**CSC326**

**Final Lab Report**

Group #: 4

Names: Anny Kim Ly (997581718)

Kevin Gumba (997585117)

**Table of Contents**

[1.0 Overview](#h.8w3gisvfa92r)

[2.0 Architecture](#h.z3mvnu4ze58f)

[3.0 Features](#h.eibjqvk79kee)

[4.0 Performance](#h.vljy0ukh639a)

[5.0 Methodology for Benchmarking and Optimization](#h.o2cyfuqq8hvp)

[6.0 Description of Design Tree](#h.en1slox0rj76)

[7.0 Contribution](#h.46e30icdgngr)

[8.0 Reflection (per individual)](#h.dzucrgmoo5qx)

[9.0 Course Feedback](#h.t9wt215otmxh)

# 1.0 Overview

Space Explorer is a search engine composed of a front and backend. The search engine allows users can log into the search engine, enter a search keyword, and retrieve URLs with the related keyword.

When users enters the search engine, they must log in to their with Google+ account. The user inputs a keyword and the returned search results are displayed according to rank, with pagination occurring for large number of results.

with HAproxy

ec2-54-204-43-178.compute-1.amazonaws.com

without HAproxy

<http://ec2-54-204-43-174.compute-1.amazonaws.com/>

[http://ec2-54-225-76-58.compute-1.amazonaws.com](http://ec2-54-225-76-58.compute-1.amazonaws.com/)

# 2.0 Architecture

A description of the architecture of the Search Engine Application is composed of a Front End, a Back End, and has data transactions with Disk and Cache.

The Front End uses the Bottle Python Web Framework and is composed of 2 main web pages: the homepage, and the results page. There are also various error pages such as error 404 Page Not Found. The templates for these web pages are stored on disk and upon startup are loaded into cache. Session data (Keywords and results from the current session) is also stored in cache for performance reasons. Google+ API is used to implement Google+ login. The Front End must access data from various database tables from disk to retrieve the results pertaining to the searched keyword.

The backend is composed of a web crawler (crawler.py) that traverse by depth from a list of given homepages (urls.txt). During the crawl, descriptions of each traversed web page is parsed and necessary data is stored. These data include:

* lexicon: word ID and corresponding word
* document index: URL ID and corresponding URL
* inverted index: word ID and corresponding URL IDs
* url title: URL ID and title
* url description: URL ID and title

After all data has been retrieved from the crawl, persistent storage is set up. SQLite3 is used. The data from the crawl is stored into a database (dbFiles.db), allowing for info to be retrieved by the front end. Since the storage is persistent, old information is not overwritten unless new information is retrieved to update information.

All front and backend code is found within an Amazon Web Server (AWS) EC2 instance. Instances allows for the public to access the server remotely. The web server contains page balancing, where users enter a webpage and are ported to one of the copies of the front end instances.

Fig 1. Block Diagram of Search Engine Application

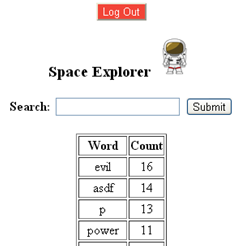
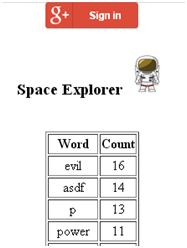
The crawler creates a persistent storage of data for the front end to retrieve. The crawler uses BeautifulSoup API to help with parsing HTML information in order to retrieve data found in tags such as paragraphs (<p>) and titles (<title>). The following is a table describing a few main aspects of the backend.

|  |  |
| --- | --- |
| **Backend** |  |
|  | **Description** |
| lexicon | * dictionary/table * contains word ID and corresponding word * one dictionary data structure   + key: word IDs   + values: corresponding words |
| document index | * dictionary/table * contains document ID and corresponding URL * one dictionary data structure   + key: document IDs   + values: corresponding URL |
| inverted index | * dictionary/table * contains word ID and document IDs containing that word * pne dictionary data structure   + key: word IDs   + values: list of corresponding document IDs |
| URL title | * dictionary/table * contains doc IDs and corresponding URL title * one dictionary data structure   + key: document IDs   + values: URL titles |
| URL description | * dictionary/table * contains sentences in the web page * one dictionary data structure with lists as value   + key: document IDs   + value: list containing texts contained in <p>, <P>, and meta tags   + implemented in \_curr\_parag function |
| Page rank | In order to determine the pagerank for all urls found within the depth set for the crawler, a new dictionary data structure is used.   * outboundLinks   + contains the outbound links for a given url   + one dictionary data structure     - key: doc id     - values: corresponding outbound links’ doc ids   The pageranks are calculated once the crawler and parsing are complete. The function rankPage is called. rankPage first calculates the inbound links in a dictionary data structure, calculates a preliminary pagerank for all urls, with PR = (1 - d) / N with d = 0.85. After, the whole pagerank algorithm is done 20 times. |

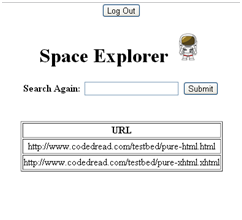
# 3.0 Features

The features provided by the search engine is listed below:

* Google+ Login & Top Keywords Count



* Display Query Results



* Error Handling



# 4.0 Performance

HAProxy is used in order to provide load balancing. Load balancing allows for many users to access the web server at the same time without crashing. The server is composed of three instances: one for the HAProxy load balancing, and two for copies of the front end. Users access the HAProxy and are transferred to one of the front end copies. This prevents the bottleneck effect when multiple users try to access the server. Refer to Data 1 to 4 of the Appendix to compare improvements of the server from lab 3 to 4. An increase from approximately 26 connections to over 1010 connections at once is a great improvement with load balancing. Even with the increase in connections, there is less strain on the cpu since cpu is kept idle for longer. In addition, since users are sent to different copies of the front end, less memory cache is used over a given time. With 1010 concurrent connections, it takes approximately 2786 ms for each request to complete, with transfer rate 1789 Kbyets/sec.

In order to improve performance, rather than requiring haproxy for load balancing, the bottle framework can be modified to allow for a larger number of connections. This allows for only one web server instance since load balancing would be incorporated within the front end code.

# 5.0 Methodology for Benchmarking and Optimization

The bottleneck of the search engine application was determined to be the disk access, because the web pages and results can be uploaded faster than the time it takes to retrieve data from the database tables on disk. Optimization is done through data caching and load balancing. Data caching minimizes communication and data retrieval from disk, thus decreasing the bottleneck passage.

First, templates are used and stored as separate files on disk. The bottle framework allows templates to be compiled only once and cached internally, but can be rendered many times with different keywords arguments. As well, a database of top keywords and their results are stored in memory. Upon start of a session, these data are pre-loaded and stored into cache. As well, the session search results are also stored in cache, in the cases that the user repeats a search.

# 6.0 Description of Design Tree

|  |  |
| --- | --- |
| **Front End** |  |
| **File** | **Description** |
| SpaceExplorer.py | The Front End code. Contains several python functions and calls on the templates. |
| loginpage.tpl | Template for Homepage. Displays logo and search engine name. A google+ login button is also implemented. Upon login, the search bar and submit button is displayed. |
| resultstable.tpl | Template for Resultspage. Displays the logo and search engine name and a table containing the search results. The page also has a search bar and submit button for continued searching. |
| error403.tpl | Error 403 Template. Displays error message “No Permission” and a button to return user back to homepage. |
| error404.tpl | Error 404 Template. Displays error message “File Not Found” and a button to return user back to homepage. |

In order to install the front and back end in a web server install, unpack installPackage.tar.gz and run the shell script install.sh.

|  |  |
| --- | --- |
| **Backend** |  |
| **File** | **Description** |
| installPackage.tar.gz | Contains all necessary files in order to set up the front and back end in a web server instance. It contains:   * BeautifulSoup.py: contains helper functions for the crawler to help parse web pages * bottle.py: set up a web page * crawler.py: parses websites * dbFile.db: an old, default copy of the database created by the crawler * templates: used by the front end to construct the search engine * install.sh: a shell script to set up the front and back end * L4SpaceExplorer.py: front end * urls.txt: starting point for crawler to start parsing webapges |
| spaceKey.pem | Private key to access amazon web server instances |

# 

# 7.0 Contribution

The project was partitioned so that one member (Anny) did the Front End and one member (Kevin) did the Back End. For the performance (lab 4), the Back End member implemented the load balancing, and the Front End member did the caching. Both helped each other during debugging.

