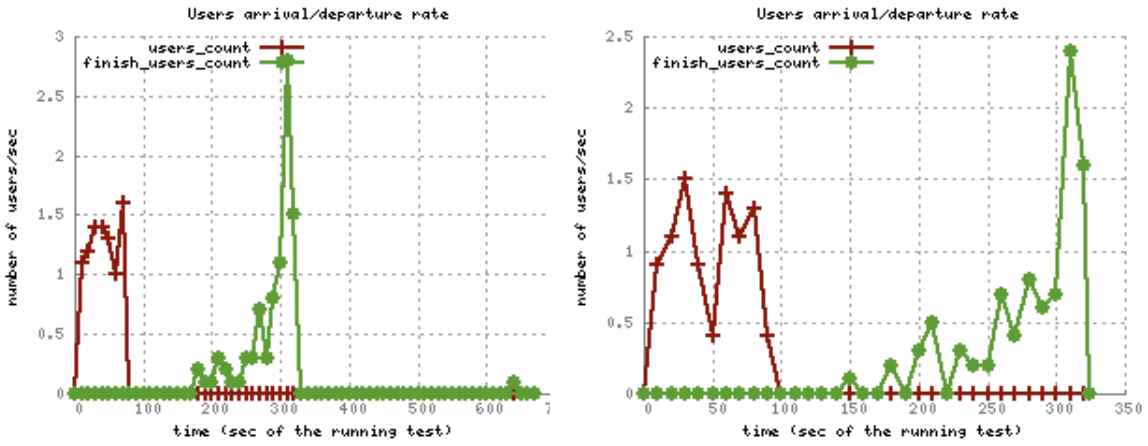
```
1   def index
2     redirect_to '/users/sign_in' unless user_signed_in?
3     location = params[:location]
4     @adventures = Adventure.where("user_id=#{current_user.id}") if
          stale?(Adventure.all, public: true)
5   end
```

Every time when we send a request to the server, it will automatically insert the ETag header in

200 OK response. Common methods of ETag generation include using a collision-resistant hash function of the resource's content, a hash of the last modification timestamp, or even just a revision number.

If a request with a matching ETag comes in, the response will be 304 Not Modified in the header and empty in the body, which indicates that there is no difference on the response between last request and current request.

To make ETags client caching works in AWS, we bundle 'rails_weak_etags' into the gem. The user arrival and departure diagrams are shown in the following figure.



(a) No Cache **(b)** Client Cache

Figure 10: User Number: Client Cache

We can see there is a small left shift for finish_user_count curve in the figure above. After implementing client cache, the first finished user shows up at $t = 150s$, compared with $t = 180s$ of the server without cache. Besides, the time that all the users finish the session is moved ahead by 10s with the use of client cache.

When we test our website in AWS with single user, the client cache shows a better performance. The average response time can be reduced to around 30ms after second time visiting, which costs only one tenth of the regular response time.

6 QUERY IMPROVEMENT

As mentioned in one of the lectures, queries made to the database side are some points we will surely investigate for performance improvement. Sure enough, we made some progress in improving the performance by modifying our implementation of queries.

Take friend search query as an example. Before the improvement, our original friend search query logic in controller works as below.

Original Query

users = **User**.*search*(**params[:search]**)

This gives a list of users that meet the constraint. Later in its corresponding view part, we still need to check the status of the friendship between every user in the list and the current user (if there is existing friendship record between them), which would be

View

```
1 For user in users
2     If user has existing friendship with current user
3         #check this friendship status and generate corresponding view
4     EndIf
5 EndFor
```

This means that every time we search a keyword, the number of queries made to the database is proportional to the number of users that meet our searching keyword. This feature makes the performance bad and unpredictable.

To visualize the viewpoint we just put forward, a screenshot of server console when searching for users with emails containing the letter ‘a’ is shown as below.

