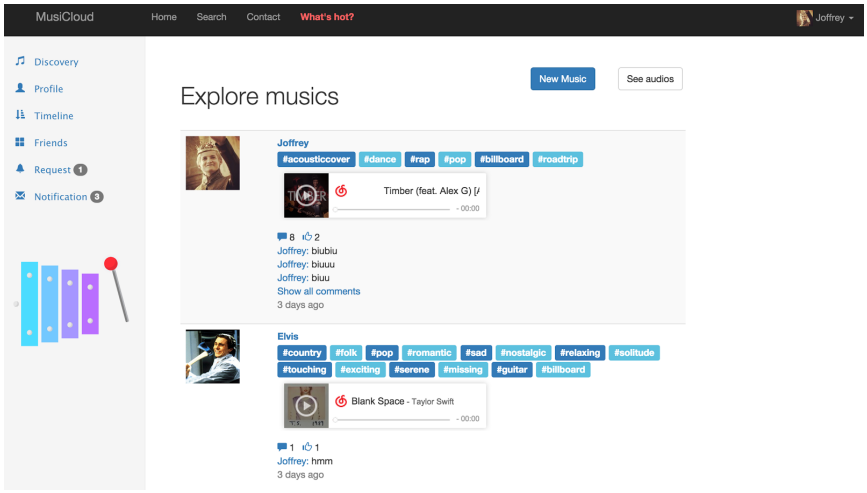
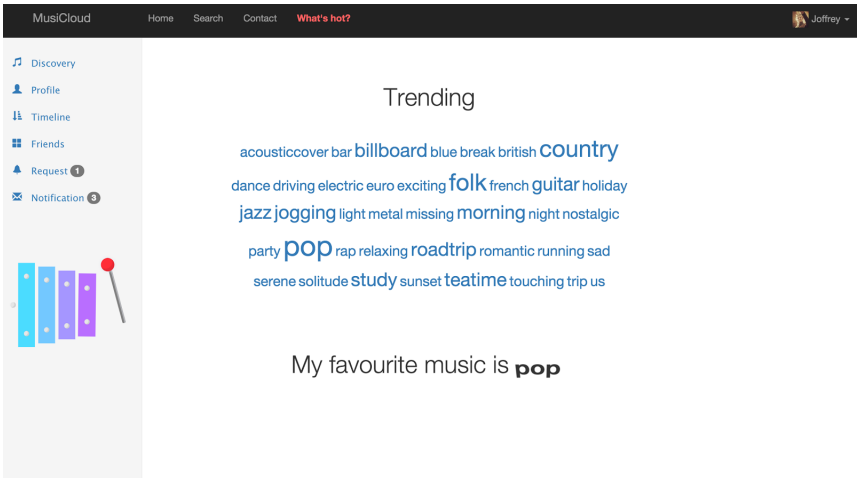
MusiCloud

Discover, share and enjoy music around the world





UCLA, CS219 – Spring 2015

Team: Michelangelo
Yuntao Zhao
Shuyuan Wang
Qi Shao
Mengnan Jia
Jie Yu

Contents

1. Introduction

MusiCloud is an online web application that focuses on the following two parts:

Music sharing

Are you tired of listening to radios and billboards over and over again? You'd either have to put up with a lot of songs that are noises to your ears, or listen to your favorite songs over and over again until you get annoyed of it. Then you think you've had enough, so you go and spend a lot of time hunting for new songs to give your playlist a new look. This could be really an annoying experience.

But don't worry! Our MusiCloud is the perfect solution to your dilemma. At MusiCloud, you can share songs with your friends, and like and comment on your friends' posted musics. What's more, you can even share playlists of songs, which shows your friends your music collections and your music tastes. Therefore, you can simply listen to your friends' playlists with similar music tastes when you want to listen to some music while you're working or studying, without having to live with your old playlists that you have listened to a million times, or take the trouble to discover new good songs.

Moreover, if you do not have many friends, MusiCloud still has a handful for you. We have this awesome feature called "What's hot", which shows you the trending hashtags of all the musics that our users share on MusiCloud in a “tag cloud”, and you can click into those hashtags to see all the relevant trending musics. And for example, if you like Jazz, you can click into the tag "Jazz" and find all Jazz music.

Audio sharing

Some of you have talents in singing or playing instruments. You might want to show your talents to your friends, but all you can do is to take the pains to set up your phone or laptop to record a video of you and upload it to Instagram or Facebook. This is troublesome. But with MusiCloud, you can simply record your voice with your phone or any device and upload this piece of audio to our website to share it with your friends. Or, you might also record a piece of live music at a concert or a show and share it on MusiCloud.

Also at times you might want to express how you feel or share what you are up to at the moment. Many people type it out and tweet it, but many others remain silent and do not bother to do it. With MusiCloud, you can simply record a voice memo with your phone and share it on our website. This is much like an audio version of Twitter.

The idea of MusiCloud comes from the current trend that people love to share their pictures, ideas and experience on varieties of platforms and apps, and listen to and discover new songs on

some music websites. In summary, MusiCloud serves as a platform for users to share their favorite songs or playlists, share their own voice or talents in playing instruments, to express their feelings when they listen to the music, and to find someone whose tastes are similar to theirs.

MusiCloud is implemented by Ruby-on-Rails (Ruby 2.2.1 and Rails 4.0.0). We use a RDBMS--MySQL, as the backend data store. After developing the functionalities, we have conducted several scalability testing and optimizations. The rest of the report discusses the detail about the development, functionality and scalability of MusiCloud.

2. Development Model

Our MusiCloud is conducted in an agile team where we build our own scalable web application and services using fundamental web technologies and the Ruby on Rails framework. We followed the sprint schedule and finally built our MusiCloud step by step.

