Analysis Me





# Table of Contents

**1. Introduction**

1. **Motivation**
2. **Related work**
3. **System design**
4. **System implementation**
5. **Experiments and analysis**
6. **Conclusions and future work**
7. **Contribution of each team member**
8. **Introduction**

Our goal is to create a social network for scientists and researchers to facilitate scientific communications and to foster collaboration. The project achieves the following main objectives:

* Each member will have a personal profile page.
* Each member can publish posts.
* Each member can follow other members.
* Each member can comment on others’ posts.
* Each member can search for other members and their posts.

1. **Motivation**

Scientific researches have become a lot more collaborative than before due to advancements in tele-communication technology. There are increasing numbers of cross-state and cross-national research teams working on the same research project. We believe it’s valuable to create a centralized social platform for scientists to share ideas and collaborate with each other.

1. **Related work**

Frontend:

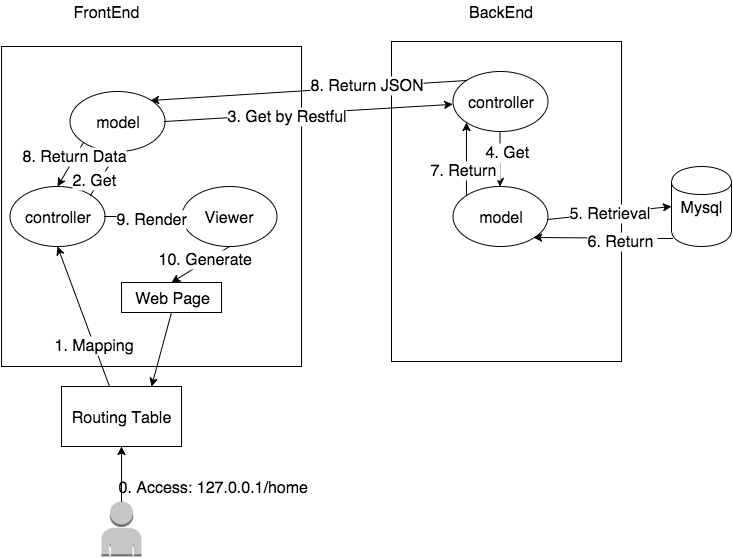
* HTML, CSS, JS, embedded scala
* play! - restful server
* bootstrap - auto resizing
* Ajax, JQuery - Asynchronous communication

Backend:

* play! - restful server
* JPA - Data modeling
* MySql - Database

1. **System design**

The system architecture consists of a frontend server and a backend server. The components of the system and the relationships between the components are described in the following diagram:

**Figure 1. System architecture design.**

1. **System Implementation**

**5.1 Basic functionalities**

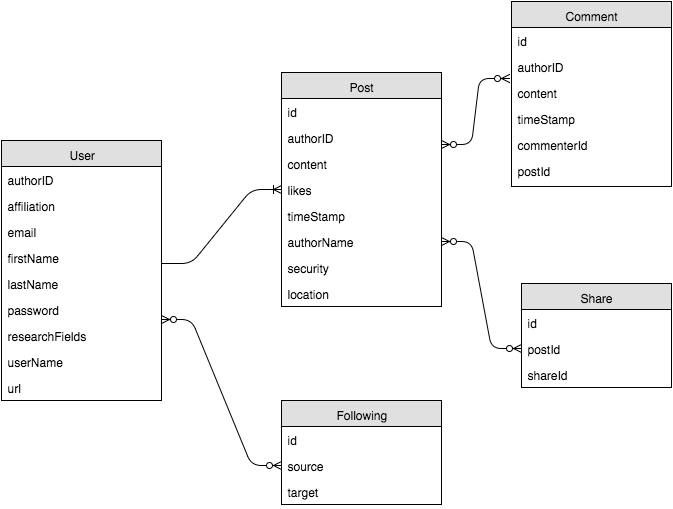
On the frontend side, we implemented the following list of functionalities:

* A user has a personal home page.
* A user can sign up as a new user or login in as a return user.
* Personalized main page has user profile and posts from themselves or people followed by them.
* User can edit their own personal data in the front end and save it in the backend.
* User can search post using a search bar at the top of the page.
* User can search user by using a search page with search options including last name, first name, and research fields.
* User can see all of his/her followers.
* Personal home page displays posts sorted by timestamp in descending order.
* User can post, delete, like and edit a post.
* User can set privacy of their post as public or private.
* User can comment on a post.
* User can follow/unfollow a researcher.
* User can share a post.
* User get autocomplete search when searching by post.
* User can only modify his/her own data and is prevented from modifying others’ data.