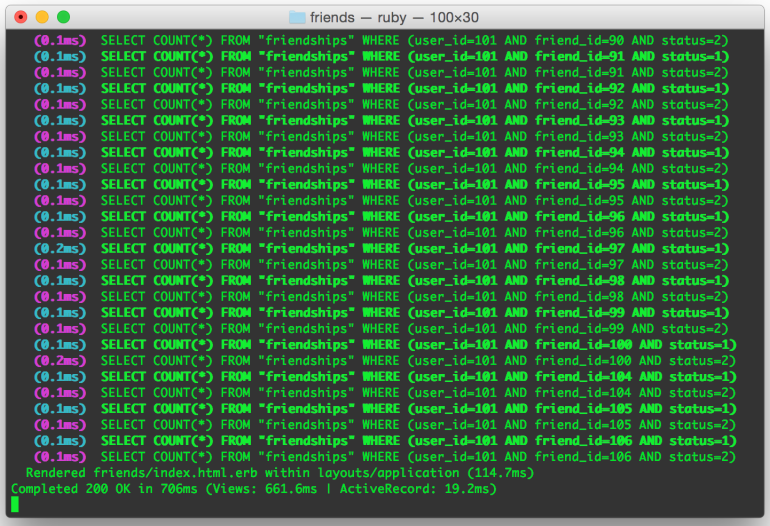


Figure 11: Original Query

Since there are more than 100 users that meet this condition, there will be more than 200 queries made at this single action. This is far more than what we could afford.

To improve this situation, we changed our query implementation to the following one.

Improved Query

```
1 @friends = User.includes(:friendships).where("email LIKE '%#{search_kw
      }%' AND friend_id=#{current_user.id}").references(:friendships)
2 @users = User.where("email LIKE '%#{search_kw}%' AND id NOT IN '%#{
      @friend_ids}%'")
```

The first query is utilizing ‘join’ to merge all the trivial queries into one big query. This may lead to longer time in one single query. However, because of the drastic decrease in number of queries, this will eventually lead to better response time. This is better shown in later paragraphs.

Now we are compressing all those queries into two queries, and will deal with these two parts separately in the corresponding view. To see the effect of this modification, we provide the following screenshot of server console when searching for the same keyword ‘a’.

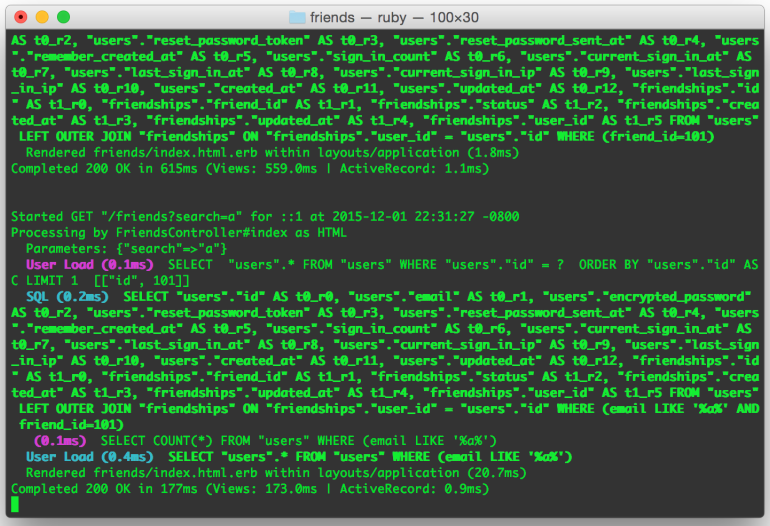
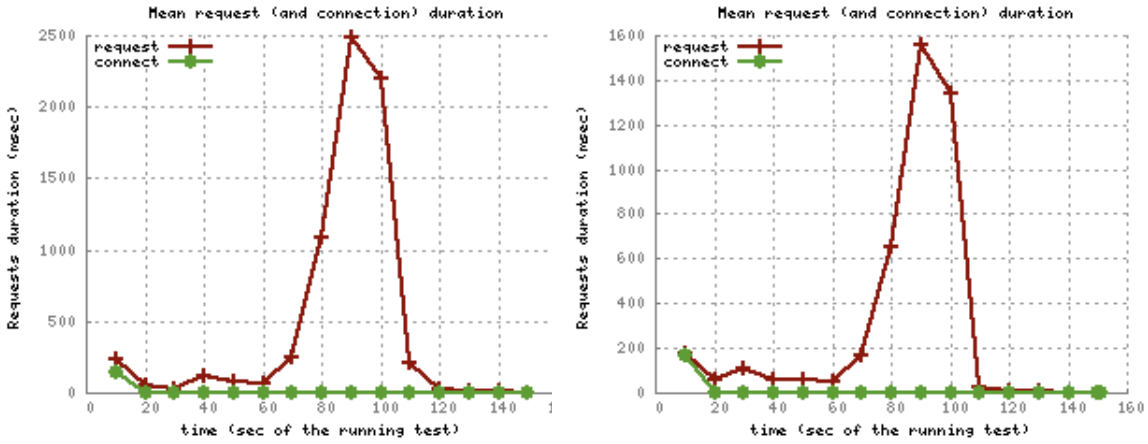