First of all, we created our github repo (https://github.com/scalableinternetservices/Michelangelo). The branching and merging capabilities of Git enable us to merge our own commits, or work on separate branches, focusing on a main topic and avoiding conflicts during our development. Every week we met twice or more to discuss our problems, present our solutions, and brainstorm our ideas, so that we can find the potential improvements and give advice to each other's work.

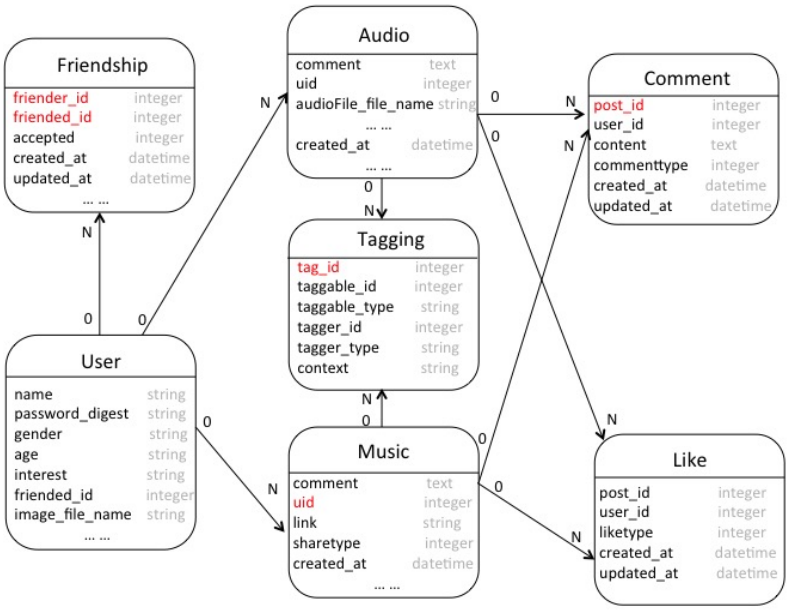
While building our web services, we have encountered lots of difficulties, and we discussed together, referred to some materials and posted questions in Piazza to go through these difficulties. At the end of our development, we gained a significant amount of hands-on experience on building web applications and scalable internet services, being able to tackle the scalability and fault-tolerance concerns.

3. Project Description

3.1 Application Architecture

Ruby on Rails is an open source web application framework written in Ruby. It is a model–view–controller (MVC) framework, providing default structures for a database, a web service, and web pages. Our MusiCloud uses the MVC model. For example, once the Views send a need, we use Controllers to query the Models for specific data and organize that data into a proper form. Then the Models will maintain the relationship between Object and Database, manipulating database records and sending back to Controllers. Finally Controllers will make a decision and trigger the Views to present the data in a particular format.

Here are the details about our database and MVC architecture. MusiCloud has 9 models and the relation between them is as follows:



3.2 Features

3.2.1 Songs & Playlists

Users can share their songs & playlists, post comments and likes on their friends' songs & playlists, and play the songs & playlists on MusiCloud.

3.2.2 Audios (voice memo/live music)

Users can upload their audios, post comments and likes on their friends' audios, and play the audios on MusiCloud.

3.2.3 Social Network (friendships, comments, likes)

MusiCloud is basically a social network, where you can edit your profile, maintain your friendships, post comments and likes, and only see your friends' posts.

3.2.4 Hashtags & Trending

Each song/playlist will have some hashtags that the user adds, and by clicking into the hashtags you can see all the songs/playlists with the same hashtag. We also have this awesome feature called "What's hot", which shows you the trending hashtags of all the musics that our users share on MusiCloud in a "tag cloud", and you can click into those hashtags to see all the relevant

trending musics. And for example, if you like Jazz, you can click into the tag "Jazz" and find all Jazz music.

3.2.5 Animations and styling

MusiCloud has several animations to give our users a better user experience, such as the animated headlines and the animations of Xylophone on the sidebar, etc. We have also styled our web pages to make it look prettier and more user friendly, such as modifying the styles of hashtags, sidebar and navigations, tag cloud, etc.

4. Load testing configurations

4.1. Tsung

We use Tsung to simulate large number of simultaneous users to test the scalability and performance of our application that is deployed on amazon EC2. We have performed load tests on vertical and horizontal scaling, as well as on different type of optimizations. We created separate branches for different optimizations on Github and deployed them on EC2 as app servers, and then deployed another instance using Tsung template to launch load testing toward the app servers.

Tsung's work load: 30sec/phase

Phase	1	2	3	4	5
Users/sec	1	2	4	8	16

4.2. Critical path

In order to simulate user's standard behaviors, we created a testing script as the critical user path for the Tsung instance. In our script, Tsung will have 5 phases, 30 seconds each, generating 1, 2, 4, 8, 16 new users per second arriving at the app server, so we will have up to 500 concurrent users around 150 seconds of running time. Every new user has 20% probability to be a new user that needs to sign up, and 80% probability to be a returning user that only needs to log in.

Here is our critical user path:
For 20% new user:

1. Sign up
2. Waiting for up to 3 seconds
3. Log in
4. Waiting for up to 3 seconds

5. Edit profile, upload avatar
6. Waiting for up to 3 seconds
7. Browse music
8. Waiting for up to 3 seconds
9. Search for some interesting user
10. Waiting for up to 3 seconds
11. Delete created user

And for 80% returning user:
1. Log in
2. Waiting for up to 3 seconds
3. Browse music
4. Waiting for up to 3 seconds
5. Post a new music share
6. Waiting for up to 3 seconds
7. Post a new Comment
8. Waiting for up to 3 seconds
9. Like a music
10. Waiting for up to 3 seconds
11. Post and upload an audio
12. Waiting for up to 3 seconds
13. Browse audios
14. Waiting for up to 3 seconds
15. Browse my Timeline
16. Waiting for up to 3 seconds
17. Search for some interesting user
18. Waiting for up to 3 seconds
19. Delete posted music

We used the above testing scripts to simulate our application's standard incoming users and their behaviors. Then we used Tsung to analyze our application's performance under different configurations.

