Just Search It -- Search Engine Final Report

Search Engine Lab
Anmol Kaur (997378029) and Vincent Lee (997454419)
Group 10

Revisions - Lab 1: October 13, 2013
Revisions - Lab 2: October 27, 2013
Revisions - Lab 3: November 10, 2013
Revisions - Lab 4: November 24, 2013
Final Lab report: December 4, 2013

Table of Contents:

Final Report:

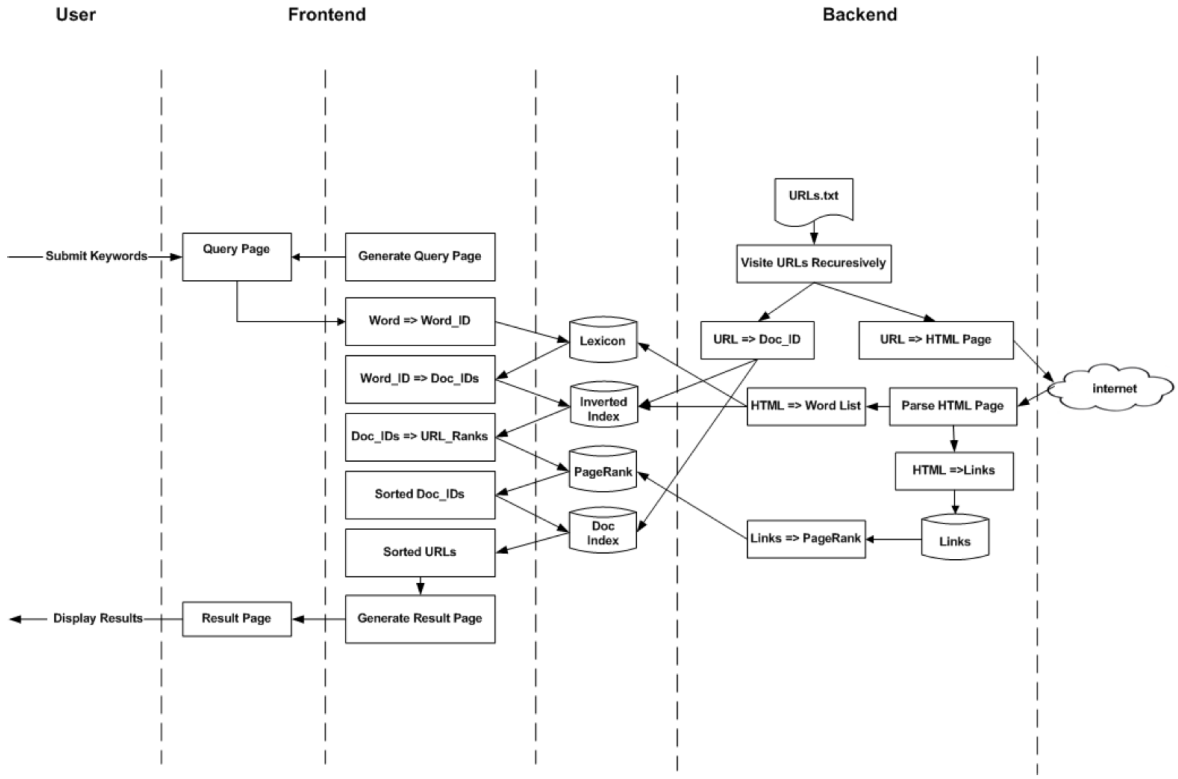
Overview:

The Just Search it search engine is a light weighted search engine that aims to have a simple interface that allows users to search the web but inputting a keyword.

The Just Search it search engine consist of a backend and a frontend. The backend crawls the internet recursively and populate website information to allow the frontend to serve end users useful search results.

Architecture

Figure 1 below shows a s system block diagram of the search engine.



Description

BackEnd

The backend crawls the URL provided by the URLs.txt recursively. It stores then populates the Lexicon, Inverted Index and Doc Index tables in the database.

After crawling all the urls provided, a PageRank algorithm will be applied to the data and calculate ranking of each web page. Search results are sorted based on this ranking to determine it's importance. The results will be stored in the PageRank table of the database

FrontEnd

The frontend takes in a keyword from the form and query the database for optimal search results. It then paginate the results and display it to the end users.

Features

Here is a list of features the just search it engine provide

- Search with keyword
- Open results in new tab
- Search results pagination
- Google Login
- Popular Search Query Caching (Backend)

Screenshots:

Search with Keyword

Figure 2 shows the landing page of the search engine

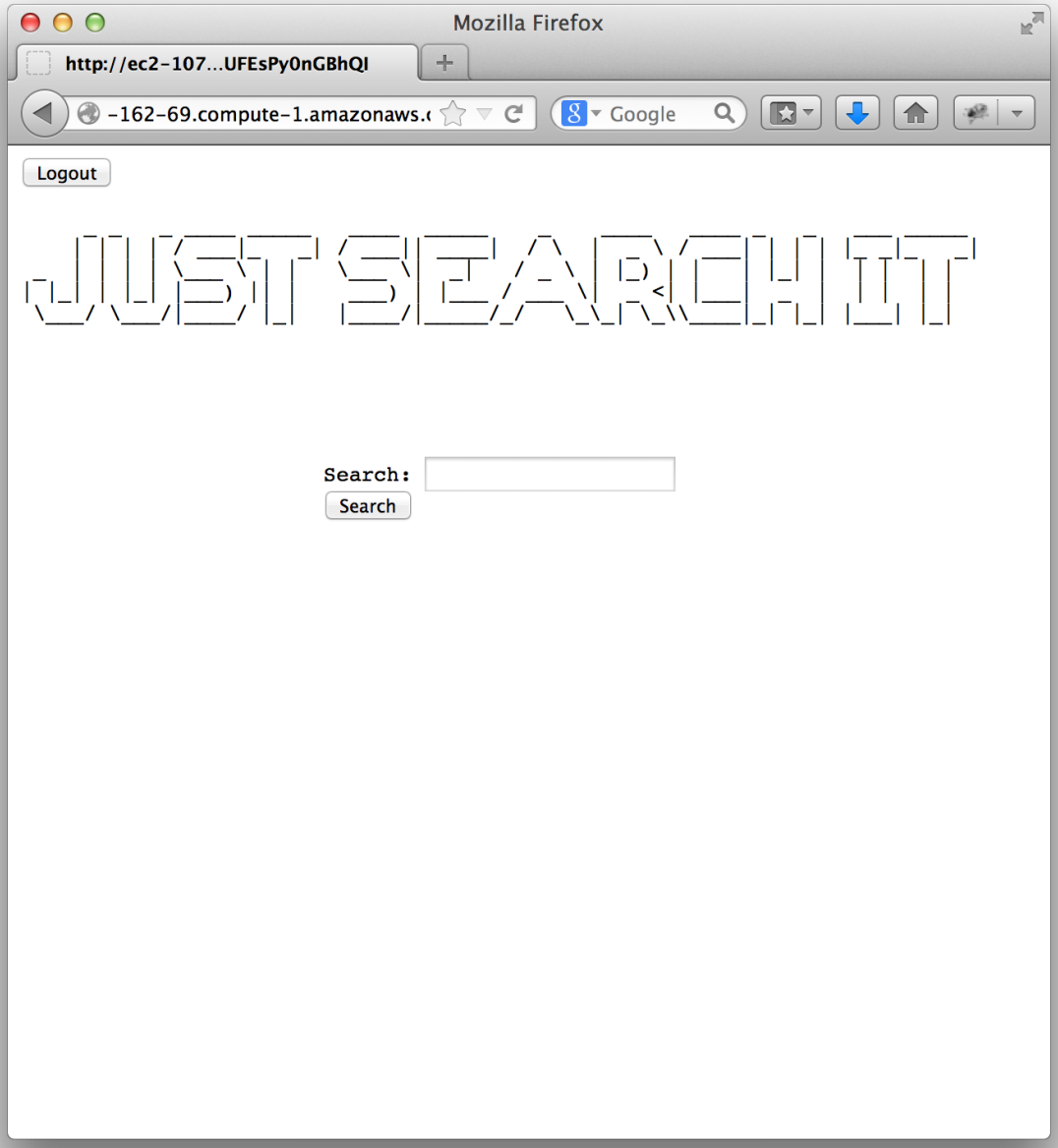