# 8.0 Reflection (per individual)

Anny

The labs were challenging, but provided the opportunity to learn and combine a variety of languages (Python, HTML, CSS, Javascript, AJAX) into a single program. The challenging part of the labs was not the requirements but the knowledge of the different languages and the ability needed to combine them. It was very helpful to have a demo (csc326.com), so we can see what is expected of us and can refer to it as an example. It would have been helpful if we were shown how to integrate the different languages in Python before implementation.

Kevin

Through the lab, I have learned a lot about the web pages and servers. I learned about how to create a dynamic website and how to make it available for public use. The most challenging part was learning an unfamiliar language or tool, such as setting up the Amazon Web Service. I have never setup a server before and I found it good experience to have implemented one. I find that a background in web programming would have been very useful in the course, such as previous experience in creating web pages in HTML, CSS, and Javascript.

# 9.0 Course Feedback

Very high work load at the end of the term. Workload should be more evenly distributed throughout the semester. As well the workload is not proportionate to its course mark weight.

Appendix A:

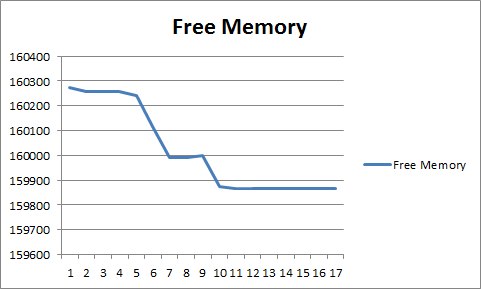
# Benchmarking Graphs & Data

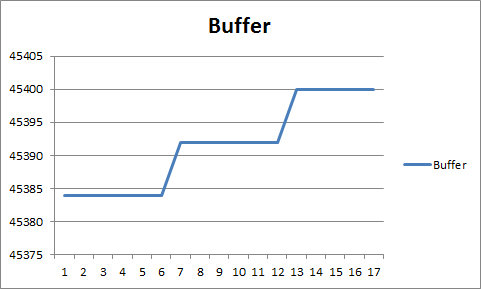
**Data 1:** Previous version (lab 3)

Running -n 2000 -c 10

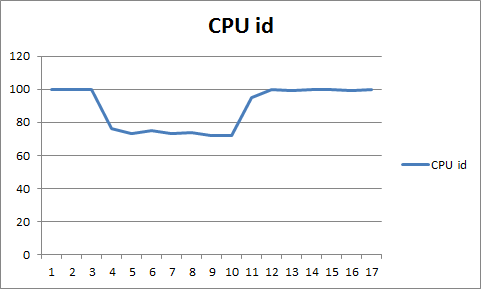
Note: X axis is in seconds.

**Memory Usage**

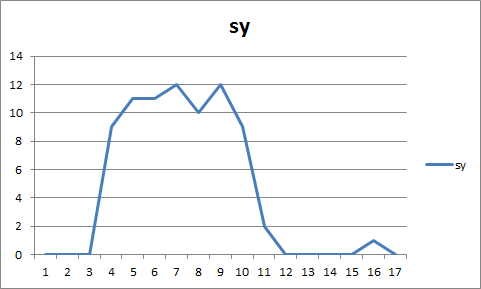




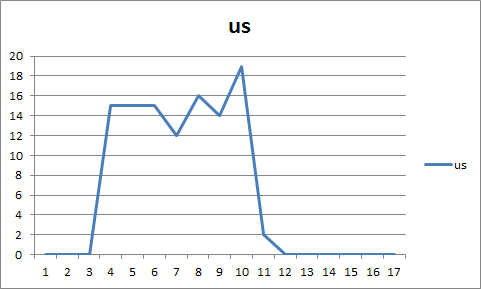
**CPU Usage**



% of time CPU is idle



% of time CPU spent in kernel mode

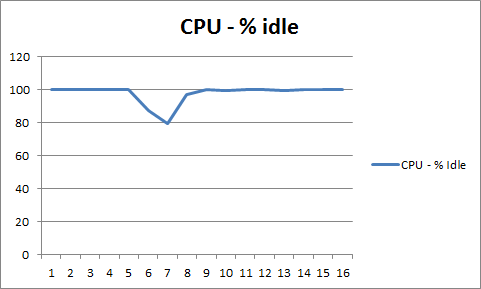


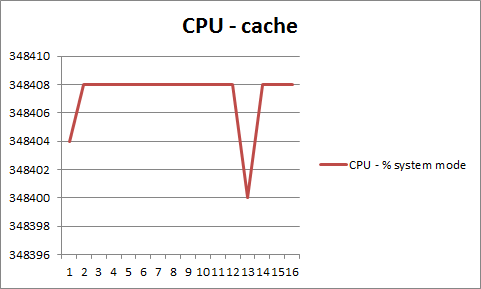
% of time CPU spent in user mode

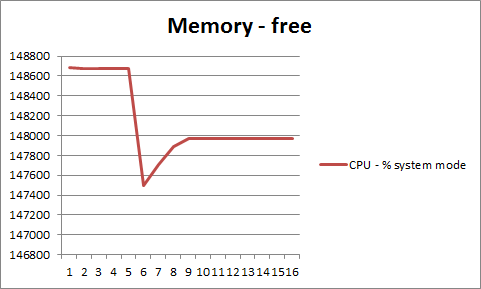
**Data 2:** Lab 4

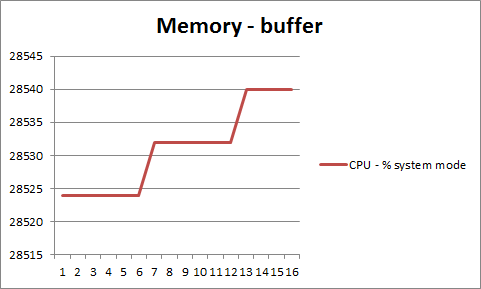
ab -n 1010 -c 1010 http://ec2-54-204-43-178.compute-1.amazonaws.com/

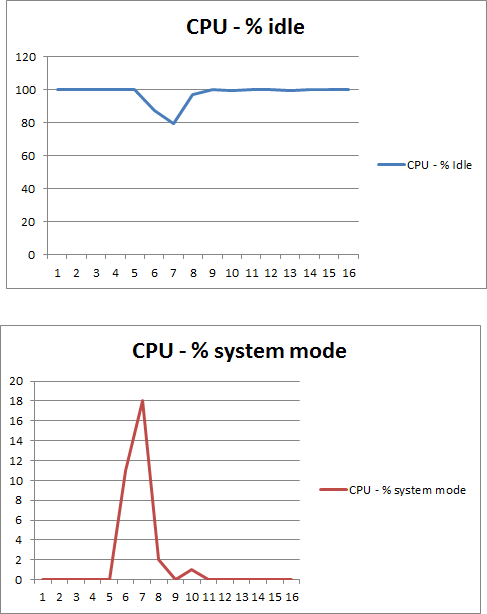
Note: x-axis in seconds











This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-204-43-178.compute-1.amazonaws.com (be patient)

Completed 101 requests

Completed 202 requests

Completed 303 requests

Completed 404 requests

Completed 505 requests

Completed 606 requests

Completed 707 requests

Completed 808 requests

Completed 909 requests

Completed 1010 requests

Finished 1010 requests

Server Software: WSGIServer/0.1

Server Hostname: ec2-54-204-43-178.compute-1.amazonaws.com

Server Port: 80

Document Path: /

Document Length: 4898 bytes

Concurrency Level: 1010

Time taken for tests: 2.786 seconds

Complete requests: 1010

Failed requests: 0

Write errors: 0

Total transferred: 5103530 bytes

HTML transferred: 4946980 bytes

Requests per second: 362.57 [#/sec] (mean)

Time per request: 2785.655 [ms] (mean)

Time per request: 2.758 [ms] (mean, across all concurrent requests)

Transfer rate: 1789.14 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 17 30 7.5 30 43