Figure 12: Improved Query

Comparing the screenshot above with the screenshot on the previous page, we can see that with the use of 'join', the massive queries are merged into two big queries in effect. Apparently, both query count and query time are reduced. The immense reduction in query count will surely make our server more tolerant towards large load. To further testify this performance improvement, we run the same Tsung script for both versions and get the following results.

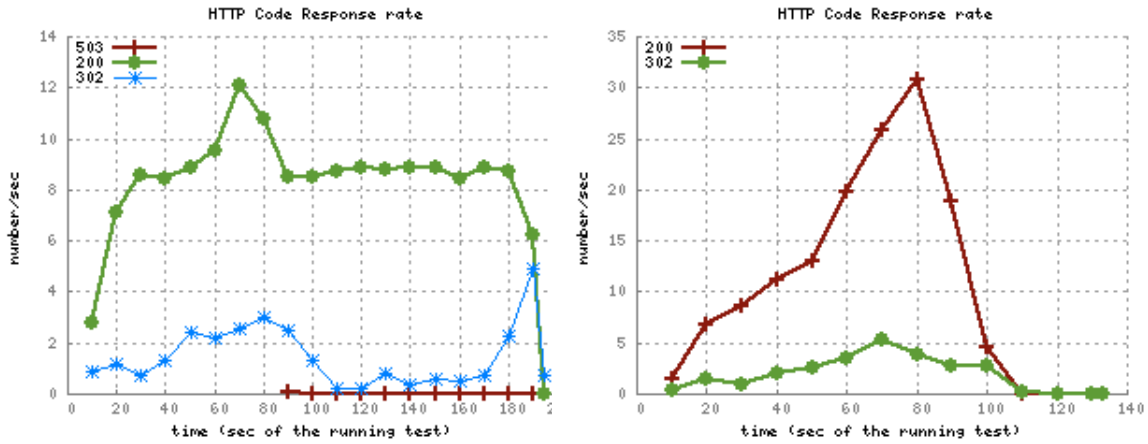


(a) Original Query **(b)** Query Improvement

Figure 13: Request Duration: Query Improvement

We can observe the significant reduction in the peak value of request duration after applying the query modification, which shows the impact of this improvement.

We continued modifying the Tsung script by increasing the arrival rate in the last phase, which further revealed the great distinction in performance between the two implementations.