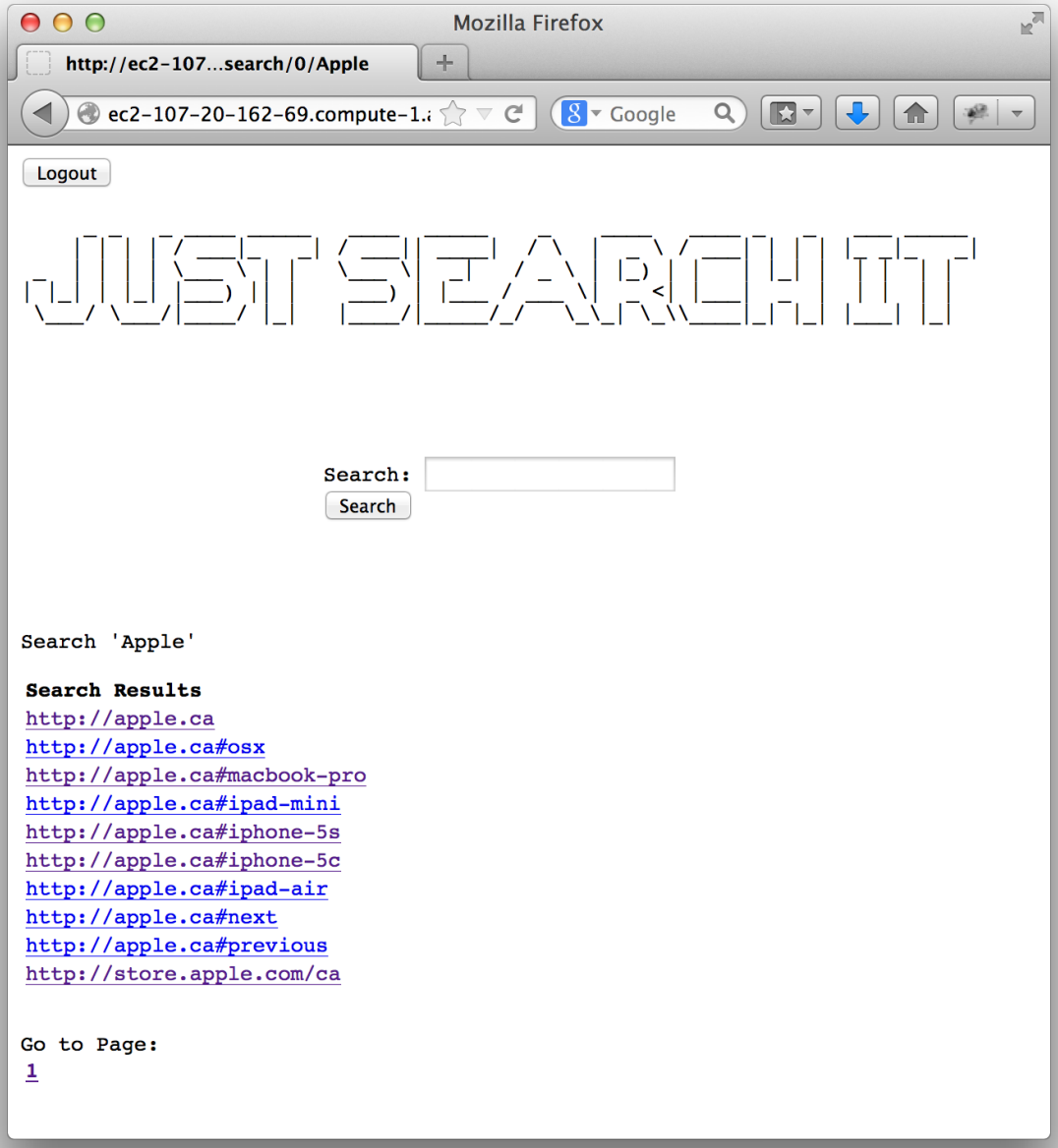
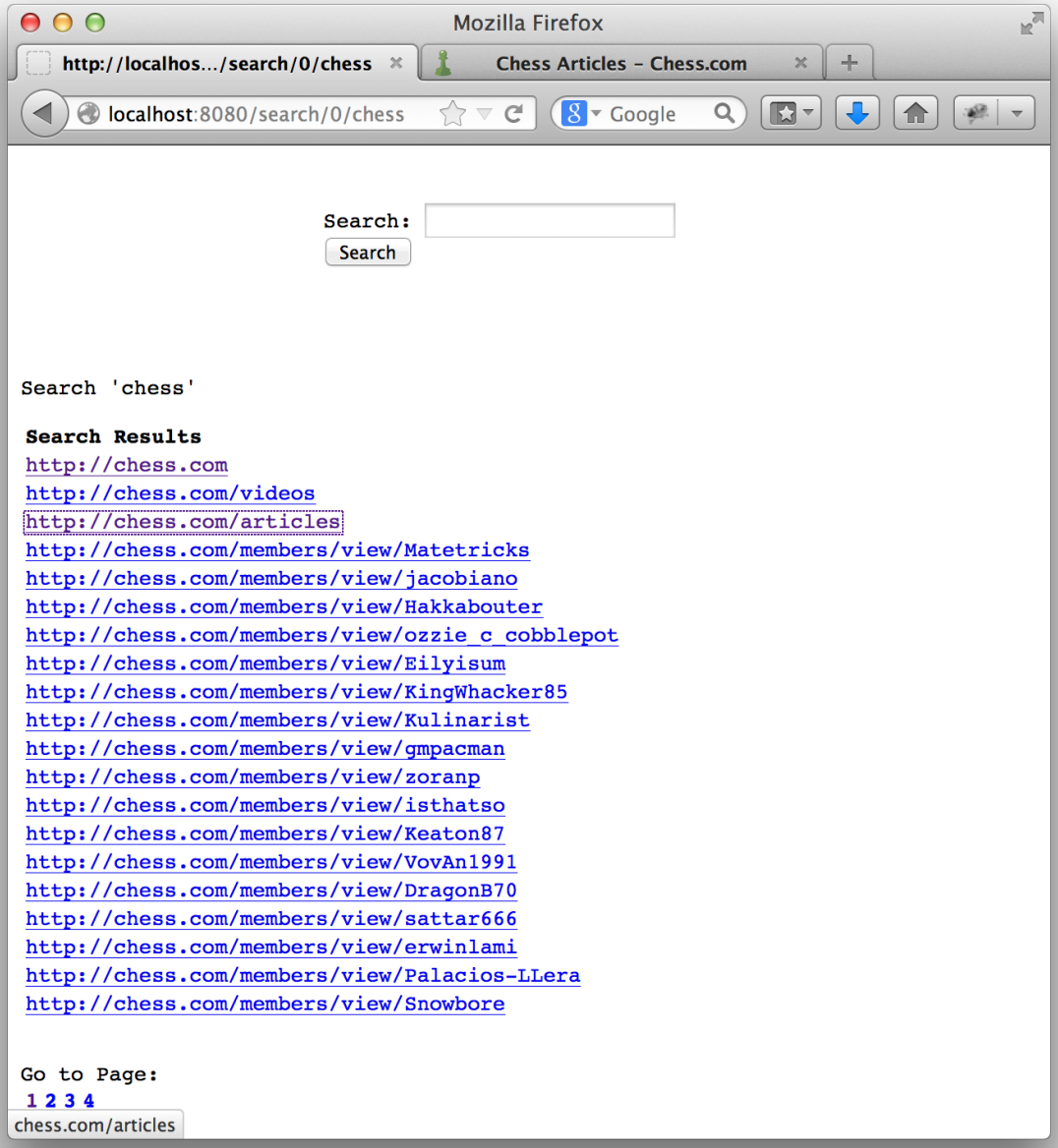