Processing: 46 680 438.7 554 2729

Waiting: 45 679 438.7 553 2728

Total: 63 710 444.4 591 2772

Percentage of the requests served within a certain time (ms)

50% 591

66% 881

75% 994

80% 1043

90% 1135

95% 1751

98% 1777

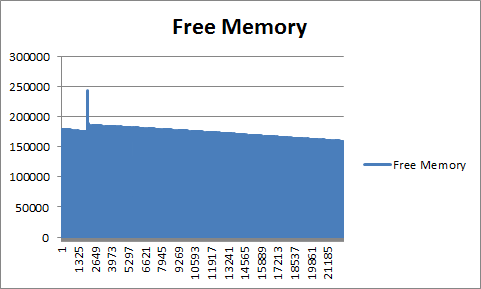
99% 1987

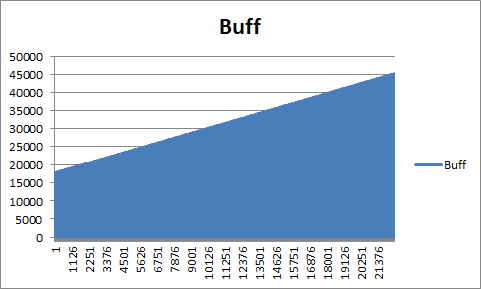
100% 2772 (longest request)

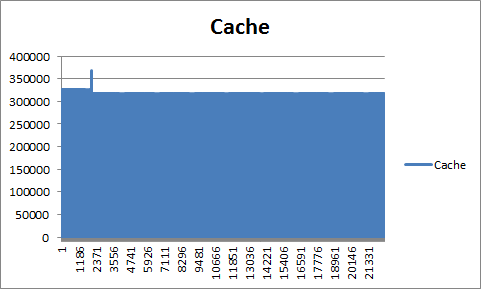
**Data 3:** Running with 100000 requests

Note: X axis is in seconds.

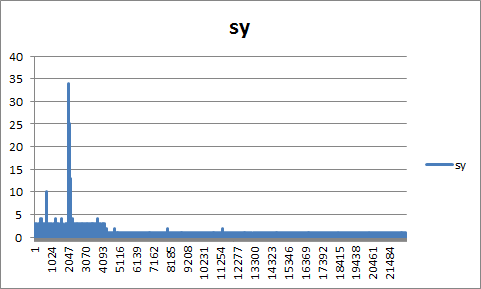
**Memory Usage**



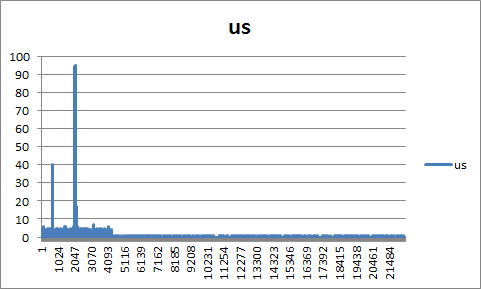




**CPU Usage**



% of time CPU spent in kernel mode

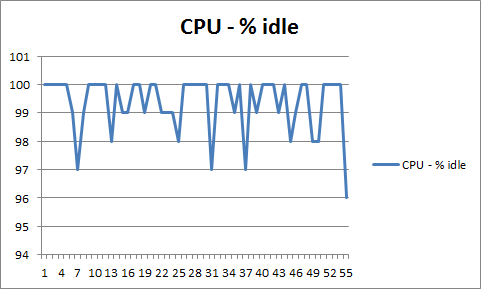


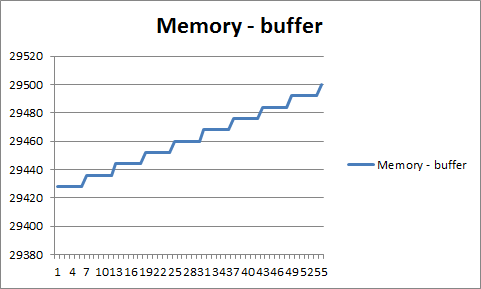
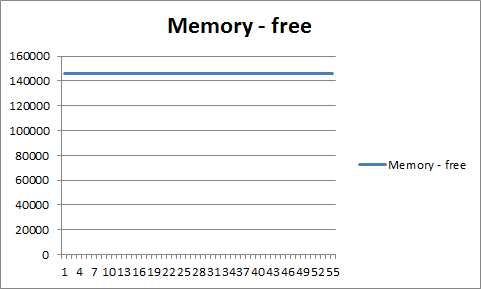
% of time CPU spent in user mode

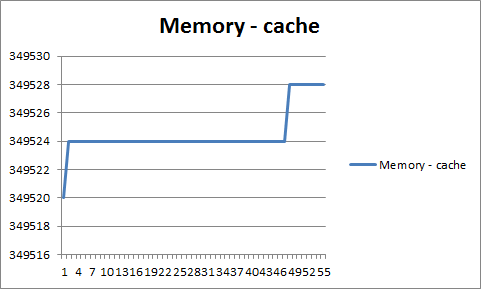
**Data 4:**

ab -n 1010 <http://ec2-54-204-43-178.compute-1.amazonaws.com/>

Note: x-axis in seconds







ab -n 1010 http://ec2-54-204-43-178.compute-1.amazonaws.com/

This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-204-43-178.compute-1.amazonaws.com (be patient)

Completed 101 requests

Completed 202 requests

Completed 303 requests

Completed 404 requests

Completed 505 requests

Completed 606 requests

Completed 707 requests

Completed 808 requests

Completed 909 requests

Completed 1010 requests

Finished 1010 requests

Server Software: WSGIServer/0.1

Server Hostname: ec2-54-204-43-178.compute-1.amazonaws.com

Server Port: 80

Document Path: /

Document Length: 4898 bytes

Concurrency Level: 1

Time taken for tests: 37.666 seconds

Complete requests: 1010

Failed requests: 0

Write errors: 0

Total transferred: 5103530 bytes

HTML transferred: 4946980 bytes

Requests per second: 26.81 [#/sec] (mean)

Time per request: 37.293 [ms] (mean)

Time per request: 37.293 [ms] (mean, across all concurrent requests)

Transfer rate: 132.32 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 17 17 0.2 17 19

Processing: 19 20 3.8 20 52

Waiting: 19 20 3.7 19 51

Total: 36 37 3.8 37 69

Percentage of the requests served within a certain time (ms)

50% 37

66% 37

75% 37

80% 37

90% 37

95% 39

98% 49

99% 63

100% 69 (longest request)

# Appendix B: Progress Report

**CSC326**

**Progress Report**

Group #: 4

Names: Anny Kim Ly (997581718)

Kevin Gumba (997585117)

**Table of Contents**

[Section 1: Lab 1](#h.5acehf9nxsvd)

[Section 1a: FrontEnd](#h.53mni7fmtns2)

[1.0 Design Decisions & Implementations](#h.xi9w4f5ed44y)

[1.0.1 Results Table](#h.qnr33r87w4o7)

[1.0.2 Query Interface](#h.1vytq4hjhxcv)

[1.0.3 Error Page](#h.51sejyaestew)

[1.1 Instructions to Run Code](#h.qai079ciq3md)

[Section 1b: BackEnd](#h.69auhcbv9v8x)

[2.0 Design Decisions & Implementations](#h.mcfw0d43mszl)

[2.0.1 Storing data](#h.egeehgqbdy9c)

[2.1 Instructions to Run Code](#h.6smb62pseoy5)

[Section 2: Lab 2](#h.x9l1v9akpmyi)

[Section 2a: FrontEnd](#h.vuchymqf5958)

[1.0 Design Decisions & Implementations](#h.lv30uanzd05w)

[1.0.1 Results Table](#h.r30flbyb28jt)

[1.0.2 Query Interface](#h.mgvfz7ovbtew)

[1.0.3 Error Page](#h.8171ue84fevb)

[1.1 Instructions to Run Code](#h.4n7urynwdf0o)

[Section 2b: BackEnd](#h.8hn1hvkj0qib)

[2.0 Design Decisions & Implementations](#h.w67tuezg7pjr)

[2.0.1 Pagerank Algorithm](#h.rq83nokwkqr8)

[2.0.2 SQLite3 Persistent Storage](#h.meqcxqprgip)

[2.1 Instructions to Run Code](#h.foxhzn6co6cx)