On the backend, we implemented the following list of APIs:

* API for posting message.
* API for retrieving posts with comments.
* API for updating a post.
* API for deleting a post.
* API for commenting on a post.
* API for retrieving followers.
* API for retrieving users being followed.
* API for following/unfollowing a user.
* API for setting security of a post.
* API for retrieving user’s location.

For system database implementation on the backend, we created additional tables to store entities User, Post, Following, Comment, and Share and their relationships, as illustrated in the following class diagram:



**Figure 2. Class diagram of User, Post, Following, Comment, and Share classes.**

5.2 **Innovative Features**

An addition to the basic requirements, we implemented several innovative features to make our social network more secure and comprehensive. In particular, we implemented a REST API with authentication, a “people you may follow” feature with Spark, and search with Lucene, which are described in detail below:

* **REST API With Authentication**
  + Every time a user is created, the system generates a secret token associated with the user.
  + Every time a user makes a REST call, the backend first authenticates the user with the token embedded within the request.
  + If the system cannot find the user associated with the token, the access is denied.
  + If the system finds the corresponding user, the user is authenticated and we proceed to controller methods.
* “**People You May Follow” with Spark**
  + We want to cluster users into different groups based on their research interests using K-means clustering algorithm.
  + For each user, we recommend potential users to follow.
  + We incorporate Spark Core and Mllib into our main app.
  + We run the K-means algorithm as a background thread every 5 minutes
* **Search with Lucene**
  + A lucene search library is applied when searching User and Post.
  + The lucene library provides both exact match and fuzzy match.
  + The lucene index db is synchronized with MySQL. When user perform any update (ex: add new user and post, modify personal profile and so on), the update is immediately available in Lucene. The user can always search the latest post or user profile.

1. **Experiments and analysis**

In this section, we will compare the performance of the different implementations on search. One experiment is implemented using MySQL, while the other is implemented using Lucene.

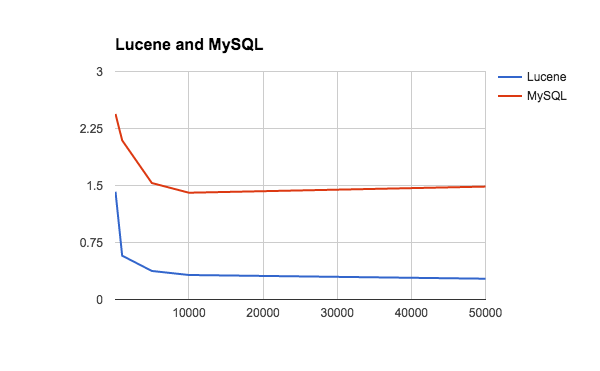
**Experiments**: function call level

Benchmark 5000 queries for search keyword “semantic” over 1000 posts,

Mysql: 15 ms/per query,

Lucene:2.9 ms/per query,

The result shows that the running time of Lucene is about 5 times faster compared to MySQL. We also tried different number of queries to see the difference.



**Figure 3. Lucene and MySQL average running time vs. number of queries.**

In the graph above, the x-axis is number of queries, and the y-axis is average running time. From the graph, we can see that Lucene always performs better than MySQL. Note that the initial average running time is slow because MySQL needs to open the connection and Lucene need to open the index database. However, as the connection and index database opening needs to be done only once, the initial cost will be averaged down when there are more queries. The cost per query will get close to zero as the number of queries gets very large.