4.3 Load tests initialization (seeds)

We have an ec2_initialize file on our branch which is automatically loaded when EC2 instance launches. In that file, we run rake db:seed once, and create 1000 users, 50000 musics, 50000 audios and totally 100000 comments on our database. This large dataset is used for the Tsung load tests.

In order to deal with the load balancing on ELB, we need to avoid writing to database simultaneously by several app servers which leads to naming conflicts. Therefore, our seeds file has a mechanism of asking the database if there exist more records than we need; if not, the seeds file will continue creating new users, musics, audios and comments. Also, we maintain a counter of existing users as the new user' name to avoid naming conflicts.

5. Optimizations

The app servers in this section were deployed on m3.2xlarge. This does not apply to other sections.

5.1 Database design & optimization

5.1.1 Initial design & pagination

Database is the foundation of our application, so we paid lots of attention to it when initially designing it in the following ways.
1.Table design: All our tables are in Boyce–Codd Normal Form (BCNF)
2. Query design: All queries in controllers and models are non-trivial and straightforward
3. Avoid repeated query: If a certain query is needed in multiple actions in a controller, we write a before-action method for it in the controller.
4. Pagination: 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.

After our initial testing and examining the logs, we noticed that a lot of run time was spent in ActiveRecord. To handle this issue, we came up with the following two ways.

5.1.2 Adding indices

We noticed that most of our models didn't take advantage of database indexes. By having an index on key columns in the database, we can significantly reduce the amount of time spent in accessing and searching in our database. The tradeoff is to have marginally slower writes, but since our application is mostly based on reads, this should not affect the application negatively in any significant way.

Therefore, we implemented indexing on all the foreign keys, and saw drastic improvements in query times throughout the entire application.

```ruby
class AddIndex < ActiveRecord::Migration
  def change
        add_index :comments, :post_id
        add_index :friendships, :friender_id
        add_index :friendships, :friended_id
        add_index :musics, :uid
        add_index :unreadcomments, :commenter
        add_index :unreadcomments, :user_id

  end
end
```

5.1.3 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. To solve this problem, includes method can be used. For instance, in the music controller, we can preload comments like this:

```ruby
def index
  current_user = User.find(session[:user_id])
  # byebug
  @musics = current_user.discover_musics.includes(:allcomments).includes(:top3comments).paginate(:page => params[:page], per_page: 5)
  @commenttype = 0
  @liketype = 0
  respond_to do |format|
    format.html
    format.js
  end
```

And we need to write the following in the music model to change the allcomments and top3comments methods into relations to make it work.

```ruby
has_many   :allcomments, ->{where(post_id: self.id, commenttype: 0 ).order("created_at DESC")}, class_name: 'Comment'
has_many   :top3comments, ->{where(post_id: self.id, commenttype: 0).order("created_at DESC").limit(3)}, class_name: 'Comment'
```

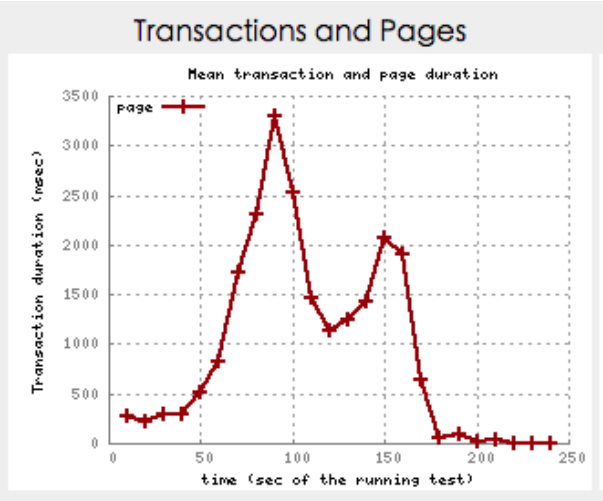
However, here comes the problem: we also need to change the discover_musics method into an Active Record relation. It is not as trivial as we do in the above two methods.

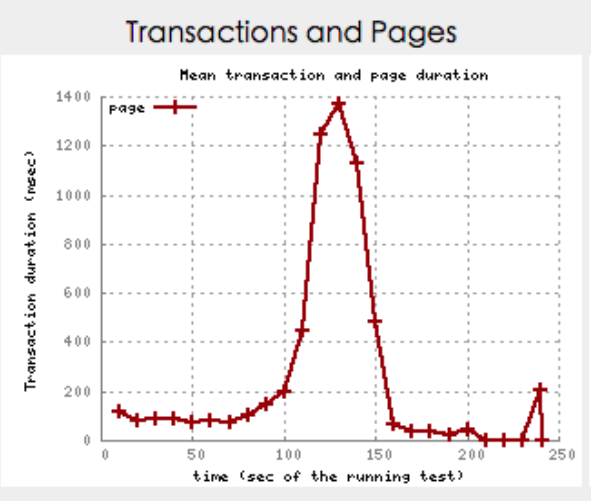
```ruby
def discover_musics
  valid_ids = self.friends.map {|friend| friend.id}
  valid_ids.push(self.id)
  Music.where(:uid => valid_ids).order("created_at DESC")
end
```

We had a hard time working on it but could not manage to figure it out, and we have seeked help from Piazza and the course instructor but neither was able to help us out. We will continue working on it, and we will greatly appreciate any further help or advice.