[Section 3: Lab 3](#h.eegsg3sq02g6)

[1.0 Benchmarking](#h.ml8f5hwgduvg)

[1.0.1 Pagerank Algorithm](#h.y9cysxrmdocr)

[2.0 Frontend](#h.bip2r5hn89st)

[3.0 Backend](#h.d1m68v7dl9el)

[Section 4: Lab 4](#h.mtberplkywgh)

[1.0 Metrics](#h.ckunbp2go7e1)

[2.0 Bottlenecks and Improvements](#h.x71quagcqbov)

[2.0.1 HAproxy](#h.q8n6t5lwwhgd)

[2.0.2 Reduce update repetition](#h.62xyvsa3xjhp)

[2.0.3 Reduce Disk I/O Access](#h.a4oclyh8iuh)

[2.0.3a Top Keywords - Preload and Caching](#h.4mp0toyx4oja)

[2.0.3b Browser Caching](#h.ql3i6un86fia)

[2.0.3c Template Caching](#h.t0r59f6u9zpx)

[3.0 Performance Results](#h.8lg3i26oe2kt)

# Section 1: Lab 1

## Section 1a: FrontEnd

### 1.0 Design Decisions & Implementations

#### 1.0.1 Results Table

The front end accesses the database created by the back end and returns a list of URLs that contains the first word of the query string. The URLs,ordered by their pagerank, are then displayed in a table format along with their title and description (BONUS).

The results table is vertically scrollable for when the 10 results are too big to fit on 1 page. The length varies depending on the descriptions of the URLs.

Pagination was used to limit the number of results displayed. A maximum of 10 URLs can be displayed per page, with the last page containing the remaining results. At the bottom of the search page, page buttons are displayed to navigate through the results.

#### 1.0.2 Query Interface

On the results page, a query interface is displayed at the top of the webpage for easy searching of new keywords.

#### 1.0.3 Error Page

If the user tries to access a web page or a file that does not exist on the website, they will be redirected to the error page that includes a button that can return them to the homepage.

### 1.1 Instructions to Run Code

**Note:** Please run “python crawler.py” in the Backend folder first before running SpaceExplorer.py. A dbFile.db must first be created in order to access persistent data.

To run the program, **simply type “python <file name>.py” in the command line** to start up the search webpage. In this case the file name is “SpaceExplorer.py”. On execution of the file, the query page can be accessed through “<http://localhost:8080/>”. The user will be presented with a simple interface to submit a keyword or phrase. Once the “search” button is clicked, the user will be redirected to the results page. There, they can then find a results table with a list of URLs ordered by their page rank along with additional detail such as title and description (BONUS). If there are more than 10 results, then pagination will occur. The first 10 results will be listed on page 1 and the next 10 results on page, etc. A maximum of 10 results are displayed per page, with the last page containing the remaining results (which will be less than or equal to 10).

## Section 1b: BackEnd

### 2.0 Design Decisions & Implementations

#### 2.0.1 Storing data

There are 3 main data files created within the modified crawler.

* lexicon (variable name: lex)
  + contains word ID and corresponding word
  + one dictionary data structure
    - key: word IDs
    - values: corresponding words
  + implemented in word\_id function
* document index (docIndex)
  + contains document ID and corresponding URL
  + one dictionary data structure
    - key: document IDs
    - values: corresponding URL
  + implemented in crawl function
* inverted index (invIndex)
  + contains word ID and document IDs containing that word
  + pne dictionary data structure
    - key: word IDs
    - values: list of corresponding document IDs
  + implemented in \_add\_text function

All three main data files use dictionaries since the data structure allow the use of keys, making quick insertion and retrieval of data. This is a a very useful feature for data handling in search engines since it reduces wait time for users. Moreover, since BeautifulSoup API creates unique word IDs and document IDs, there is no need to continually modify the key and value pairs once inserted. Therefore, data structures like lists are not required since its main use is for continual addition, modification, and deletion of data.

The tradeoff for using dictionaries is that the data structure does not have built in functions to modify stored data (such as list.sort() or append). Therefore, if needed, all values must be moved to another data structure, such as lists, and then be used for data handling. However, dictionaries allow lists as its value (as seen with our inverted index). Therefore, if needed, dictionaries can later be modified to include this feature.

The inverted index contains a dictionary with a list as its value since multiple document IDs can correspond to a single word ID key. A list in a dictionary allows for many document IDs to be appended without overwriting the previous values.

For the bonus mark, the URL’s title and page descriptions are stored.

* URL title (variable name: docTitle)
  + contains doc IDs and corresponding URL title
  + one dictionary data structure
    - key: document IDs
    - values: URL titles
  + implemented in \_visit\_title function
* URL description (docParDescript)
  + contains sentences in the web page
  + one dictionary data structure with lists as value
    - key: document IDs
    - value: list containing texts contained in <p>, <P>, and meta tags
  + implemented in \_curr\_parag function

Two separate data structures are created for the bonus mark to allow for simplicity of code. Separate dictionaries allow for independence of data, decreasing the chance for errors when coding. Moreover, dictionaries allow the use of keys, making quick access to data.

The main tradeoff for creating two separate data structures for URL titles and descriptions is memory use. These information could have been stored in the same data structure as the document index. However, this would lead to more complex code which is prone to bugs. Additional data handling would have been required (ex. more for loops), possibly increasing the wait time for data retrieval. Also, since access to data is more of a priority than memory, this feature of having one main dictionary was not taken into consideration.

### 2.1 Instructions to Run Code

The main web pages used to crawl the web are crawler test pages. This was chosen for its simplicity, allowing for easy debugging of code. Use of more complex web pages may hide bugs and would make fixing code a lot harder. Once the basics of the crawler work on the test pages, for the next phase, a more complex URL could be used.

* urls.txt:
  + <http://www.codedread.com/test-crawlers.html>
  + <http://www.york.ac.uk/teaching/cws/wws/webpage1.html>

Two main functions are required in order for the user to access crawled data.

To run the function, **simply type “python <file name>.py” in the command line** to receive the output. Note that depth is set to 1 in the main function.

* crawler.get\_inverted\_index()
  + lab1tester1.py
  + Returns a dictionary
    - key: word IDs
    - values: set of corresponding document IDs
* crawler.get\_resolved\_inverted\_index()
  + lab1tester2.py
  + Returns a dictionary
    - key: document IDs
    - values: set of corresponding URLs

# Section 2: Lab 2

## Section 2a: FrontEnd

### 1.0 Design Decisions & Implementations

#### 1.0.1 Results Table

The front end accesses the database created by the back end and returns a list of URLs that contains the first word of the query string. The URLs,ordered by their pagerank, are then displayed in a table format along with their title and description (BONUS).

The results table is vertically scrollable for when the 10 results are too big to fit on 1 page. The length varies depending on the descriptions of the URLs.

Pagination was used to limit the number of results displayed. A maximum of 10 URLs can be displayed per page, with the last page containing the remaining results. At the bottom of the search page, page buttons are displayed to navigate through the results.

#### 1.0.2 Query Interface

On the results page, a query interface is displayed at the top of the webpage for easy searching of new keywords.

#### 1.0.3 Error Page

If the user tries to access a web page or a file that does not exist on the website, they will be redirected to the error page that includes a button that can return them to the homepage.

### 1.1 Instructions to Run Code

To run the program, **simply type “python <file name>.py” in the command line** to start up the search webpage. In this case the file name is “SpaceExplorer.py”. On execution of the file, the query page can be accessed through “<http://localhost:8080/>”. The user will be presented with a simple interface to submit a keyword or phrase. Once the “search” button is clicked, the user will be redirected to the results page. There, they can then find a results table with a list of URLs ordered by their page rank along with additional detail such as title and description (BONUS). If there are more than 10 results, then pagination will occur. The first 10 results will be listed on page 1 and the next 10 results on page, etc. A maximum of 10 results are displayed per page, with the last page containing the remaining results (which will be less than or equal to 10).

## Section 2b: BackEnd

### 2.0 Design Decisions & Implementations

#### 2.0.1 Pagerank Algorithm

An iterative damping factor is used in order to calculate the page ranks of the urls. The calculation is shown below from <http://en.wikipedia.org/wiki/PageRank> under Computation, Iterative:  
 

PR is the page rank of a given url, d is the damping factor, and N is the amount of outbound links for that page. A summation of the inbound links’ page ranks divided by the amount of outbound link they have are included.  
  