Figure 3 shows the search results for the keyword “Apple”



Results in new tab

Figure 4 show the a Firefox window with search results opened in new tab. Keeping the search results in another tab.



Pagination

Figure 5 show the first page of the search results for the keyword “chess”

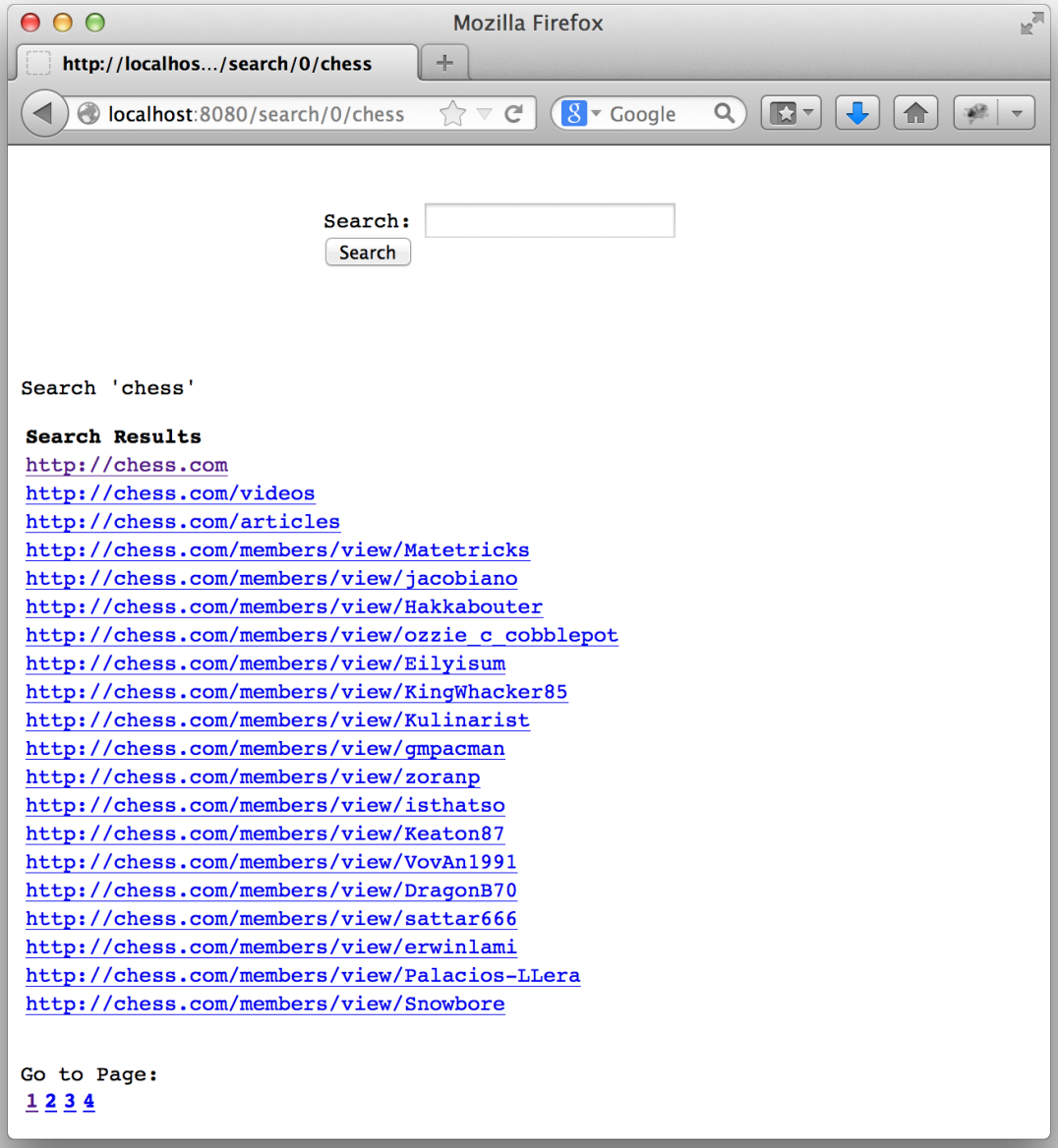
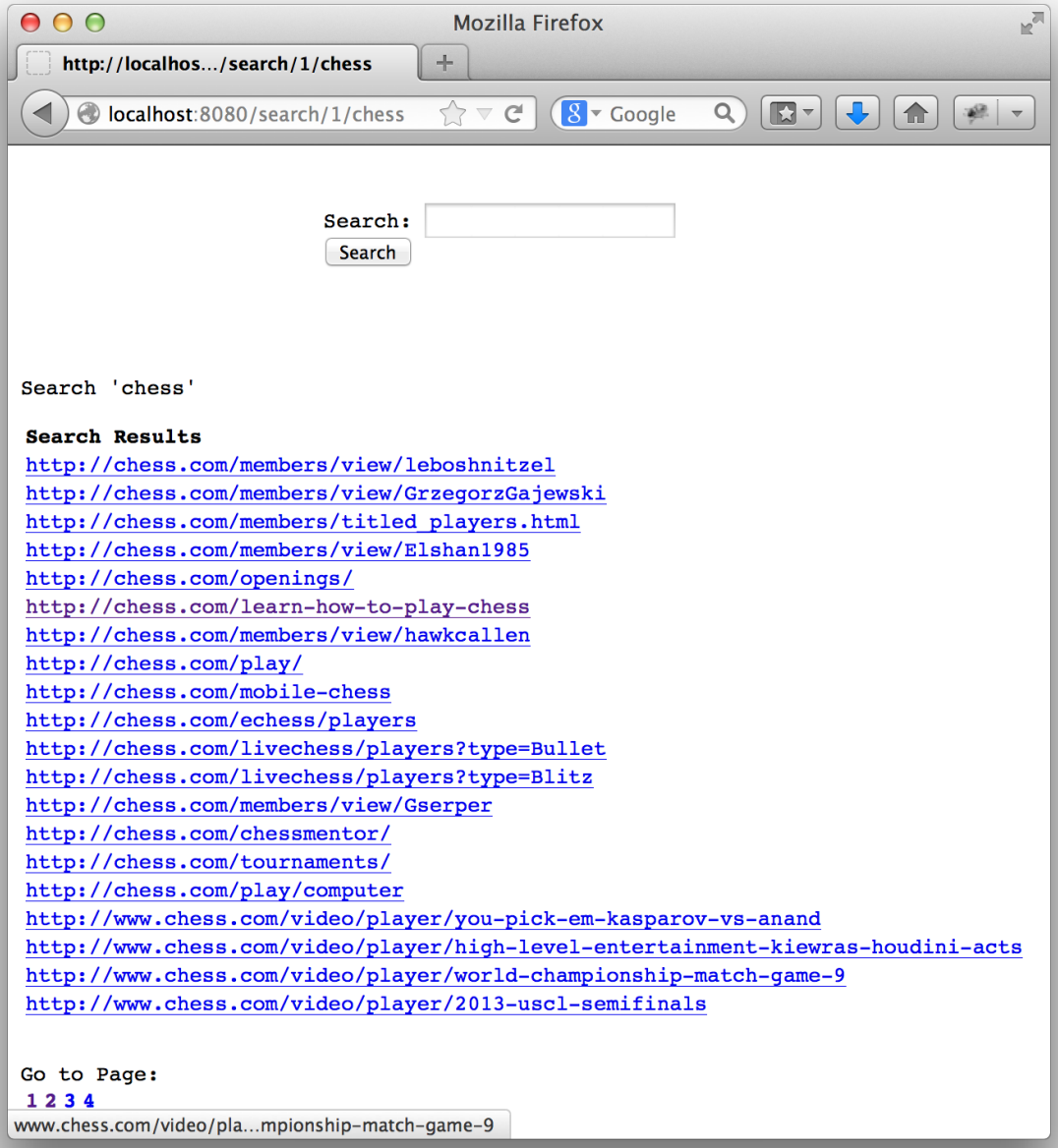


Figure 6 show the second page of the search results for the keyword “chess”



Performance

Latency and throughput of the search engine is discussed in great details in Appendix: Lab3 Progress report. Please refer to that section for more details

Possible optimization could be running multiple instances of the server in different parts of the world. Running multiple processes of the search engine with a load balancer to spread out the load.

Methodology for Benchmarking and Optimization

In Lab 3 we used UNIX commands ab, vmstat and mpstat to benchmark the application. In Lab 4 we are given the chance to optimize the search engine. In Lab 4 we have used Chrome Network Develop tool to evaluate the bottleneck of the search engine load up time. We discovered that database query was the major bottleneck and implemented caching for queries.