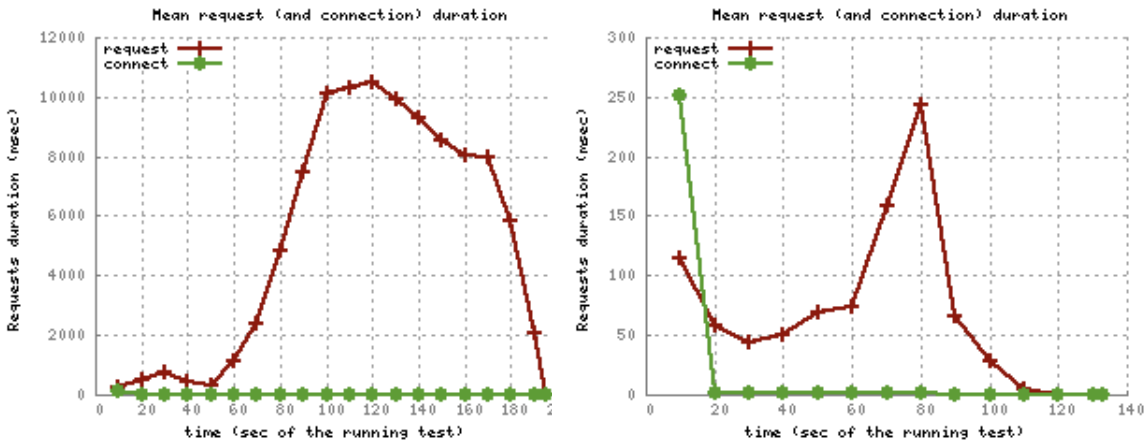


(a) Original Query **(b)** Query Improvement

Figure 14: HTTP Response: Query Improvement

The HTTP response rate under the original query scheme is comparably lower than that with the

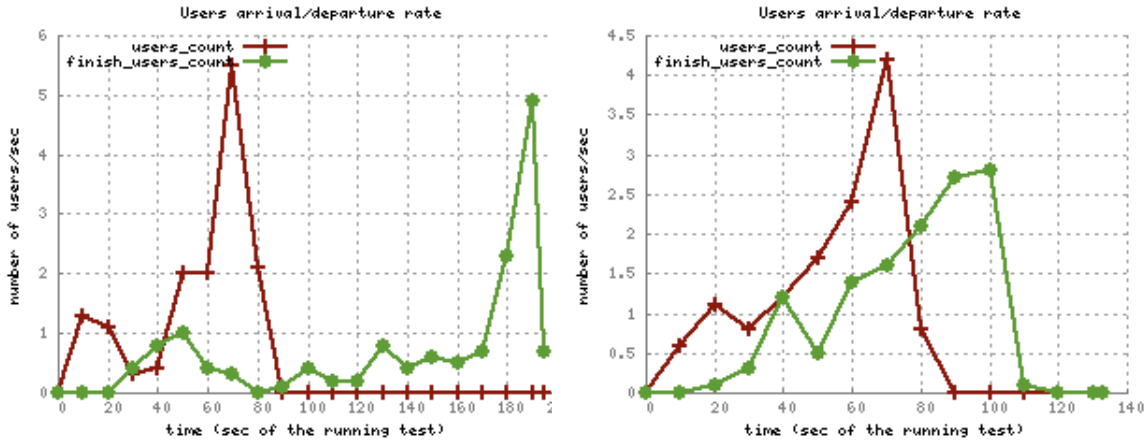
improved query scheme. Besides, HTTP 503 are observed when we use the original query logic.



(a) Original Query **(b)** Query Improvement

Figure 15: Request Duration: Query Improvement

Before modifying the query implementation, request duration starts to exceed 2000ms from 70s after executing the script, while with the improvement in query, the request duration stayed below 250ms all the time.



(a) Original Query **(b)** Query Improvement

Figure 16: User Number: Query Improvement

Every simulated user would sign out after receiving the response they are expecting. With the query improvement, we can see that users would stay for around 20s on average. However, we may notice that using the original query logic, almost all users stay till the server is shutdown, which may well indicate a timeout. This can also be seen in the response duration figure, where the average response time goes up to over 10000ms after 100s.

7 PAGINATION

Due to the large number of entries being displayed on screen, the rendering speed is pretty slow, and users may only need the top-10 entries. Therefore, instead of displaying all the information at the first time, we could use pagination to limit the amount of data being displayed on screen at any given time, resulting in reducing the rendering time.In our application, we use “will_paginate" gem for paginating query results from database and also array objects.