In order to determine the pagerank for all urls found within the depth set for the crawler, a new dictionary data structure is used.

* outboundLinks
  + contains the outbound links for a given url
  + one dictionary data structure
    - key: doc id
    - values: corresponding outbound links’ doc ids

The pageranks are calculated once the crawler and parsing are complete. The function rankPage is called. rankPage first calculates the inbound links in a dictionary data structure, calculates a preliminary pagerank for all urls, with PR = (1 - d) / N with d = 0.85. After, the whole pagerank algorithm is done 20 times.

The drawback for this algorithm is that it is not memory efficient since three data dictionaries are created. However, the code allows for easy readability and debugging.

#### 2.0.2 SQLite3 Persistent Storage

All dictionary data structures are stored using sqlite3. All dictionary data structures are stored in their own table. This allows for simplicity of code for easy debugging and data handling in the front end. The drawback is the inefficiency use of memory since many tables are created.

Storing of data structures with SQL is done once the crawler and parsing is complete. Function storeInSQL is called. Tables are first created if they do not currently exists. This allows to preserve and not overwrite old tables. Later, checks using insert or update are used in order to determine if a tuple exists or not.

### 2.1 Instructions to Run Code

To run the function, **simply type “python <file name>.py” in the command line** to receive the output.

* crawler.get\_page\_ranks()
  + lab2tester1.py
  + Returns a dictionary with page ranks. Print page ranks for sites found within depth 1.
* crawler.get\_SQL\_from\_word(word)
  + lab2tester2.py
  + Outputs all information stored in SQL corresponding to word ”about”. Prints corresponding url, title, description, and page rank

# Section 3: Lab 3

### 1.0 Benchmarking

#### 1.0.1 Pagerank Algorithm

The Apache benchmarking tool was used to evaluate the performance of the web server. Performance is measured through the maximum number of connections and maximum number of requests that can be handled. CPU and memory usage was also tracked during the performance runs. As well, Google PageSpeed Insights was used to check for performance optimization methods.

1.0.2 Pagerank Algorithm

*a) Maximum number of Connections*

Our current server can handle 27 connections before connections are dropped. The connections are increased until a maximum number of connection is found. The following pages shows cpu and memory usages during connection and data retrieved while running apache benchmarking software.

Data 1 shows the successful performance results from running the apache benchmark using 27 concurrent requests with 27 requests in total, thus using 27 connections. Data 2 shows that the experiment using 28 connections, with one request per concurrent request, failed before completion. The connection to the web server gets reset after 27 requests thus the maximum number of connections is determined to be 27.

*b) Maximum number of requests that can be handled*

Our current server can go over 9 000 requests before the server crashes. The test is left running overnight to collect the results.

Data 3 shows the successful results from running the apache benchmark using 100000 requests.

*c) CPU and Memory Usage*

Data 4 and Data 5 shows the CPU and memory usage during the apache benchmarking experiments. Please note x axis is in seconds.

As the amount of users request the website, the EC2 memory decreases over time. However, CPU becomes idle at certain times.

Data 4 shows server receiving 100000 requests. Evident by the graphs, more memory is used as more uses requests to access the website. However, without setting an amount of connections (no -c), CPU is kept near idle almost all the time.

Data 5 showsrunning apache with -n 2000 -c 10. As more connections are used, more CPU is used in order to handle the amount of requests. More memory is used as well during the time of requests.

**Data 1:** Max 27 connections

ab -n 27 -c 27 http://ec2-54-225-76-58.compute-1.amazonaws.com/

This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-225-76-58.compute-1.amazonaws.com (be patient).....done

Server Software: WSGIServer/0.1

Server Hostname: ec2-54-225-76-58.compute-1.amazonaws.com

Server Port: 80

Document Path: /

Document Length: 5362 bytes

Concurrency Level: 27

Time taken for tests: 1.604 seconds

Complete requests: 27

Failed requests: 0

Write errors: 0

Total transferred: 148959 bytes

HTML transferred: 144774 bytes

Requests per second: 16.83 [#/sec] (mean)

Time per request: 1603.828 [ms] (mean)

Time per request: 59.401 [ms] (mean, across all concurrent requests)

Transfer rate: 90.70 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 17 17 0.3 17 18

Processing: 19 600 590.6 468 1586

Waiting: 18 599 590.6 467 1585

Total: 36 617 590.9 485 1603

Percentage of the requests served within a certain time (ms)

50% 265

66% 708

75% 712

80% 1597

90% 1601

95% 1602

98% 1603

99% 1603

100% 1603 (longest request)

**Data 2:** Lost connection at 28

ab -n 28 -c 28 http://ec2-54-225-76-58.compute-1.amazonaws.com/

This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-225-76-58.compute-1.amazonaws.com (be patient)...apr\_socket\_recv: Connection reset by peer (104)

Total of 27 requests completed

**Data 3:** Running with 100000 requests

ab -n 100000 http://ec2-54-225-76-58.compute-1.amazonaws.com/

This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-225-76-58.compute-1.amazonaws.com (be patient)

Completed 10000 requests

Completed 20000 requests

Completed 30000 requests

Completed 40000 requests

Completed 50000 requests

Completed 60000 requests

Completed 70000 requests

Completed 80000 requests

Completed 90000 requests

Completed 100000 requests

Finished 100000 requests

Server Software: WSGIServer/0.1

Server Hostname: ec2-54-225-76-58.compute-1.amazonaws.com

Server Port: 80

Document Path: /

Document Length: 5362 bytes

Concurrency Level: 1

Time taken for tests: 3671.039 seconds

Complete requests: 100000

Failed requests: 0

Write errors: 0

Total transferred: 551700000 bytes

HTML transferred: 536200000 bytes

Requests per second: 27.24 [#/sec] (mean)

Time per request: 36.710 [ms] (mean)

Time per request: 36.710 [ms] (mean, across all concurrent requests)

Transfer rate: 146.76 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 16 18 15.8 17 1023

Processing: 18 19 6.5 18 678

Waiting: 17 18 6.5 18 677

Total: 34 37 17.1 35 1042

Percentage of the requests served within a certain time (ms)

50% 35

66% 35

75% 36

80% 36

90% 36

95% 37

98% 60

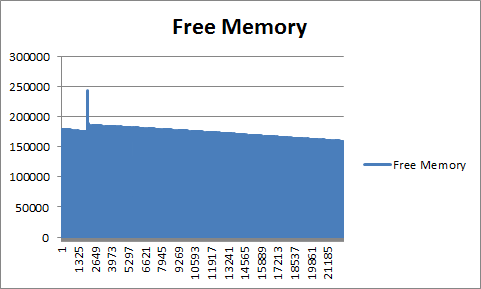
99% 63

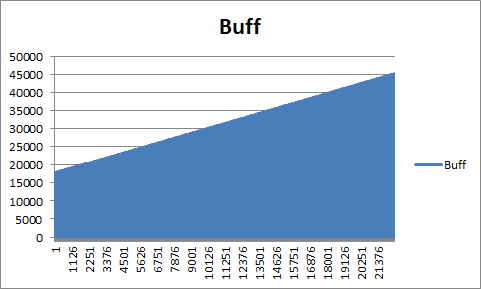
100% 1042 (longest request)

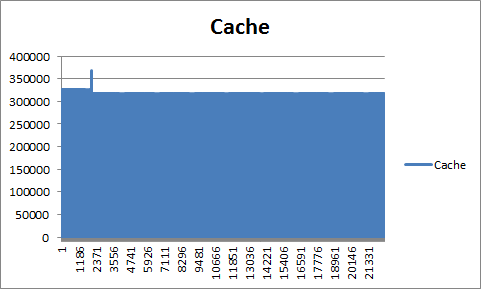
**Data 4:** Running with 100000 requests

Note: X axis is in seconds.