Description of Design Tree

Provide a brief description for each file or package included in your code submission.

FrontEnd.py

This file provides the functions used for generating the frontend HTML serving to end users.

crawler.py

This file is the backend of the search engine. It crawls the urls provided in the urls.txt and populates the database

install.sh

This script configures a brand new ubuntu installation to run a search engine on Amazon AWS

installPackage.tar.gz

This tar ball contains all the files required to run an AWS instance.

Logo.txt

This file holds the plaintext of the JUST SEARCH IT ASCII art

urls.txt

This file contains all the urls that the backend will crawl.

README.md

Contains all the instructions for an admin to launch a new AWS instance.

DNS_NAME

This file holds the domain name for the search instance on the AWS

bottle.py

This is the file for the Bottle framework for all the magic to happen.

Contribution

Anmol was responsible for the frontend of the search engine and Vincent was responsible for

the backend of the search engine. It worked out really well as each lab has roughly the same amount of work in both backend and frontend. We helped each other out in terms of Database handling and frontend routing. We both had the chance of being exposed to areas that we've never experienced, and have learned a lot through this project.

Reflection

Anmol

Through completing this project, I've learned how to use the Bottle framework, which is really interesting. It is my first time developing a website and using Python. I alas had the opportunity to apply my knowledge from the Database course to this project. I find this project immensely valuable for me personal development and makes me a better programmer.

Vincent

Through completing this project, I have learned how to use Python, and the SQL Query language. And database operations. The most challenging part of the lab was to set up an AWS instance of the search engine.

Prior to taking this course, I worked as a software developer at my PEY placement. At that the placement, I had the chance of working on a Ruby on Rails web application. This gave me the opportunity to understand HTTP REST API and webpage routing. I find this experience invaluable to give me intuitions about how the Bottle Framework works and how to get the project finished in general.

Course Feedback:

The distribution of course load is extremely uneven and should be adjusted accordingly for the upcoming year.

Lab 1:

Search Engine Overview

Just Search It is a search engine that will use a web crawler to create a query page of relevant websites based on a user search.

In this lab, we created a dictionary data structure to store Document Index (ordered by document ID), Lexicon and forward Index. We modified the crawler to include two new functions - one to return the inverted index and another which returns resolved data in string and URL strings.

For the front end, we created a web browser with our logo and a search box. In this lab, a search result returns "Search *'User's search text'*" and a list of all the words the user searched and the number of times they occurred in that search.

Front End Design Decisions

The group made 2 key design decisions for the Front End - creating a plain text logo and using an HTML table to format the word occurrence text.

The first decision was to decide whether to use an image or plain text for our logo. We considered the trade-offs; an image would be easier to format and display, however would take up more bandwidth. On the other hand, using plain text would take some formatting but would give the logo a hacker-like feel. We found the plain text logo was more reflective of the Just Search It engine and since it also used lower bandwidth, we decided to go with the plain text logo.

The second decision was to decide how to display the word occurrences such that it presented a clean, aligned and consistent look. After some research on different display spacing options, we decided to use an HTML table so the text would remain aligned regardless of the lengths of the words inputted.

Team Contributions

The team has split up the work into frontend part and backend part. Anmol was responsible for the front end implementation and Vincent was responsible for the backend implementation.

Instructions to run code

To execute the crawler, use the following command:

```
%>./crawler.py
```

This commands runs the crawler script and crawls Professor Stark Draper's website.

To execute the front end, use the following command:

```
python Frontend.py
```

This will connect you to the server.
Next, on your browser, go to: "localhost:8080/search". This will open a browser page with a search box.

Bonus

The front end bonus was not completed for this lab.
The back end bonus was not completed for this lab.

Lab 2:

Lab 2 Overview

Just Search It is a search engine that will use a web crawler to create a query page of relevant websites based on a user search.

In this lab, we created a database which contains 5 tables. See schema below:

```
DocIndex(doc_id INTEGER , url TEXT, PRIMARY KEY(url, doc_id));
InvertedIndex(word_id INTEGER, doc_id INTEGER, PRIMARY KEY(word_id, doc_id));
Lexicon(word_id INTEGER PRIMARY KEY,  word TEXT UNIQUE);
Links(from_url INTEGER, to_url INTEGER);
PageRank(doc_id INTEGER PRIMARY KEY, rank INTEGER);
```

The tables of the database will be updated every time the crawler is ran. Tables populated by the crawler will be queried by the front end every time the user uses the search engine. After the crawler has finished crawling the web, I will calculate the pagerank of each site.

For the front end, we query the user's input against the database of words from the crawler and return any relevant links in order of relevance. If over 20 search results are returned, they are split across multiple pages to create a more user friendly experience.

Front End Design Decisions

The group made a few key design decisions for the Front End to create a user friendly web page. One decision was how many search result to display per page. Using trial and error, it was determined that 20 was a comfortable fit per page. It allowed users enough sites on one page such that they did not have to flip through multiple pages to find a search they were looking for and also so that the page did not offer a scroll that uncomfortably long.

Another key decision that was made for the front end was whether to use a pagination framework such as Flask to split the search results over multiple pages or to write custom code to create the pagination effect. After careful consideration of both options, we found that the framework may not integrate well with our current bottle framework and was also inconsistent with the the HTML code we had already set up from Lab 1. To keep the pages consistent and well integrated, we decided to write our own code.

Note - we removed the section returning the occurrence of each word searched as we assumed that was not meant to be continued to lab 2.

Back End Design Decisions

We have decided to use SQLite3 as our database as it is recommended by the instructor.
Took out the functions implemented in Lab 1 (get_resolved_inverted_index() and get_inverted_index()).

Team Contributions

The team has split up the work into frontend part and backend part. Anmol was responsible for the front end implementation and Vincent was responsible for the backend implementation.

Instructions to run code

To execute the crawler, use the following command:

```
%>./crawler.py
```

This commands runs the crawler script and crawls multiple websites listed in urls.txt and calculates the PageRank of pages. Submission contains a database populated by the crawler, running the crawler before starting the front end is not required.