5.1.4 DB optimization results

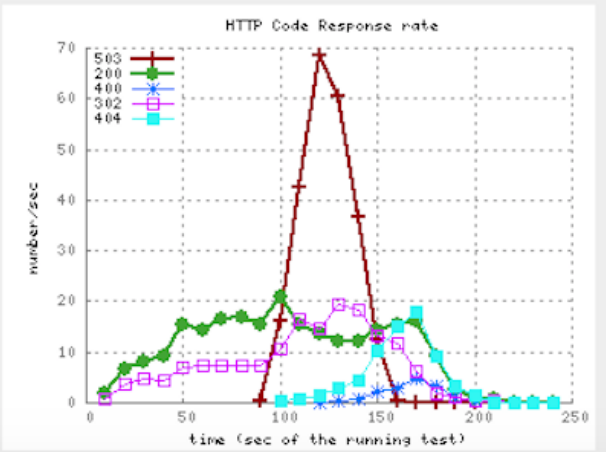
Performance on master vs Performance on master_db_optimization:

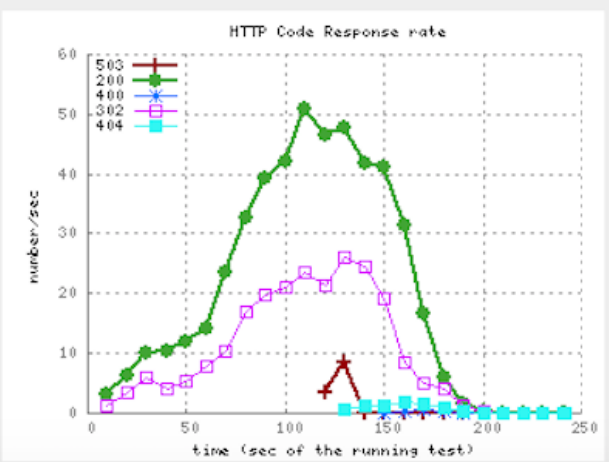




From the above two figures, we can see that:
Without any optimizations, the application can handle about 2-4 users arriving each second.
With DB and SQL optimizations implemented alone (without other optimizations), the server can handle up to 16 new users arriving each second easily.

Comparing the HTTP code response rates:





From the above two figures, we can see that:
Without any optimizations, 404 and 503 error codes started to occur at around 90 seconds, and the number of 200 and 302 response codes indicating success are very low.
With DB and SQL optimizations implemented alone (without other optimizations), 404 and 503 error codes started to occur at around 120 seconds, and there are much more 200 and 302

response codes than the ones in the master branch before the server goes down, meaning that all of the users are doing well before the server hits its limit.

Therefore, we can conclude that there has been a significant improvement in the performance and a great boost in the server's capability, because of the database and SQL optimizations.

5.2 Server-side Caching

Our server is repeatedly responding to HTTP requests, which requires computations and I/O that can be very expensive. However, there is a great deal of similarity between those requests, so in order to create responses that are repeated, we have implemented various server-side caching methods.

5.2.1 Memcached

Memcached is a commonly used implementation of a remote cache server. It has the following advantages:

- Keeps a cache in memory
- Accepts TCP connections and returns lookup requests
- Distributed key-value store
    - Keys can be up to 250 bytes, values can be up to 1MB
    - Can scale horizontally
- When it runs out of space, it uses a simple LRU mechanism to make more space
- Lightweight features, everything is constant time.

Therefore, we have chosen Memcached to store our cached contents.

5.2.2 Fragment caching

Fragment caching caches a portion of a rendered view for reuse on future requests. We have implemented fragment caching to cache the posts (music or audio, together with comments, likes etc.) in the discovery pages and timeline pages, as well as the users in the search page for users.

By doing this, when a post needs to be loaded on a web page, if its content, comments, likes and the user who posted it have not been changed, then the post will be simply retrieved from the cache store using its cache key without having to be generated by creating a bunch of html codes and calling several SQL queries. Similarly, when a user needs to be loaded on a web page, if its profile picture has not been updated, then the user will be simply retrieved from the cache store using its cache key without having to be generated by creating a bunch of html codes and calling several SQL queries.

5.2.3 Russian doll caching

Russian doll caching nests cache fragments. So fragment caching of posts and users are nested inside another layer of caching, which caches the whole table on the page, while the posts and users are only the rows in the table.

For example, the whole Discovery page (which shows all the posts from the user's friends) is cached, and when it needs to be loaded it is simply retrieved from the cache store if none of the user's friends' posts have been updated. Similarly, the whole user searching page (which shows the searching results of a search key, searching for another user) is cached, and when it needs to be loaded it is simply retrieved from the cache store if none of the searched users have not been updated. The timeline pages for music and audios respectively are also cached this way.

The overall structure is illustrated below:

```
cache whole music/audios/users table:
        for each row (for one music/audio/user):
                cache this row
        end
end
```

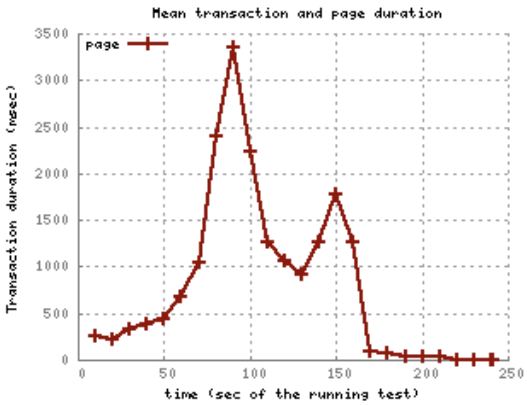
5.2.4 SQL caching

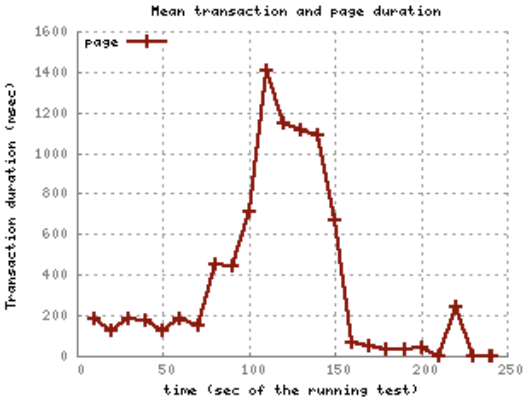
Query caching is a Rails feature that caches the result set returned by each query so that if Rails encounters the same query again for that request, it will use the cached result set as opposed to running the query against the database again.