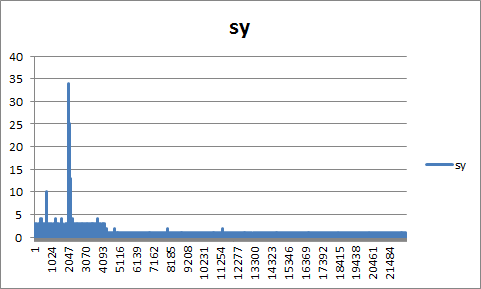
**Memory Usage**



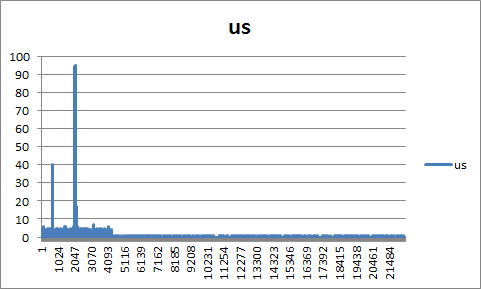




**CPU Usage**



% of time CPU spent in kernel mode

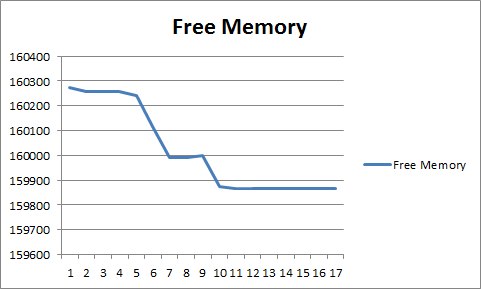


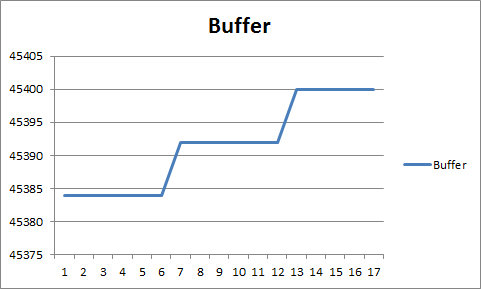
% of time CPU spent in user mode

**Data 5:** Running -n 2000 -c 10

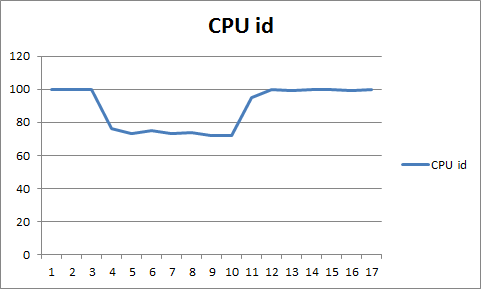
Note: X axis is in seconds.

**Memory Usage**

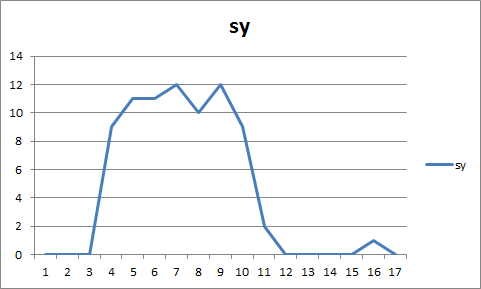




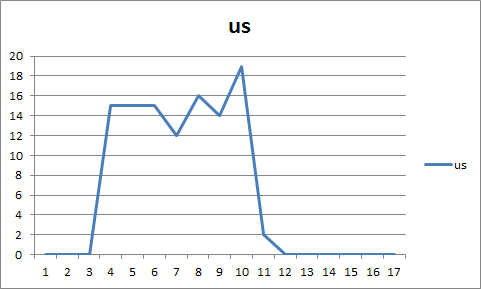
**CPU Usage**



% of time CPU is idle



% of time CPU spent in kernel mode



% of time CPU spent in user mode

#### **2.0 Frontend**

2.0.1 Homepage

The homepage of the search engine only displays the name and logo, as well as the Google+ sign-in button. After the user signs in to Google+ and allows the search engine application access, the search bar and submit button is then displayed. The Google+ sign-in button is then replaced by a log out button. The Google+ Sign In mechanism is done with Javascript.

2.0.2 Results page

On the results page, a log out button is displayed at the top of the webpage for users to sign out of their Google+ account. Once they are signed out, they will be redirected to the home page and not allowed to access to the search engine feature until they sign in again.

2.0.3 Error code

If the user tries to access a web page without signing in to their Google accounts, they will be redirected to the error 403 page that includes a button that can return them to the homepage.

2.0.4 Instructions to run code

**Note:** Please keep **pop-ups disabled** when accessing the search engine.

Please run “python crawler.py” in the Backend folder first before running SpaceExplorer.py. A dbFile.db must first be created in order to access persistent data.

To run the program, simply type “python <file name>.py” in the command line to start up the search webpage. In this case the file name is “SpaceExplorer.py”, thus do “python Space Explorer.py”.

On execution of the file, the home page can be accessed through “[http://ec2-54-225-76-58.compute-1.amazonaws.com](http://ec2-54-225-76-58.compute-1.amazonaws.com/)”. The user will first be presented with a login page consisting of a Google+ sign in button and the name and logo of the search engine. Once signed in, a simple interface to submit a keyword or phrase will appear. A log out button can also be accessed at the top of the webpage. Once the “search” button is clicked, the user will be redirected to the results page. There, they can then find a results table with a list of URLs ordered by their page rank along with additional detail such as title and description. Pagination will occur if there are more than 10 results. The user can also sign out using the log out button at the top of the results page.

#### **3.0 Backend**

2.0.1 Elastic IP Address

The site is located at <http://ec2-54-225-76-58.compute-1.amazonaws.com/>

The server is run on an ongoing basis using the screen command. A screen is created inside the instance, the contents on the installPackage.tar.gz is placed inside and and installed, and afterwards detached. Using the screen command on Ubuntu allows for reattachment later or termination if the search engine is needed to be taken down. The public IP address is 54.225.76.58 and is accessed using the spaceKey.pem.

2.0.1 installPackage.tar.gz

The tar file containing the search engine is extracted using command tar ­zxvf installPackage.tar.gz

The tar file installPackage contains 6 main files:

* BeautifulSoup.py - Aids crawler.py in parsing web pages
* bottle.py - Aids L3SpaceExplorer.py in running a server
* crawler.py - Back end code that data handles and stores web page information
* install.sh - Installs and runs search engine
* dbFile.db - Allows the server to have default values.
* L3SpaceExplorer.py - Front end code that creates user interface
* urls.txt - Contains sites where crawler.py shall crawl first

Run command tar ­zxvf installPackage.tar.gz inside the instance to extract the files.

2.0.1 install.sh

The search engine server is installed in an unmodified Ubuntu AMI by running bash install.sh

The shell script allows running the crawler, the allowing permission for the front end L3SpaceExplorer.py to use restricted ports, and to finally run the front end code.

Rather than importing files via shell script, files were downloaded and placed into the zip file. This makes sure that the correct version of each source is correct in exchange for increase memory needed to store in the tar file.

Run command bash install.sh once tar file is extracted.

**Data 6:** aws ec2 describe-instances

Reservations": [

{

"OwnerId": "125137181039",

"ReservationId": "r-831be0e5",

"Groups": [

{

"GroupName": "SpaceCadet",

"GroupId": "sg-833878e8"

}

],

"Instances": [

{

"Monitoring": {

"State": "disabled"

},

"PublicDnsName": "ec2-54-225-76-58.compute-1.amazonaws.com",

"RootDeviceType": "ebs",

"State": {

"Code": 16,

"Name": "running"

},

"EbsOptimized": false,

"LaunchTime": "2013-11-06T18:37:26.000Z",

"PublicIpAddress": "54.225.76.58",

"PrivateIpAddress": "10.147.176.235",

"ProductCodes": [],

"StateTransitionReason": null,

"InstanceId": "i-1791a072",

"ImageId": "ami-d0f89fb9",

"PrivateDnsName": "ip-10-147-176-235.ec2.internal",

"KeyName": "spaceKey",

"SecurityGroups": [

{

"GroupName": "SpaceCadet",

"GroupId": "sg-833878e8"

}

],

"ClientToken": null,

"InstanceType": "t1.micro",

"NetworkInterfaces": [],

"Placement": {

"Tenancy": "default",

"GroupName": null,

"AvailabilityZone": "us-east-1b"

},

"Hypervisor": "xen",

"BlockDeviceMappings": [

{

"DeviceName": "/dev/sda1",

"Ebs": {

"Status": "attached",

"DeleteOnTermination": true,

"VolumeId": "vol-879bfcc4",

"AttachTime": "2013-11-06T18:37:30.000Z"

}

}

],

"Architecture": "x86\_64",

"KernelId": "aki-88aa75e1",

"RootDeviceName": "/dev/sda1",

"VirtualizationType": "paravirtual",

"AmiLaunchIndex": 0

}

]

}

]

}

# Section 4: Lab 4

**Site:** ec2-54-204-43-178.compute-1.amazonaws.com

## 1.0 Metrics

In order to test the front and backend of the server, Apache benchmarking is used. Apache benchmarking allows to monitor the network, CPU, and memory usage of the web instance.