To execute the front end, use the following command:

```
%>python Frontend.py
```

This will connect you to the server.
Next, on your browser, go to: “localhost:8080/search”. This will open a browser page with a search box. Based on our root sites, some good words to search are:

- apple
- draper
- toronto

Bonus

The front end bonus was not completed for this lab.
The back end bonus was not completed for this lab.

Lab 3:

Lab 3 Overview

Just Search It is a search engine that will use a web crawler to create a query page of relevant websites based on a user search.

In this lab, we have deployed the search engine on to the Amazon Elastic Computing Cloud. The Search engine will be accessible through the following url for two weeks: http://ec2-107-20-162-69.compute-1.amazonaws.com/search

An installation script is also developed so that the search engine can be deployed onto multiple server instances on the cloud through scripts.

For the front end, we added a Google Sign-in page asking users to authenticate themselves through their Google accounts before accessing the search engine. Also, a logout button was implemented. In this lab, we've set up a session management such that if a user is logged in on one tab, they do not have to re-login on another tab. The session is terminated either when the user logs out or the browser is closed. The cache was also accounted for and cleared such that if a user logs out, they cannot use the browser back page button to return back to their session.

Benchmarking methodology

The performance of our web server was analyzed using the Apache benchmarking tool. Various tests were conducted to better understand the servers performance and the results were recorded and compared for the different cases.

Tests include:

- CPU usage with increasing connections
- CPU usage with increase number of requests on one connection
- Memory usage with increasing connections
- Memory usage with increase number of requests on one connection
- Max number of connections that can be handled by the server
- Max number of requests given one connection
- Max number of requests given max number of connections

Benchmarking analysis and results

Testing max number of requests:

Requests Sent	Connections	Requests Processed
1000	50	994
1000	100	971
1000	200	812
1000	500	768
1000	1	1000
2000	1	2000

From this trend, we can see that as more connections are added, not all requests are processed. However, with just one connection, the number of requests that can be processed is upwards of 2000.

Testing max number of connections:

Requests Sent	Connections	Connections connected
400	400	390
300	300	295
250	250	218
200	200	200

Need to interpolate between 200 and 250 connections:

Requests Sent	Connections	Connections connected
230	230	230
240	240	206
237	237	237

The largest number of concurrent connections that run without any dropping is 237.

Memory and CPU usage:

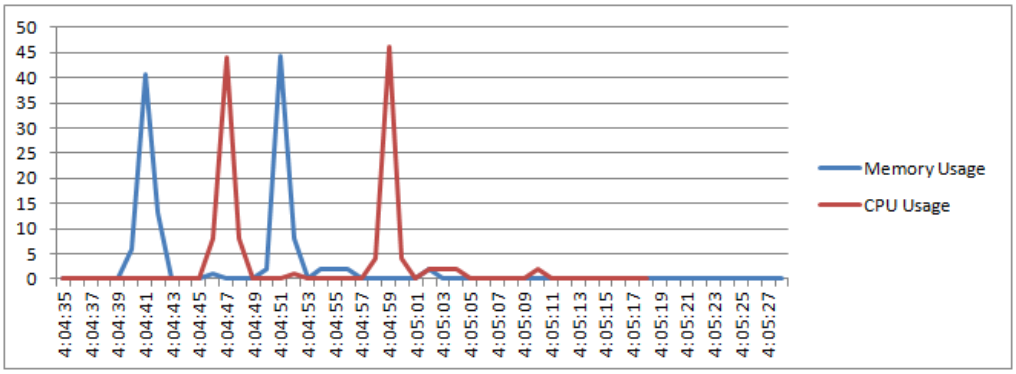


Figure 1 shows the CPU utilization and memory usage of the EC2 instance over a small burst of requests

Team contributions

The team has split up the work into frontend part and backend part. Anmol was responsible for the front end implementation and Vincent was responsible for the backend implementation. The benchmarking was done together.

Instructions to run code

To use the installation package:

Untar the tar ball using the following command:

```
%>tar -tzf installPackage.tar.gz
```

Run the installation script “install.sh” with root permissions. It will download the source code required from Github and install required libraries for running the program.

Lab 4:

Lab 4 Overview

In this lab, we identified potential bottlenecks for the Just Search It search engine. 3 key metrics were identified to collect baseline data on, potential bottlenecks were identified, and one bottleneck was addressed and optimized for.

1. Metrics

To collect baseline data, the two metrics identified were:

- Search speed - how long does the engine take to return results
- Database query time
- Crawler Time - how long does the crawler take to crawl the web and update database

Search speed

Search speed can be used to evaluate the search engine's front end. To collect baseline data, we used Google Chrome's Network Console to record search and load times of several search instances.

Figure 1 below shows search and load speed for the search "apple" as 15 milliseconds. Figure 2 shows another search at 14 milliseconds.

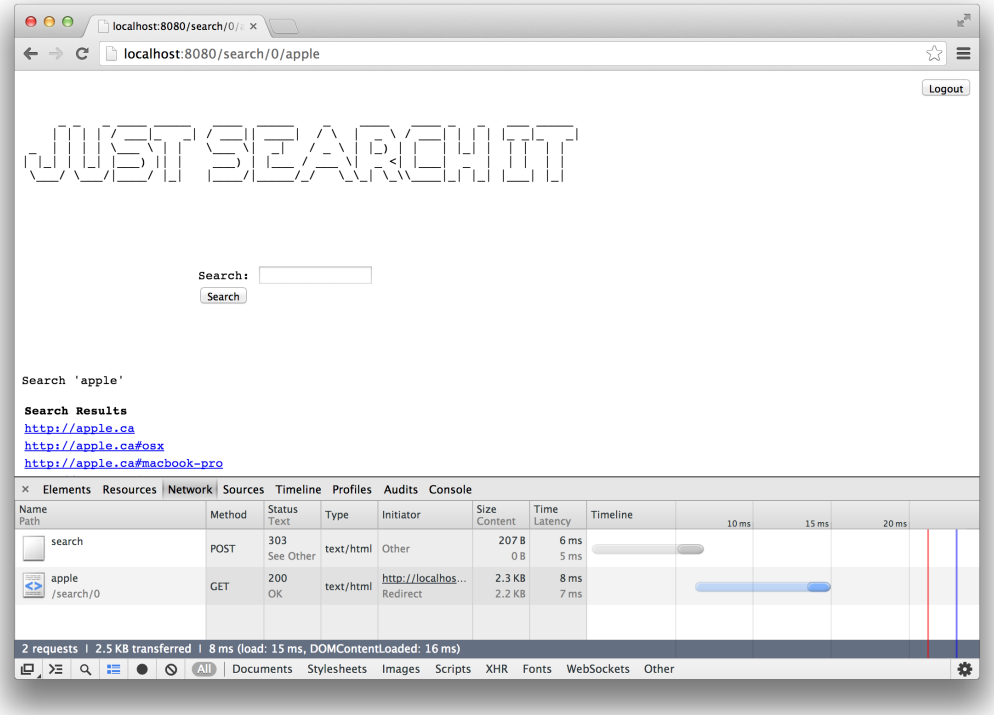


Figure 1: Baseline search and load times for “apple”