For example, if "@musics = Music.all" is run once, and later encountered again, it won't be run again. We do not need to do any configurations as Rails automatically implements this feature for us.

5.2.5 Server side caching results

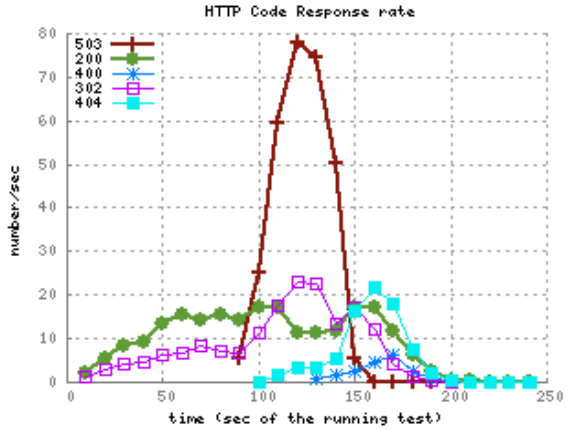
Performance on master vs Performance on master_server_side_caching:

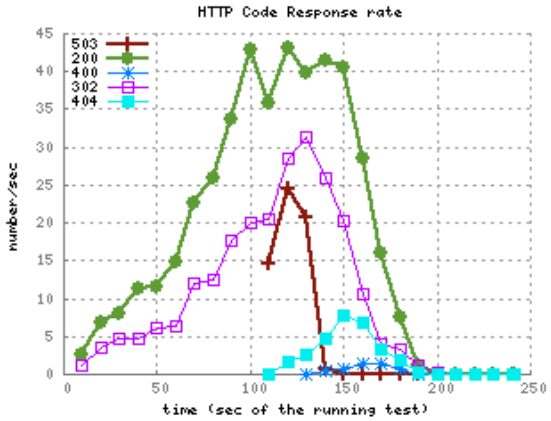




Without any optimizations, the application can handle 2-4 users arriving each second.
With the server-side caching implemented (using Memcached) alone (without other optimizations), the server can now handle up to almost 16 new users arriving each second.

Comparing the HTTP code response rates:





Without any optimizations, 404 and 503 error codes started to occur at around 90 seconds, and the number of 200 and 302 response codes indicating success are very low.
With the server-side caching implemented (using Memcached) alone (without other optimizations), 404 and 503 error codes started to occur at around 110 seconds, and there are much more 200 and 302 response codes than the ones in the master branch before the server goes down, meaning that all of the users are doing well before the server hits its limit.

Therefore, we can conclude that there has been a significant improvement in the performance and a great boost in the server's capability, because of the server side caching.

5.3 Client-side Caching

To reduce the number of requests the server needs to handle, the browser can cache the recently accessed pages and use them later under some circumstances. Data that is read from the server is kept in the cache, and if the same data is read again, it is taken from cache instead of from the server.

5.3.1 HTTP caching

In rails, it is simple to use client-side caching. We use client-caching in the music page, audio page search page and personal timeline page. In the code, we check if the requested page has changed since last time using “if stale?”. If it has changed, we return the updated page; otherwise the browser can read the cached page instead.

5.3.2 Client side caching results

Theoretically, the client side caching can reduce lots of networking traffics caused by returning accessed pages to users repeatedly. However, according to the Tsung test reports, performing client side caching has not significantly improved the performance. We think the reason is that Tsung cannot correctly test on the client side caching (does not support http caching).

5.4 Javascript & CSS optimizations

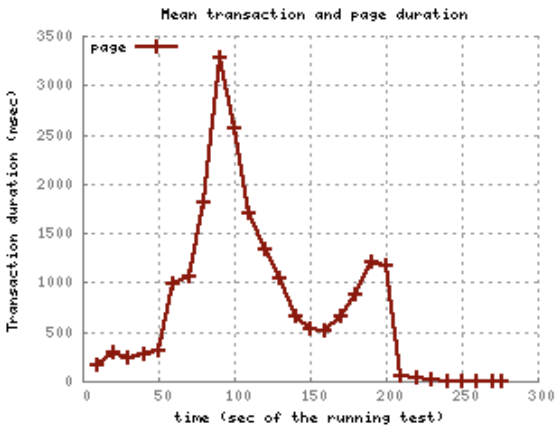
We have used Turbolinks to optimize JavaScript and CSS. Turbolinks makes following links in your web application faster. Instead of letting the browser recompile the JavaScript and CSS between each page change, it keeps the current page instance alive and replaces only the body (or parts of) and the title in the head.

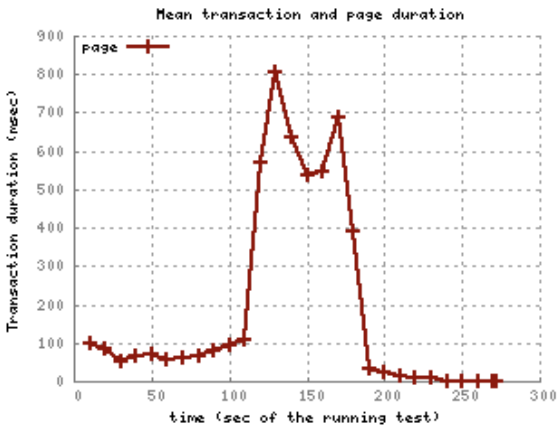
Generally speaking, the more CSS and JavaScript we have, the bigger the benefit of not throwing away the browser instance and recompiling all of it for every page. In any case, the benefit can be up to twice as fast in apps with lots of JS and CSS. Our MusiCloud used lots of JS and CSS because of the animations, user interactions and front end stylings, so Turbolink will greatly improve the speed of our web app.