## 2.0 Bottlenecks and Improvements

### 2.0.1 HAproxy

The server is set up with HAproxy as the load balancer to 2 frontend instances. Through the Apache benchmarking, it is evident that the server can only hold up to approximately 40 clients at a time before crashing. A more efficient server should be able to hold more clients, of up to 1000 at a time. In order to handle this bottleneck, HAproxy is used. HAproxy acts as a load balancer to transfer user requests to different instances of the web page. This allows for more users to access the server at the same time.

Refer to Data 1 to 4 to see improvements of the server from the old CPU and memory usage to the new results.

### 2.0.2 Reduce update repetition

Before lab 3, the document and URL ID is continually updated everytime the same web page is traversed. Repeatedly updating variables when not needed is time consuming, especially when it is much desirable for crawler.py to finish quickly. In order to delete repetition of code, a simple if statement is added so that the ID is only updated once.

### 2.0.3 Reduce Disk I/O Access

Disk I/O Access contributes greatly to the bottlenecks of the search engine. For each keywords, the Front End must retrieve the results from the database stored in memory. To reduce this, caching is implemented in multiple areas.

#### 2.0.3a Top Keywords - Preload and Caching

On initial loading of the results page, the results of the top 20 keywords are retrieved and preloaded into the cache. If the user searches for a keyword that is one of the top 20 keywords, then the results can be retrieved from the cache instead of from disk.

#### 2.0.3b Browser Caching

During each session, the search results are temporarily stored in the cache. At the end of the session, the cache is cleared. If the user searches for a keyword that was previously searched in that session, then the results can be retrieved from the cache instead of from disk.

#### 2.0.3c Template Caching

Making use of Bottle’s template engine, the web pages are stored as templates in separate files having a .tpl extension. Templates are usually compile only once and cached internally, but can be rendered many times with different keywords arguments.

## 3.0 Performance Results

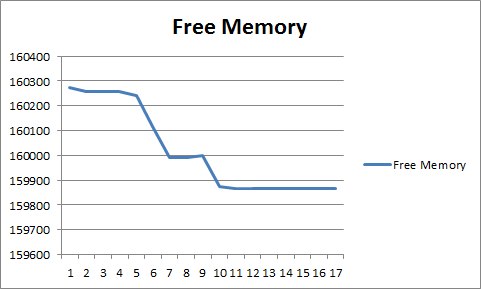
By running 1010 connections using apache, the server is able to handle all requests. CPU only drops to approximately 80% while in our previous version, drops to almost 70% (Refer to Data 1 and 2 graphs related to CPU - % idle). Running with 1010 requests, less memory is used over time as a result of caching (Refer to Data 3 and 4).

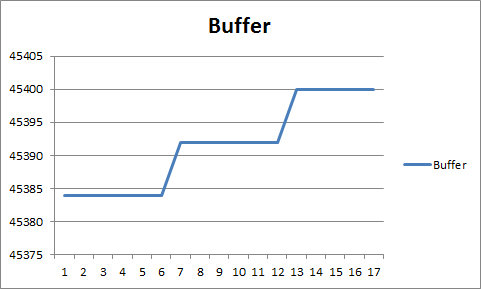
**Data 1:** Previous version (lab 3)

Running -n 2000 -c 10

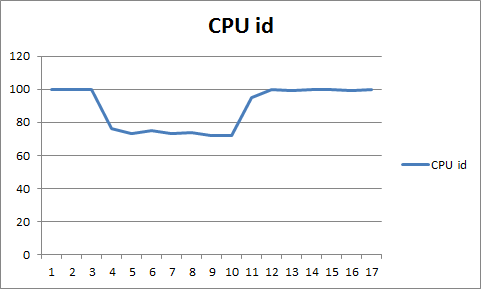
Note: X axis is in seconds.

**Memory Usage**

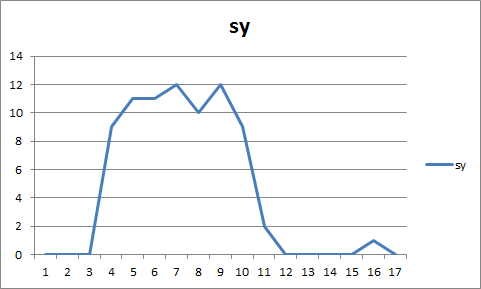




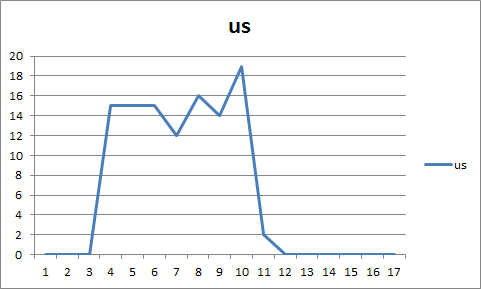
**CPU Usage**



% of time CPU is idle



% of time CPU spent in kernel mode

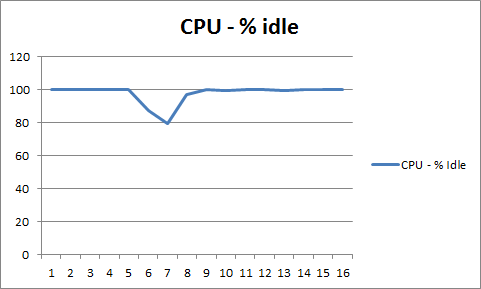


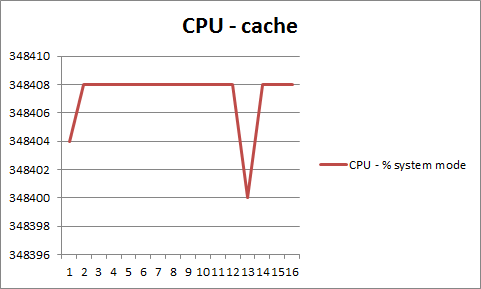
% of time CPU spent in user mode

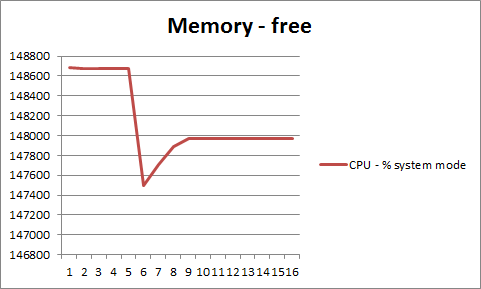
**Data 2:** Lab 4

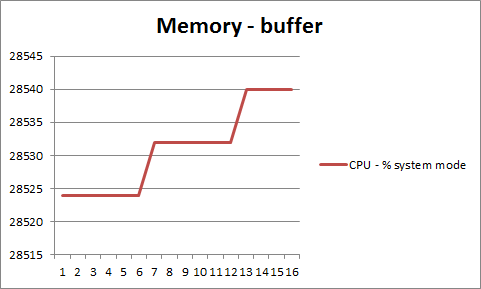
ab -n 1010 -c 1010 http://ec2-54-204-43-178.compute-1.amazonaws.com/