**Experiments**: Restful Service/Throughput

We also tested the restful service level performance. Because the frontend will not access the backend via API functions, the frontend access the backend through restful service instead. Therefore, we also checked out the restful service level’s performance. We used wrk to test the throughput, and the library can be found in the following link: <https://github.com/wg/wrk>. The experimental results are shown below:

Lucene: 4 threads and 100 connections

Thread Stats Avg Stdev Max +/- Stdev

Latency **19.82ms** 21.25ms 530.85ms 94.50%

**Requests/sec: 5960.25**

Transfer/sec: 10.91MB

4 threads and 100 connections

Thread Stats Avg Stdev Max +/- Stdev

Latency 169.37ms 122.00ms 704.61ms 60.07%

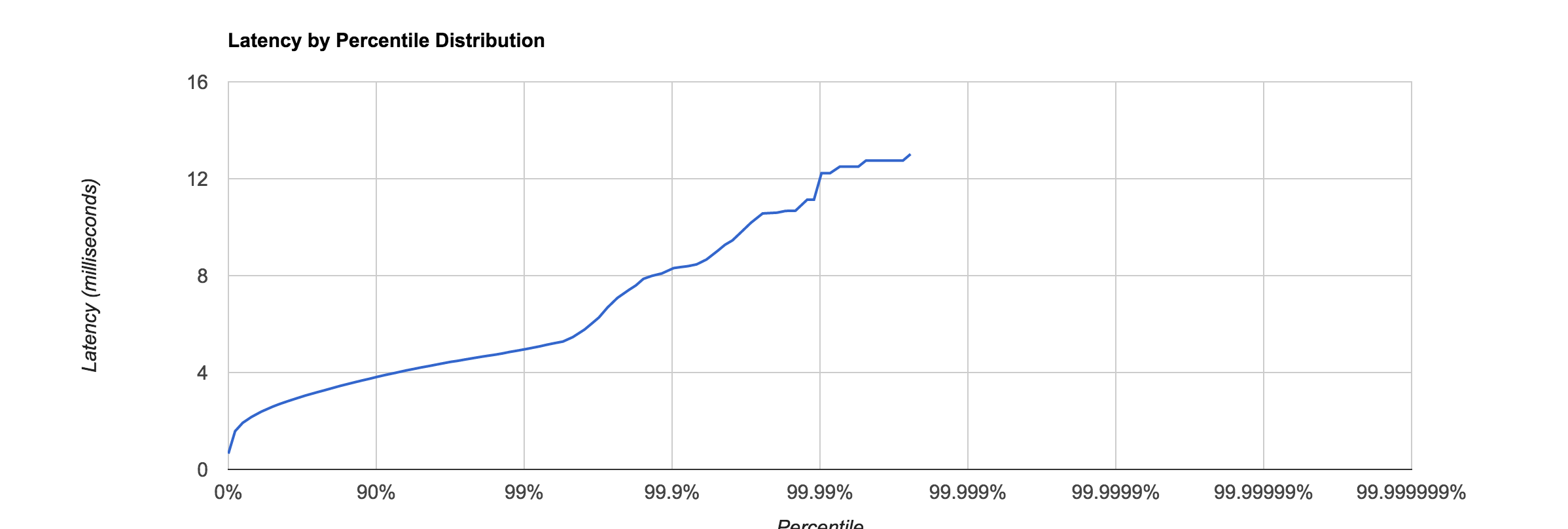
Requests/sec: 612.05

Transfer/sec: 3.66MB

The result shows that Lucene’s throughput is 10 times higher than MySQL, the average latency is 8 times shorter than MySQL.