However, with Turbolinks, pages will change without a full reload, so we can't use jQuery.ready() to trigger some of our JavaScript code. We have resorted to the “jquery-turbolinks” gem to solve this problem.

5.5 Overall optimization results

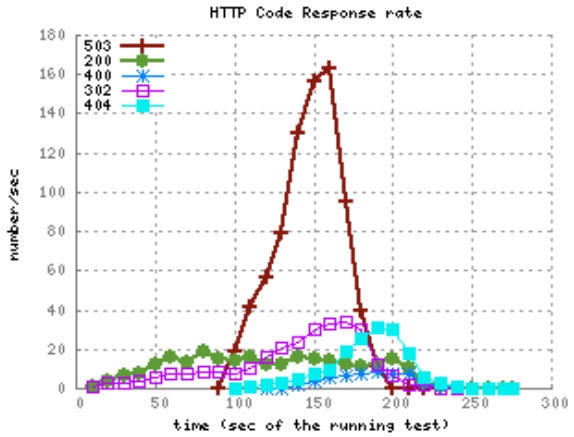
Performance on master vs Performance on master_final_optimization:

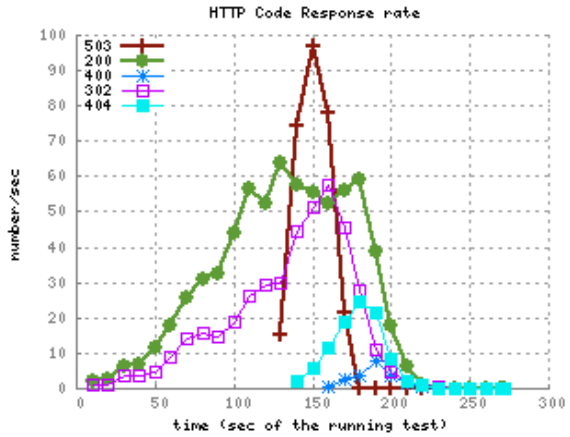




Without optimizations, the application can handle 2-4 users arriving each second.
With all the optimizations , the server can now handle up to almost 16 new users arriving each second, and the peak transaction duration is even less than 1 second.

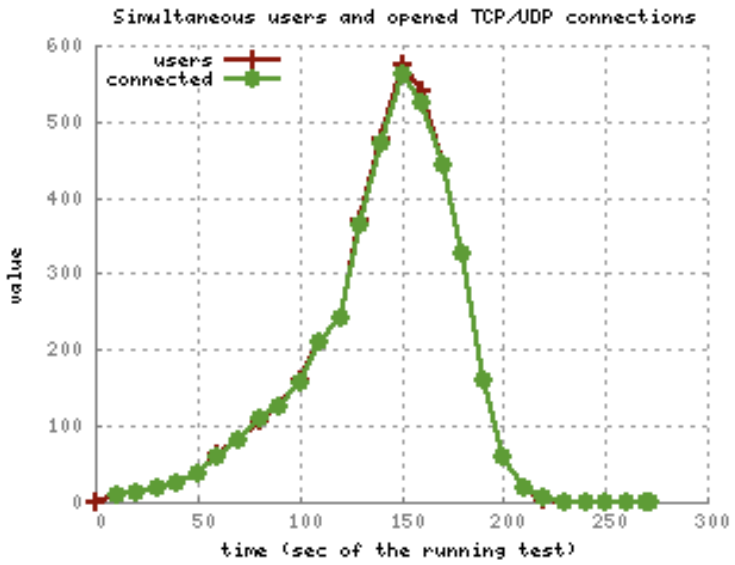
Comparing the HTTP code response rates:





Without optimizations, 404 and 503 error codes started to occur at around 90 seconds, and the number of 200 and 302 response codes indicating success are very low.
With all the optimizations, 404 and 503 error codes started to occur at around 130 seconds, and there are much more 200 and 302 response codes than the ones in the master branch before the server goes down, meaning that all of the users are doing well before the server hits its limit.

Number of simultaneous users:



With all the optimizations, our server can serve up to 600 simultaneous users before it goes down.

Comparing load testing statistics:
on master:

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.18 sec	1.22 msec	23.1 / sec	2.13 msec	1339
page	3.27 sec	3.46 msec	213.7 / sec	0.88 sec	14748
request	3.15 sec	3.46 msec	225.7 / sec	0.84 sec	15444
session	1mn 39sec	25.81 sec	11 / sec	55.31 sec	881

on master_final_optimization:

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.12 sec	1.22 msec	19.4 / sec	2.41 msec	995
page	0.81 sec	8.34 msec	196.8 / sec	0.42 sec	14496
request	0.75 sec	8.34 msec	208.9 / sec	0.41 sec	15192
session	1mn 14sec	19.98 sec	16.6 / sec	47.47 sec	860

We can see that the time taken in the connections and requests are much shorter. Our optimizations have decreased the average time for responding to requests by more than 50% compared to the results before optimization.

Based on all of the above results, we can conclude that there has been a significant improvement in the performance and a great boost in the server's capability, because of all our optimizations.

5.6 Future optimizations

We plan to take advantage of the gem “Octopus” to perform sharding with multiple databases. This would be very powerful if we have zero relations across shards, and it works best when partitions grow with usage. But we should be very careful sharding pages and databases so that it does not inhibit feature developments.

We also plan to test our client side caching on other methods than tsung, so that we can make sure it works. We are actually thinking about publishing our website, so we should make sure it is really scalable before it goes public.

Lastly, besides scalability, we want to make sure that our website has no security issues such as SQL injection, Cross-site Scripting, and issues about firewalls, etc.