The log results are shown as follows:

Pagination Response Time

	With Pagination	Without Pagination
Search for all adventures	Completed 200 OK in 185ms (Views: 160.7ms | ActiveRecord: 17.4ms)	Completed 200 OK in 3473ms (Views: 3312.0ms | ActiveRecord: 118.8ms)

From the table above, we can see that a significant reduction in view rendering as well as a dramatic reduction in database query are performed when searching for a large number of adventures.

8 SCALING UP AND OUT

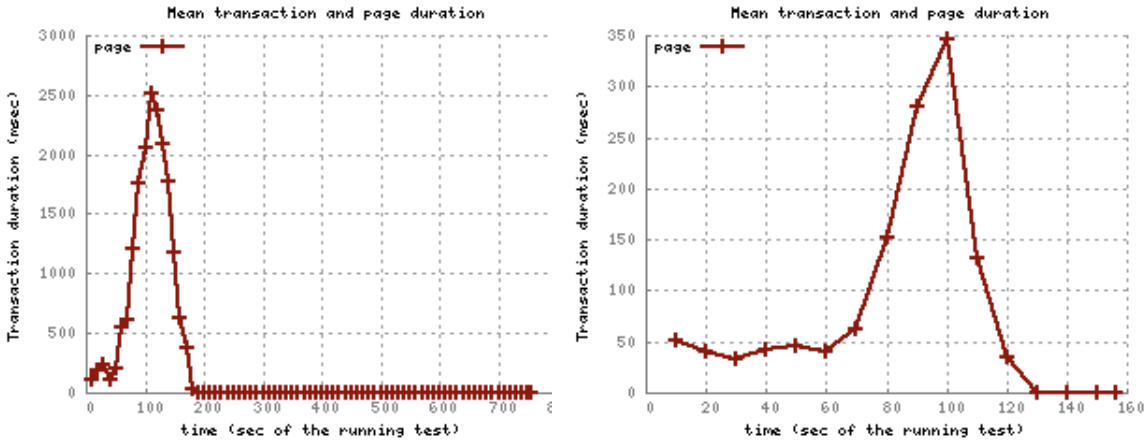
8.1 Vertical Scaling

Vertical Scaling is a quite straightforward way to scale a web application, however, it hits limits eventually. Through utilizing a more powerful server machine (faster CPU, larger Memory size, larger SSD and faster Network I/O), we can easily push our application to handle larger user population and heavier workload. We have investigated AWS M3 instances in this project. According to the AWS documentation, the M series instances provide a balance between memory, computation power, and network resource which are used for small and mid-sized databases, data processing tasks that require additional memory, caching fleets, and for running backend servers for SAP, Microsoft SharePoint, cluster computing, and other enterprise applications. The M3 instances are compared in the table below.

EC2 Instance

Model	vCPU	Mem (GB)	SSD Storage (GB)
m3.medium	1	3.75	1x4
m3.large	2	7.5	1x32
m3.xlarge	4	15	1x40
m3.2xlarge	8	30	2x80

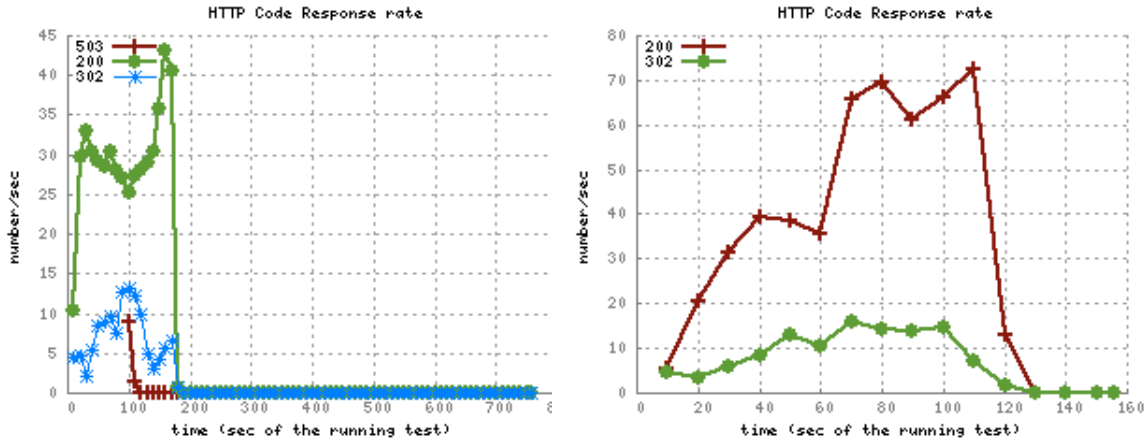
We have applied the same load testing script upon a **m3.large** instance and a **m3.xlarge** instance. Obviously, **m3.xlarge** instance outperformed the **m3.large** instance, and the results are shown as below.