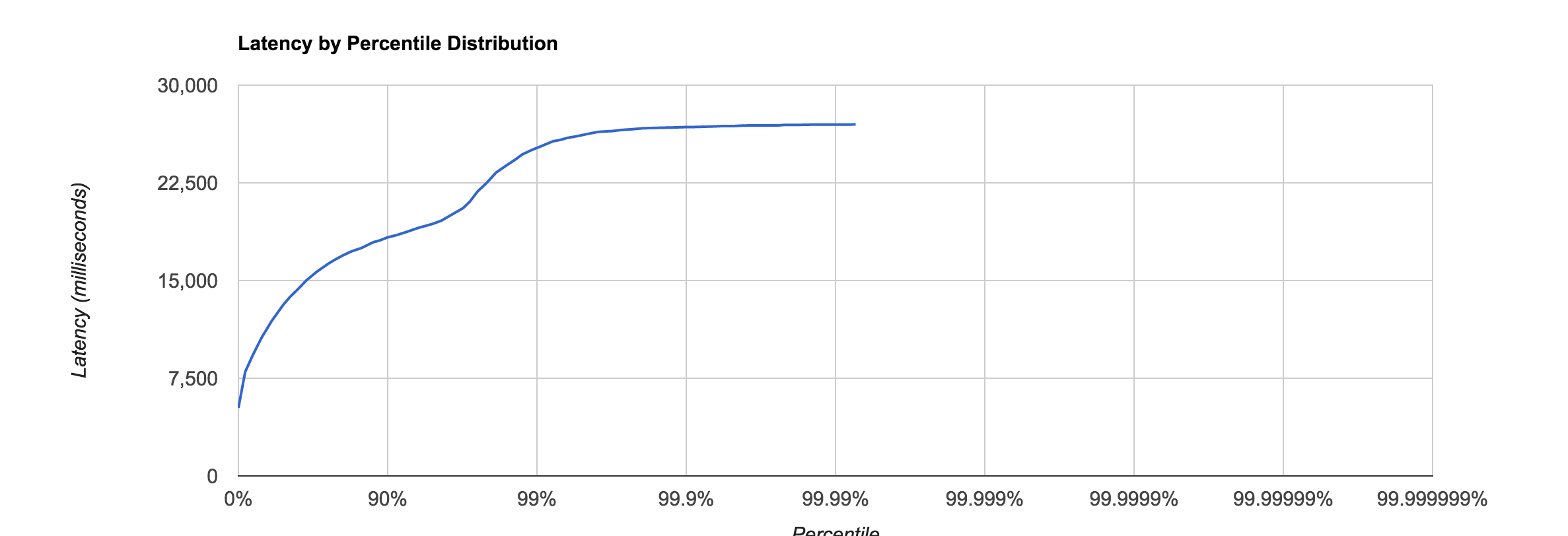
**Experiments**: Restful Service/response time

Here we use <https://github.com/giltene/wrk2> to test the performance.



**FIgure 4. Lucene latency vs. percentile.**

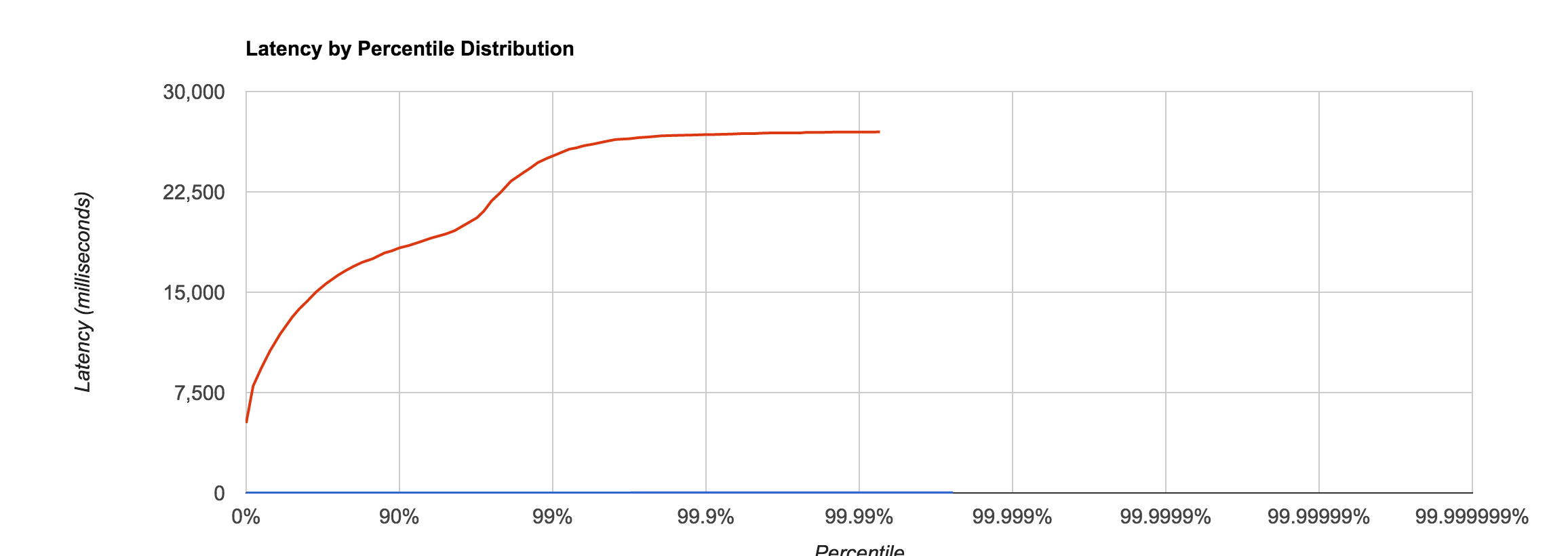
In the figure shown above, the x-axis shows percentile and the y-axis shows latency. From the graph, it shows that 90% of the request will be handled in 4 millisecond, but in 99% percentile, the average latency is 7 seconds. Basically, the graph explains that the majority of requests can be handled in 4 milliseconds. When the percentile grows larger, we can see that the latency becomes larger. For example, in 99.9% percentile, the average latency is 8 milliseconds, and 0.1% of request will get more delay. The phenomenon that causes this is that the JVM needs time for garbage collection, and when that happens, the JVM will perform “stop the world” to stop all the threads, which is why there will be long latency for some requests.