6. Vertical Scaling

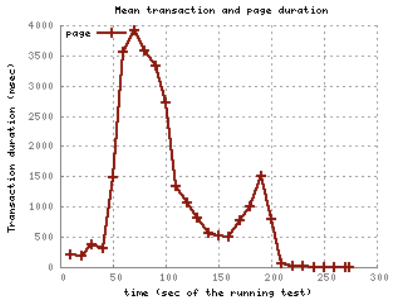
Vertical scaling means buying “bigger” and more expensive hardwares and scaling up, which adds more CPU and memory or changes from HDD to SSD in our existing machines. We tried three different instance types on EC2: M1 and M3 to test the application’s vertical scalability. For M1, we used both medium and large; for M3, we use XXlarge instance.

These instances all provided a good balance between computational power, memory and network resources, and M3 for sure had performances that were much better than M1’s. This is because the M3’s SSD-backend storage is delivering a higher I/O performance and also has more CPU cores and larger memory which provide more computational power. The model instances are compared in the table below.

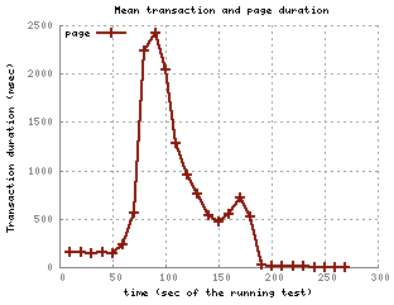
model	CPU	memory	Storage	SSD	Network
M1.medium	1	3.75	410G	none	moderate
M1.large	2	15	2x420G	none	moderate
M3.large	8	30	none	2x80G	high

For each test we deployed using NGINX + Passenger + memcached on EC2, and use above critical user path to launch a load testing from Tsung server. Here are the results:

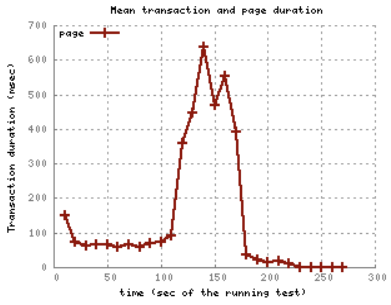
Mean transaction and page duration



M1.medium

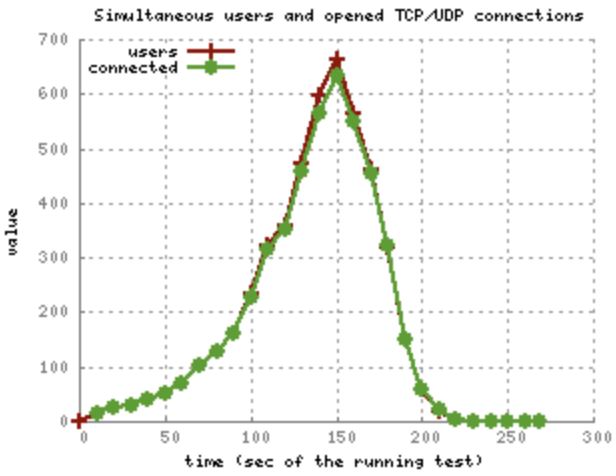


M1.large



M3.XXlarge

In terms of the mean page duration, we observed considerable improvements when vertical scaling our application as expected. M1.medium can not handle the second phase of load test, which has 2 new user arrivals per second and about 30 concurrent users. The M1.large has a better result, which can handle the entire second phase, nearly 100 concurrent users. The mean page duration time decreases from 3500ms to 250ms under 100 concurrent users, which is a huge improvement. It is because we now have two CPUs and much more memories. When it comes to the M3.XXlarge instance, the result is even much better: it can hold up to phase 4 and about 320 concurrent users. It is nearly a 10 times improvement compared to the M1.medium.



Concurrent Users

We have five phases of load testing, each doubling new arrival users, so the number of concurrent users kept increasing from 0 to about 650 and began to drop around 150 second.

Here are the statistics of three tested instances:

m1medium

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.18 sec	1.74 msec	24.5 / sec	2.59 msec	1490
page	3.91 sec	6.24 msec	213.4 / sec	0.93 sec	15192
request	3.67 sec	6.24 msec	224.5 / sec	0.89 sec	15922
session	1mn 19sec	30.96 sec	11.6 / sec	56.27 sec	901

m1large

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.15 sec	1.72 msec	24.5 / sec	3.54 msec	1335
page	2.42 sec	12.20 msec	231.1 / sec	0.69 sec	15552
request	2.27 sec	12.20 msec	245 / sec	0.66 sec	16290
session	1mn 21sec	27.06 sec	16.9 / sec	52.46 sec	927

m3large

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.28 sec	1.26 msec	21.2 / sec	3.35 msec	996
page	0.64 sec	10.24 msec	224.6 / sec	0.33 sec	14976
request	0.59 sec	10.24 msec	239.3 / sec	0.32 sec	15686
session	1mn 11sec	20.30 sec	14.3 / sec	46.03 sec	893

In terms of the mean page duration time, it decreases from 0.93 sec to 0.33sec and reduces 67% response time going from M1.medium to the M3.XXlarge.

7. 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 app servers. We inserted the database on a single m3.2Xlarge stack. Then we created several Passenger application servers, with an NGINX HTTP-server on top of that. For the app servers we used m3.2xlarge 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.