(a) Original **(b)** Vertical Scaled

Figure 17: Transaction Duration: Vertical Scale

The figure on the left shows the mean transaction and page duration from the **m3.large** instance, and the right one shows the data from the **m3.xlarge** instance. We can see the peak mean duration shrank from 2500ms to 350ms by vertical scaling.



(a) Original **(b)** Vertical Scaled

Figure 18: HTTP Response Code: Vertical Scale

From the HTTP response code we can see that after 100 seconds of the load test, the **m3.large** instance started to generate HTTP 503 status which indicates server failure. However, the figure

on the right shows that **m3.xlarge** instance worked well at the time of 100 seconds after the start of load test. The benefits of vertical scaling is quite intuitive. Since it is very easy to vertically scale a web application (if the budget is abundant), it is a practical way to handle larger workloads. But the vertical scaling method has a clear limit. When the workload is too large for the most powerful server, vertical scaling will be no longer applicable. In contrast, horizontal scaling is more preferable since it in general has no limit in terms of scalability.

8.2 Horizontal Scaling

The idea behind horizontal scaling is to use multiple instances to share the heavy work load. A load balancer will allocate the requests to the array of instances which respond to the requests. The architecture of horizontal scaling is shown in the figure below.

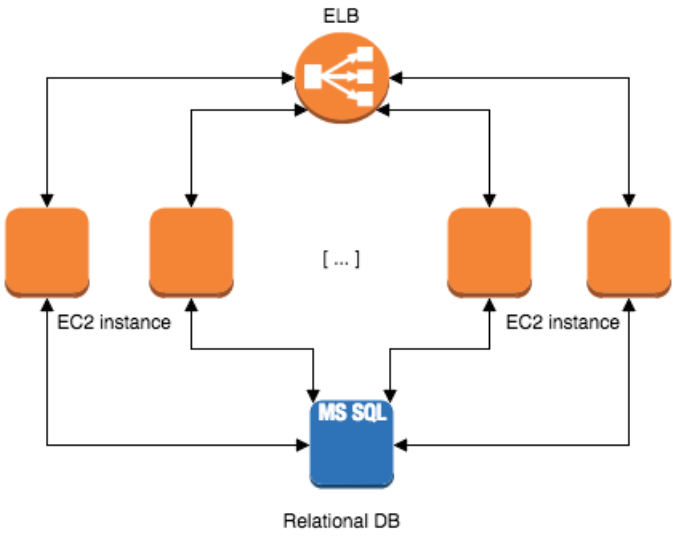
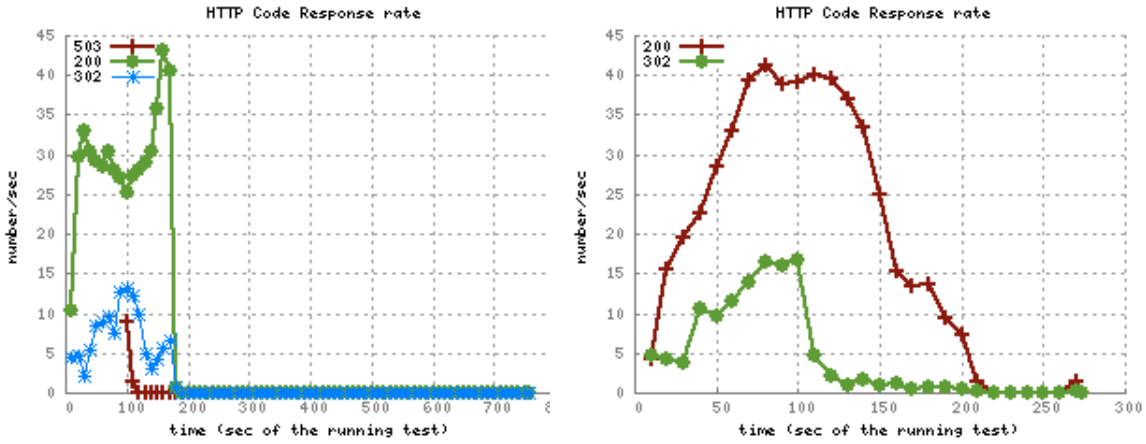