**Figure 5. MySQL latency by percentile distribution.**

The graph above shows that the latency grows even faster, from which we can deduce that the MySQL implementation in our system also uses JPA for data abstraction. The JPA will use reflection to convert the data in MySQL into instances, so it will become even slower. Besides, MySQL does not support full-text search, which is why Lucene’s optimized index will have much better performance.

Experiment: Response time MySQL vs Lucene



**Figure 6. Latency of MySQL vs. Lucene.**

When we compare the performance of lucene (blue line) and MySQL (red line), the graph clearly show that Lucene is better in not only request per second but also response time.

1. **Conclusions and future work**

There are several improvements that can be made for future work. For stronger security protection, we can add an authorization layer on top of REST authentication. To further improve performance, we would also like to reduce the number of round-trips to DB by creating additional classes to serialize complicated objects. Additionally, we can load data on-demand.

Our good wish for the project is for high availability and distributed cache as the following:

* + High Availability: introducing redundancies to remove single point of failure. Example: load balancing between machines.
  + Distributed Cache: independent cache server to improve DB performance with concurrent read and concurrent write.

1. **Contribution of each team member**

All our team members co-work with each other through the entire project, so it’s hard to differentiate each other. If there is anything markable, what we have done are as follows:

**Bowen Zhang**

* Spark recommendation for “People You May Follow”
* Security token authentication for backend API
* Post publish/delete/edit functionality
* Post comment functionality
* Post sharing functionality
* Post like functionality
* Initial Signup/Login process
* Initial home page UI

**Chia-Chuan Wu**

* Lucene for user/post search
* Autocomplete for search keyword
* Privacy control for post and profile
* Profile editing
* Popular post ranking

**Ning Du**

* Share location combine Google API and our system
* Following/unfollowing user from both frontend and backend
* Main page post retrieving API
* Home page post retrieving API

1. **Tutorial**

Please see the tutorial document under /document/tutorial.docx

**Appendix:**

-Check in everything onto GitHub under the predefined directory including the following items

-Readme file: Describe briefly the purpose of the project, how to download and install the software, how to use the software

-API (sub-directory): instruct APIs as well as descriptions and examples

-Test Suite (sub-directory): a collection of test examples and descriptions

-src (sub-directory): include all source code categorized by packages

-lib (sub-directory): include all related library packages needed to support the project

-conf (sub-directory): include any configuration settings and files

-app (sub-directory): any applications built on top of the APIs

-contact: please provide every team member’s contact information (cell number, personal email)

-Documents (sub-directory): in different WORD files

-access information: URL, user name/password

-download and installation documents with step-wise descriptions

-executive summary

-background and motivation

-assumptions and considerations

-design documents (architectural design documents and various diagrams e.g., UML files)

-presentations (ppt file)

-tutorial: step-by-step usage file with screenshots included

-future work: to-do list and descriptions

-technical report

-Transit the knowledge to either Advisor or a signed student (schedule time to sit down for transition)