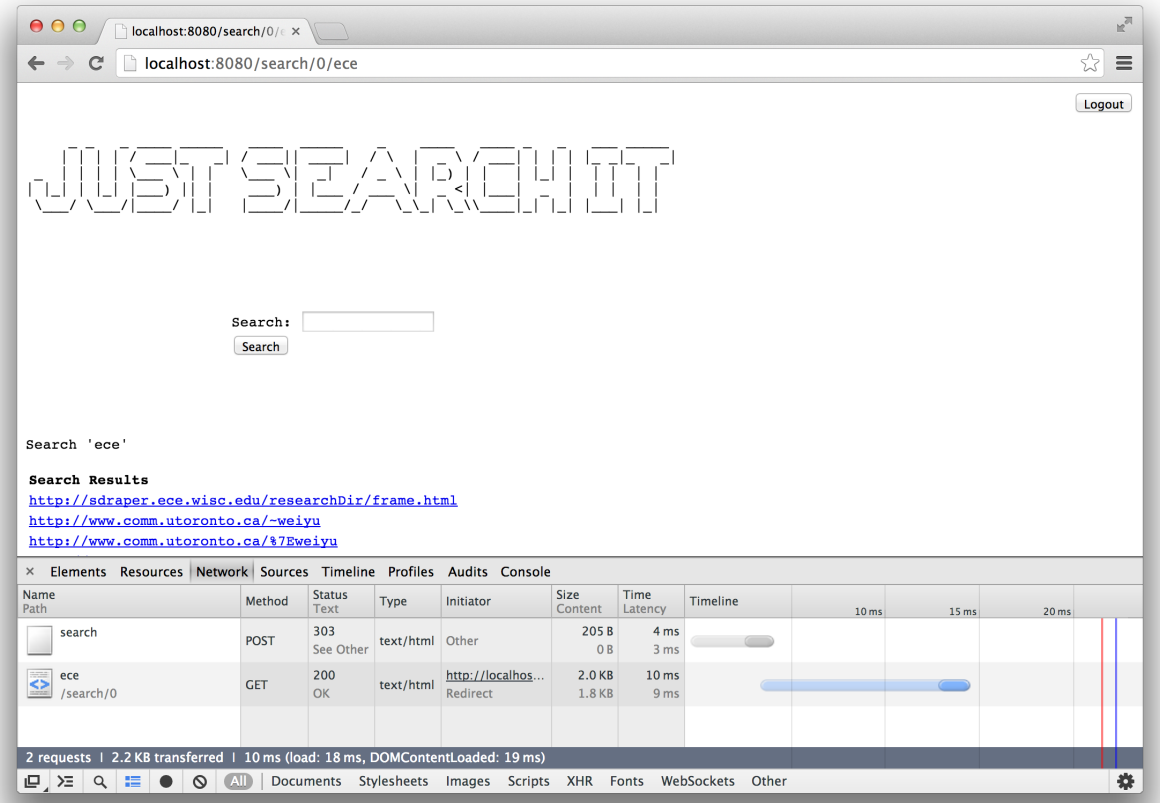


Figure 2: Baseline search and load times for “ece”

Database Query Time

The search speed can further be broken down to different components including the database query time. While we identify this as a potential area for evaluate performance, baseline data for this metric was not collected in this lab.

Crawler Time

One metric to evaluate the performance of the backend is the time the crawler takes to crawl the web. To collect baseline data, the crawler was tested with different numbers of root websites. The crawler time is affected noticeably as the number of links crawled increases. This could be caused by slow server responses crawled by the crawler. Table 1 shows the results of these tests.

Table 1: Crawler run time

Number of links crawled	Crawler run time	Time spent/ link
864	26 seconds	30ms
2217	46 seconds	20ms
14774	2 minutes, 17 seconds	9ms
15751	2 minutes, 51 seconds	10ms

2. Identifying Bottlenecks

Once some key metrics were baselined, we identified potential bottlenecks of the application. Some bottlenecks include:

- Database query time
- Response time of servers being crawled

3. Optimization

Based on the potential bottleneck and baseline metrics, we chose to focus on optimizing search speed in this lab. To optimize search speed, a cache was implemented in the front end code, such that if a user re-searches a particular word, the results from the previous search are returned and the search is not conducted again.

The cache implementation noticeably improved the search times. Once the data was cached, we re-searched the words from the baseline data, “apple” and “ece”. The results are shown in Table 2 and Figures 3 and 4 below.