Figure 19: Scale Out

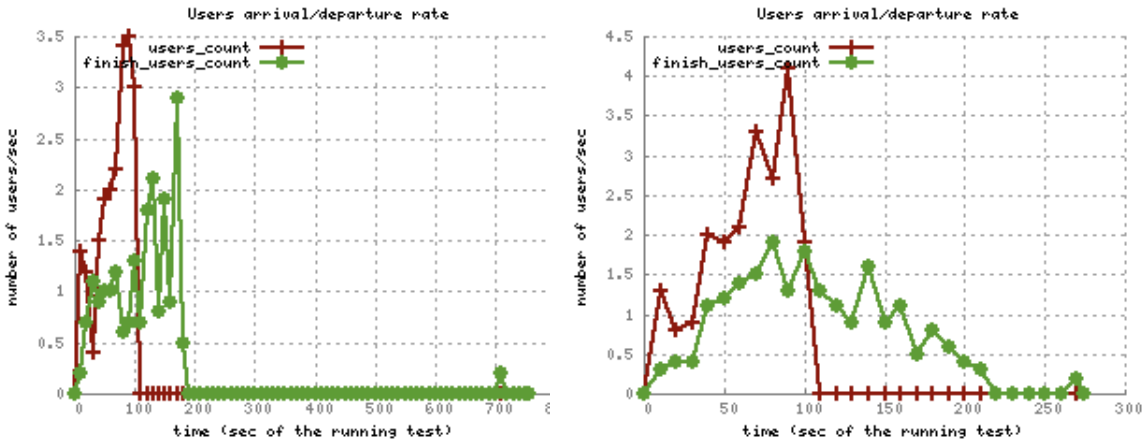
The ELB(Elastic Load Balancer) will direct traffic to different EC2 instances. But the database is still a centralized node in the data flow, so the performance is not going to be scaled linearly with the number of instances. We have created a Multiple Instances Nginx Passenger server (4 instances) using **m3.large** instances to compare with the performance of a single **m3.large** instance. The same load testing script was used in this test. The results are presented below.



(a) Original **(b)** Horizontal Scaled

Figure 20: HTTP Response Code: Horizontal Scale

The left figure shows the HTTP response code generated by the server. We can see that the server fails at $t = 100s$ since code 503 appears. The figure on the right shows the HTTP response code output from the horizontally scaled server. There are no code 503 in the result which indicates that horizontal scaling enables the application to deal with larger workload.