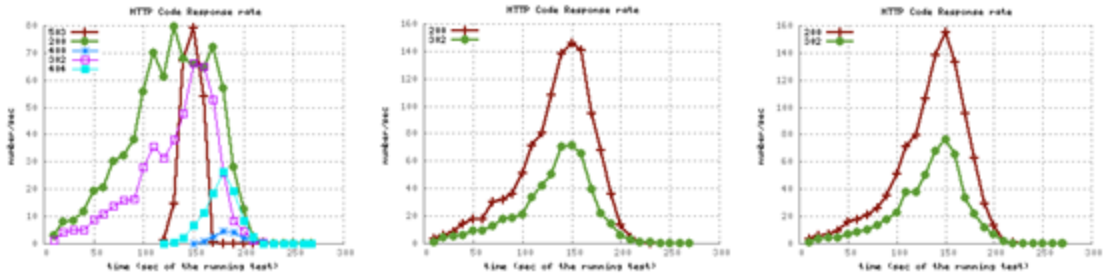
We created three different setups, and tested them using the critical path discussed before. They have a single, two and four app server each. Here are the results:

Mean transaction and page duration

Single app two app four app

According to the mean page duration, we observed a considerable improvement when vertically scaling our application as expected. A single app server can handle the first three phases of load test, which have 4 new user arrivals per second and about 300 concurrent users. Two app servers have a better result, which can handle up to half the forth phase, with nearly 550 concurrent users. The mean page duration time decreases from 700ms to 70ms under 300 concurrent users, which is a huge improvement. When it comes to the Four App server, the result is even better: it can hold until phase 5 finishes and about 600 or more concurrent users maximum. The performance is so good that we need to add extra phase and heavier load testing to find its limit.

HTTP return code rate:



Single app two app four app

The single app server returns 503 error and server goes down due to the heavy load which leads to many “404 page not found” around 120 second – meaning that the single server can serve about 300 concurrent users maximum. However, the two and four app servers do not return these unmoral errors.

Here are the statistics of three tested instances:

Single app

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.10 sec	1.95 msec	17.8 / sec	4.61 msec	922
page	0.89 sec	16.11 msec	205.1 / sec	0.40 sec	15510
request	0.84 sec	16.11 msec	217.9 / sec	0.38 sec	16253
session	1mn 48sec	20.05 sec	16 / sec	46.93 sec	921

Two app

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.23 sec	1.99 msec	16.8 / sec	5.12 msec	927
page	0.34 sec	23.30 msec	205.7 / sec	0.13 sec	15654
request	0.32 sec	23.30 msec	217.5 / sec	0.13 sec	16411
session	1mn 21sec	30.83 sec	13.8 / sec	42.81 sec	926

Four app

Name	highest 10sec mean	lowest 10sec mean	Highest Rate	Mean	Count
connect	0.27 sec	1.79 msec	15.3 / sec	6.63 msec	897
page	0.19 sec	22.71 msec	219.2 / sec	62.66 msec	15084
request	0.18 sec	22.71 msec	230.5 / sec	59.79 msec	15806
session	1mn 36sec	29.03 sec	14.1 / sec	41.59 sec	896

In terms of the mean page duration time, it decrease from 0.40 sec to 0.063 sec and reduce 84% response time going from Single app to the Four app server, which is a huge improvement. It is due to the higher network throughput that multiple instances have. We can see the difference in network throughput between single app and two app servers:

Network throughput

Name	Highest Rate	Total
size_rcv	7.93 Mbits/sec	101.20 MB
size_sent	2.36 Mbits/sec	20.31 MB

Single app

Name	Highest Rate	Total
size_rcv	13.71 Mbits/sec	132.35 MB
size_sent	2.25 Mbits/sec	20.60 MB

two app

8. Conclusions & Course Takeaways

In conclusion, we have completed all the features that we planned to implement, and we have achieved great optimization results. After signing up on MusiCloud, a user can now have the following amazing experiences:

- Search for the existing users and send friend requests to them. Once the other user accepts the request, there will be a friendship between you and that user.
- Manager your profile and friend relationship, handle friend requests and check notifications for comments and likes.
- Share a single song or a playlist by uploading a link from NeteaseMusic, a music website with a large database and extensive collections of music, posting your comment to this song or playlist, and adding hashtags. MusiCloud automatically converts the link to an out－chain music player, which makes it possible to for the shared music/playlist to be played on MusiCloud.
- Record your own voice or some live music with any device, upload your audio file to MusiCloud and share it with your friends.
- Discover the music and audios that your friends share, and post comments and likes.
- Discover all the trending hashtags in a "tag cloud" by clicking "What's hot?". You can click into those hashtags to see all the relevant trending musics.

This project is extremely rewarding to all of our team members. We managed to realize all the original blueprints in our mind, and build and deploy a powerful and scalable web application. But what we learned from this project is much more than this fancy web application.

As we work as a team, it is not an easy job to balance everything well between team members. We had trouble assigning tasks to each team member at the very beginning, but things become easier as we got more familiar with each other's work style and strengths and weaknesses, and as we became more familiar with the Rails framework.

Pair programming is a fresh experience to all of us. It is an agile software development technique in which two programmers work as a pair together on one workstation. One, the driver, writes code while the other, the observer, pointer or navigator, reviews each line of code as it is typed in. In addition to improving the quality of our code, which is the basic advantage of this programming method, we also learned a lot from each other about the way we code, and the logical way of thinking in each other's mind.

Test-driven development (TDD) is another valuable experience for us. It a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. This way of development has given us clear goals of our tasks, pointed out the detailed paths towards the goals, and prevented us from making simple mistakes that we could otherwise have made.

Besides these, there are two more concepts that have really broadened our horizons.

The first one is actually the focus of this course: scalable internet services. It is not the concept about awesome ideas, but about how to make the awesome ideas become true and really able to survive in real life situations. Through lectures and the out-of-class readings, we have learned to realize scalability in different ways and have a better understanding of many other basic web-related concepts.

The other one is about optimization, which we didn't pay much attention to prior to this course. We were used to focusing on achieving functionalities in our prior projects. However, one of the most significant differences between course projects and real-world applications is the actual performance that is worth lots of attention to optimize. In our project, we have optimized our application in many ways and are really amazed by the improvements they made. In particular, query optimizations and caching are two of the cheapest but most powerful ways to improve the performance of web services under heavy loads. And of course both vertical and horizontal scalings can provide great alternatives to optimizations. We believe the power of these optimizations can help us a lot in our future projects and works.

To sum up, we have benefited a lot both from the project itself and from this course where we could learn a lot of knowledge, expertise, and professionalism from an experienced computer scientist from industry.