Table 2: Search times before and after optimization

Search word	Before Optimization	After Optimization	Percent Decrease
apple	15 milliseconds	8 milliseconds	46.67%
ece	14 milliseconds	4 milliseconds	71.42%

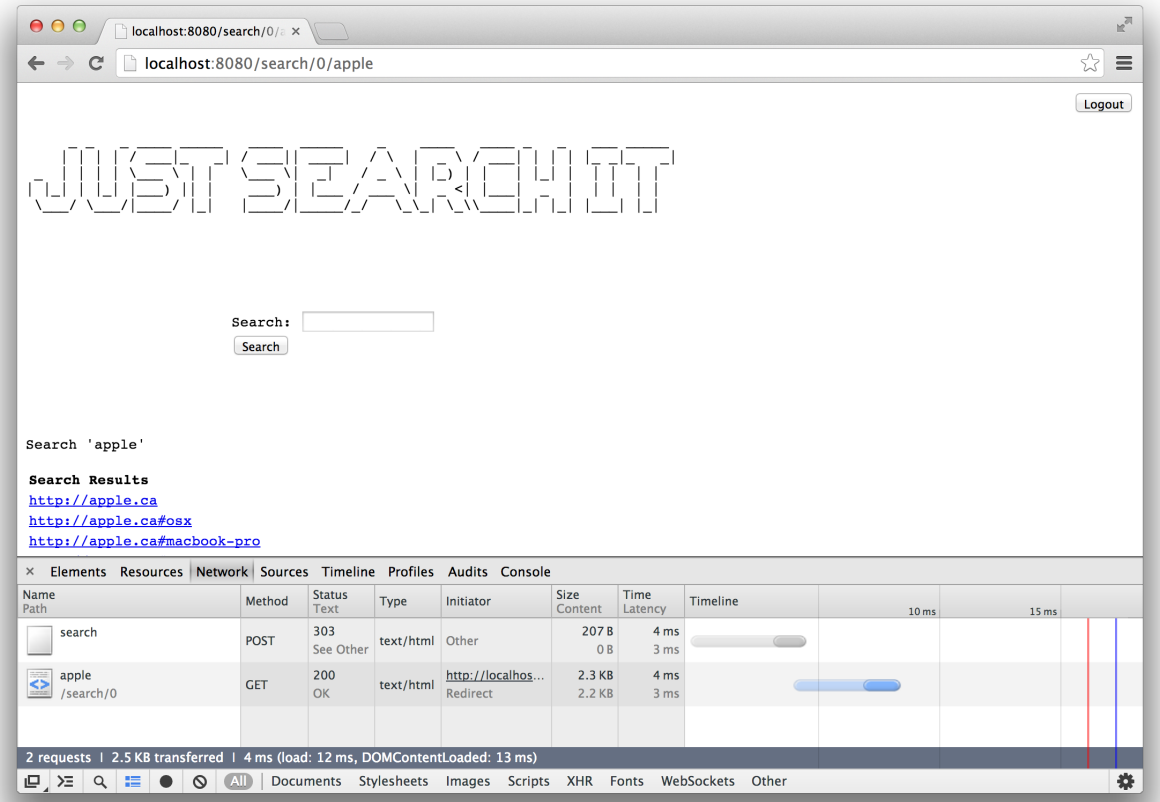


Figure 3: Search and load time for “apple” after optimization

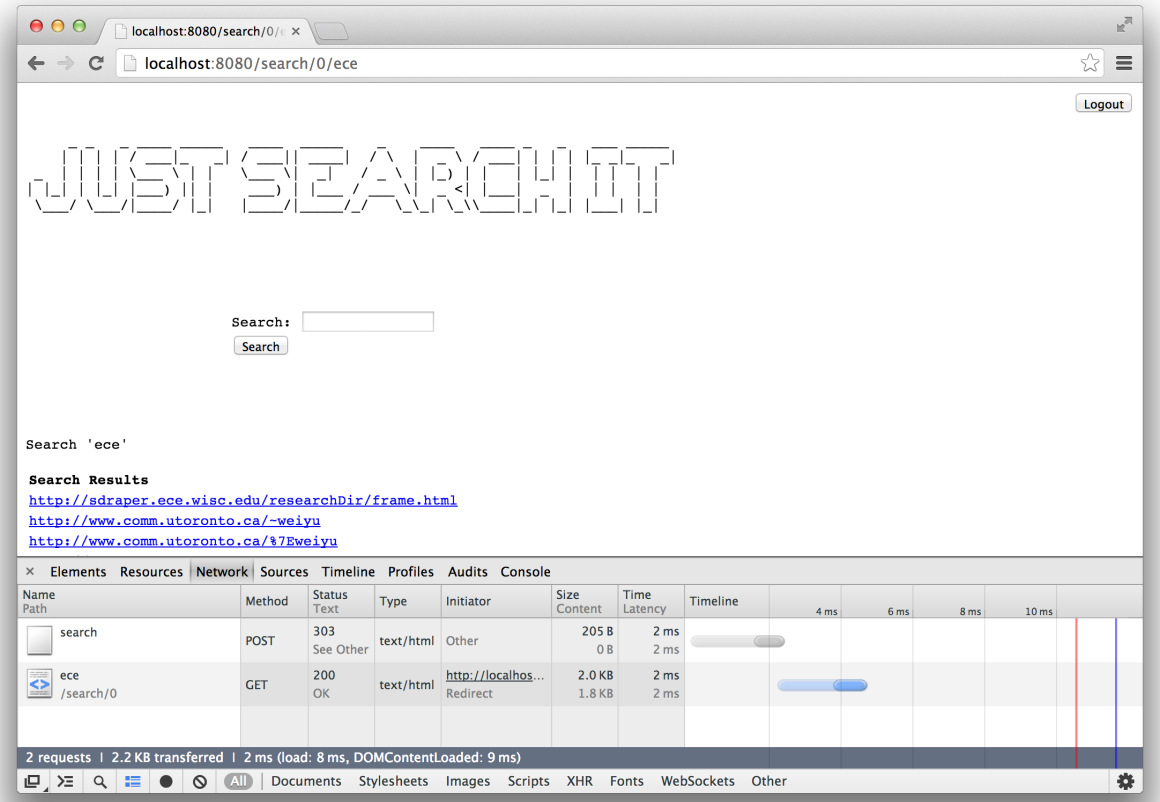


Figure 4: Search and load time for “ece” after optimization

Team contributions

The team worked together on this lab. Both Anmol and Vincent contributed equally to identifying metrics, collecting data, and optimizing the front end code.

Instructions to run code

To use the installation package:

Untar the tar ball using the following command:

%>tar -tzf installPackage.tar.gz

Run the installation script “install.sh” with root permissions. It will download the source code required from Github and install required libraries for running the program.

Appendix

Additional screenshots of your application.

Graphs for benchmarking

Full record of your progress report from Lab 1 to Lab 4

Your report should be self contained that readers without previous knowledge about the project

should be able to understand your achievement in this work.

you should include a README file that provides instructions for setting up the database service.