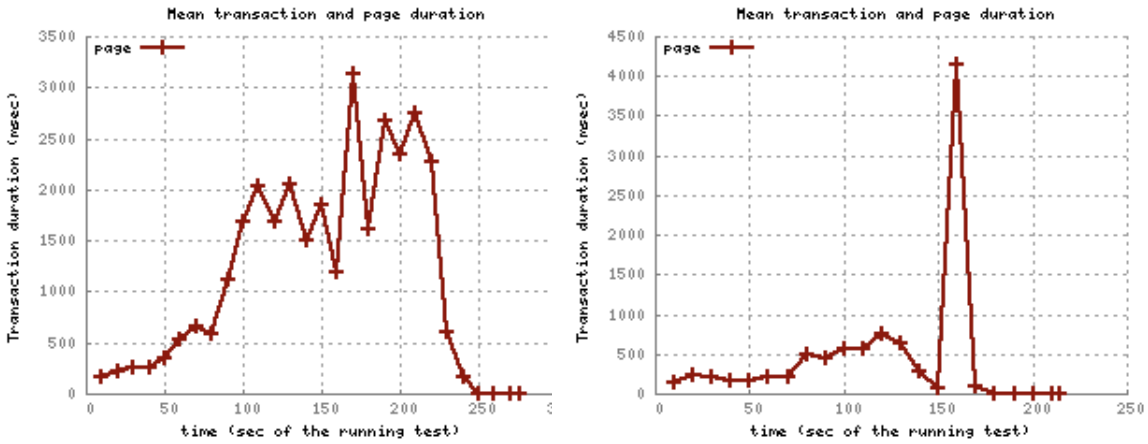


(a) Original **(b)** Horizontal Scaled

Figure 21: User Number: Horizontal Scale

The two figures above report the user/finished-user status of both single **m3.large** instance and

horizontal scaled instance. We can see that there is no great optimization for horizontal scaling scenario in terms of request response latency. This is because the bottleneck in this case is not on the application server but on the centralized relational database. Even the capacity of handling concurrent requests is higher, the database is overwhelmed by intensive queries. In order to get through the database bottleneck in the testing workload, we have deployed a horizontally scaled server with data caching. The large number of concurrent requests will be taken care of by the load balancer and the high volume same queries will be cached to alleviate the database stress. The result of this solution is shown below.



(a) Horizontal Scaling Without Data Cache **(b)** Horizontal Scaling With Data Cache

Figure 22: Transaction Duration: Horizontal Scale

The figure on the left shows the mean transaction and page duration of the horizontally scaled server without data caching, on the other hand the right one shows the corresponding result of data cached horizontal scaling server. The reduction of mean transaction duration is obvious, thus proves the benefits of data caching.

9 FUTURE WORK

From all the methods we applied on the web services, we can see many great improvements. However, there are still some aspects we can work on to make the website more robust and reliable.

First, even we used weak ETags in the client cache and we can get 304 response code, in the load test we still can not get any 304 response from the server. We could do more research on that to improve the client cache further.

We can try other different improvement methods on the website. For instance, we can test the influence of adding indices to the database, like using foreign key to make improvement on query search. We can also attempt to adjust our javascript code to see whether there would be some other implementations that cost less.

When testing scaling out, i.e., horizontal scaling, we increased the number of App Servers to allow the website to handle more requests at the same time. On the database side, we were still using single database, which could be the bottleneck of our website. If we could use sharding on database, we could have evaluated the effects that multiple databases could bring about in different situations. We can assess robustness and stability of the website under different read-write ratio.

10 CONCLUSION

Through the course we have learned how to build a Ruby on Rails web application and techniques to make it scalable, available and secure. We have gained hands on experience in Agile Software Develop and deploying our application to Amazon Web Service. Moreover, during the second half of the quarter we have performed quite intensive load testing on our application and learned a lot of lessons.

We have explored several common practices that we have learned through the course to improve the scalability and performance of our web application. We have seen a significant performance gain brought by caching technique. More specifically, both server side fragment view caching and data caching presented appealing results in terms of shorter response time and lighter database stress. However, we struggled on seeing the expected performance gain on client side caching from load testing. Although we have achieved client caching (304 response code and much shorter response time) on our local machine, we didn't see the promising client caching effect in load testing environment. The reason may be the configuration issue with Tsung to perform client

caching properly. Besides caching, we also optimized our database queries to minimize the number of queries in friends search function. The result was quite amazing. We have compressed 100 queries into 2 queries to get the same result. We have also managed to optimize the performance by pagination and vertical scaling as well as horizontal scaling.

We were all new to web development at the beginning of the quarter, so we made efforts to learn the whole new tool sets. The outcome of the course is very rewarding to us. We have not only learned the fundamental concept of web techniques, but also got exposed to the cutting edge technologies of the internet industry. We would like to sincerely thank Professor Mutz and Li Zhang for assisting us throughout the quarter.