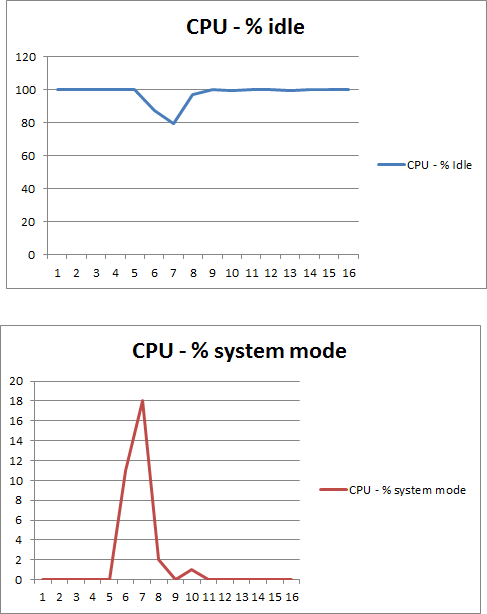
Note: x-axis in seconds











This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-204-43-178.compute-1.amazonaws.com (be patient)

Completed 101 requests

Completed 202 requests

Completed 303 requests

Completed 404 requests

Completed 505 requests

Completed 606 requests

Completed 707 requests

Completed 808 requests

Completed 909 requests

Completed 1010 requests

Finished 1010 requests

Server Software: WSGIServer/0.1

Server Hostname: ec2-54-204-43-178.compute-1.amazonaws.com

Server Port: 80

Document Path: /

Document Length: 4898 bytes

Concurrency Level: 1010

Time taken for tests: 2.786 seconds

Complete requests: 1010

Failed requests: 0

Write errors: 0

Total transferred: 5103530 bytes

HTML transferred: 4946980 bytes

Requests per second: 362.57 [#/sec] (mean)

Time per request: 2785.655 [ms] (mean)

Time per request: 2.758 [ms] (mean, across all concurrent requests)

Transfer rate: 1789.14 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 17 30 7.5 30 43

Processing: 46 680 438.7 554 2729

Waiting: 45 679 438.7 553 2728

Total: 63 710 444.4 591 2772

Percentage of the requests served within a certain time (ms)

50% 591

66% 881

75% 994

80% 1043

90% 1135

95% 1751

98% 1777

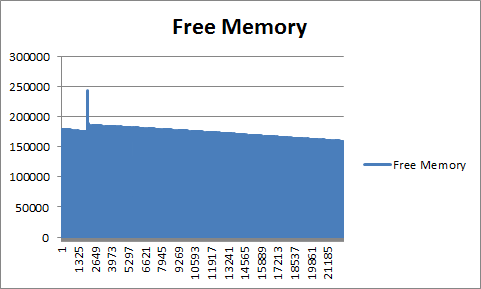
99% 1987

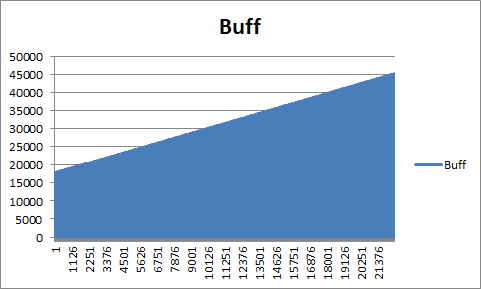
100% 2772 (longest request)

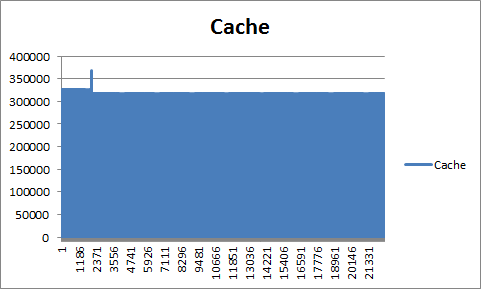
**Data 3:** Running with 100000 requests

Note: X axis is in seconds.

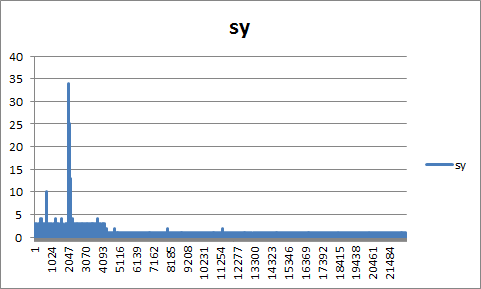
**Memory Usage**



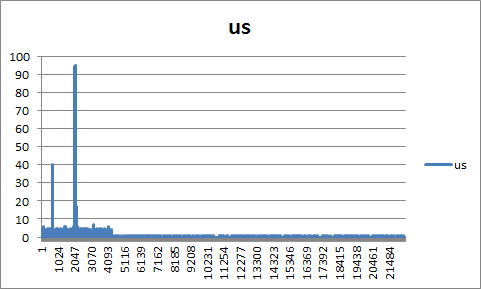




**CPU Usage**



% of time CPU spent in kernel mode

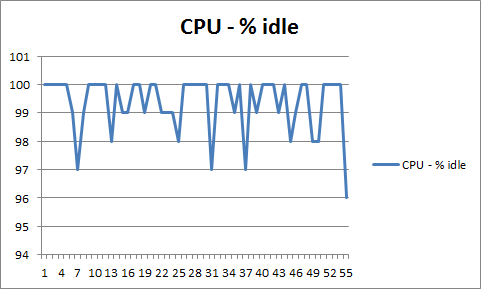


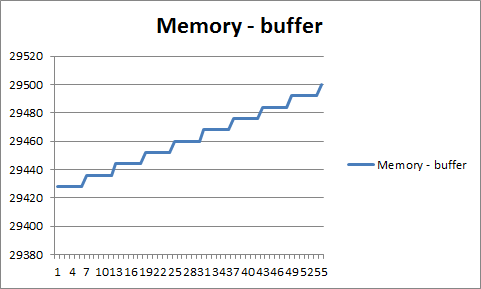
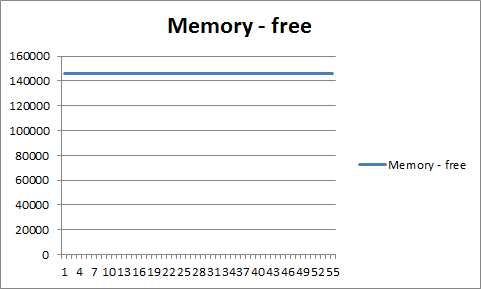
% of time CPU spent in user mode

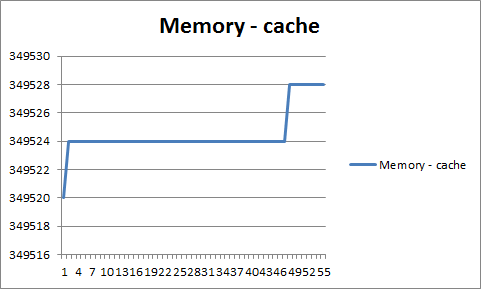
**Data 4:**

ab -n 1010 <http://ec2-54-204-43-178.compute-1.amazonaws.com/>

Note: x-axis in seconds







ab -n 1010 http://ec2-54-204-43-178.compute-1.amazonaws.com/

This is ApacheBench, Version 2.3 <$Revision: 655654 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking ec2-54-204-43-178.compute-1.amazonaws.com (be patient)

Completed 101 requests

Completed 202 requests

Completed 303 requests

Completed 404 requests

Completed 505 requests

Completed 606 requests

Completed 707 requests

Completed 808 requests

Completed 909 requests

Completed 1010 requests

Finished 1010 requests

Server Software: WSGIServer/0.1

Server Hostname: ec2-54-204-43-178.compute-1.amazonaws.com

Server Port: 80

Document Path: /

Document Length: 4898 bytes

Concurrency Level: 1

Time taken for tests: 37.666 seconds

Complete requests: 1010

Failed requests: 0

Write errors: 0

Total transferred: 5103530 bytes

HTML transferred: 4946980 bytes

Requests per second: 26.81 [#/sec] (mean)

Time per request: 37.293 [ms] (mean)

Time per request: 37.293 [ms] (mean, across all concurrent requests)

Transfer rate: 132.32 [Kbytes/sec] received

Connection Times (ms)

min mean[+/-sd] median max

Connect: 17 17 0.2 17 19

Processing: 19 20 3.8 20 52

Waiting: 19 20 3.7 19 51

Total: 36 37 3.8 37 69

Percentage of the requests served within a certain time (ms)

50% 37

66% 37

75% 37

80% 37

90% 37

95% 39

98% 49

99% 63

100% 69 (longest request)