Sombra: Pagerank and Indexing for Searching the English Wikipedia
CS560: Programming Assignment 3

Elliot D. Greenlee and Jared M. Smith

April 31st, 2017

1 Introduction

We implemented a web portal capable of querying results a ranked version of the Simple English Wikipedia dataset for the third programming assignment in CS560: Advanced Operating Systems. For our backend we designed and implemented a distributed version of the PageRank algorithm based on the AWS cloud computing platform. This scalable system calculates the importance of Wikipedia articles by observing internal references. We named our system after Sombra, a character from Overwatch (`https://playoverwatch.com/en-us/`), a modern team-oriented combat game, because this particular character is a supporting hacker, providing information about which enemies should be prioritized. The ultimate goal of our system is to provide the first iteration of a scalable Wikipedia search tool.

Figure 1: Sombra: An Overwatch Hero

2 Design and Implementation

We implemented Sombra as multiple separate portions across a variety of languages, platforms, and programming models. First, preprocessing was done on the Simple English Wikipedia. Next, we applied Google's PageRank algorithm to this data. We incorporated our prior work, Winston (github link), in order to create an inverted index of the data for querying. Lastly, we adapted our existing webportal to work with the new PageRank system.

Web portal = external user interface. Flask app in python, allows user queries on the pre-generated inverted index file, returning the locations in a separate window.

2.1 Preprocessing

Where is the Code: The preprocessing code for the inverted index and PageRank is in src/wikiextractor/WikiExtractor.py and src/pagerank/pagerank.scala

The original dataset for this project came from Simple English Wikipedia (`https://dumps.wikimedia.org/simplewiki/`). Each page is represented in XML in the general format

```
<page>
<title> Page_Name </title>
(other fields we do not care about)
<revision>
<text> (Page body goes here) </text>
</revision>
</page>
```

Specific formatting information is also possible in these features, and must be removed. For the inverted index, initial extraction was completed using the Python WikiExtractor tool (`http://medialab.di.unipi.it/wiki/Wikipedia_Extractor`). WikiExtractor performs template expansion by preprocessing the whole dump and extracting template definitions. The result of this is a JSON object per page with the URL, the text page cleaned content, the page title, and the page ID. Here is an example of one of these pages, with the body text cut down.

```
{"url": "https://simple.wikipedia.org/wiki?curid=1", "text":
    "April\n\nApril is the ... of winter. For example: \"April showers
    bring May flowers.\"\n\n\n", "id": "1", "title": "April"}
```

For PageRank, we used two separate files as the inputs. The file simplewiki-20170401-pagelinks.sql lists each page ID and the pages it links to as SQL insert statements, metadata for the Wikipedia dump. This allowed us to ignore pages that are placeholders or redirections, as they do not contain meaningful texts. Additionally, we did not have to worry about considering only terms that are

titles of other pages. This data was converted for the Spark Graphx implementation of PageRank. Later, once the PageRank algorithm was complete, we converted from page ids to page titles using simplewiki-20170401-page.sql, another metadata file.

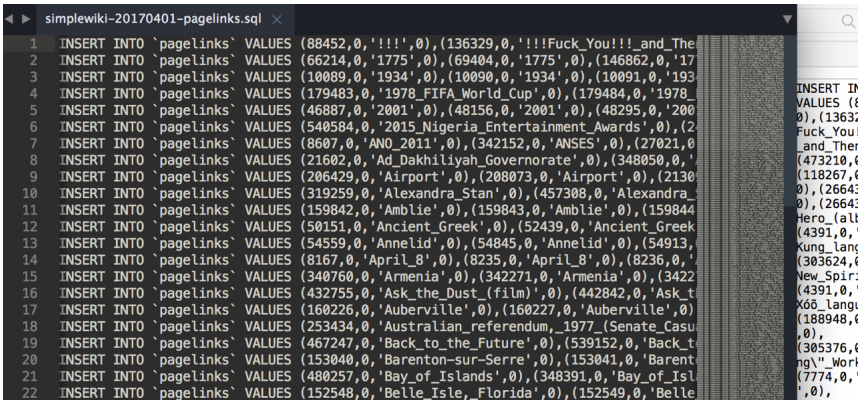

Figure 2: An example from the pagelink.sql metadata file

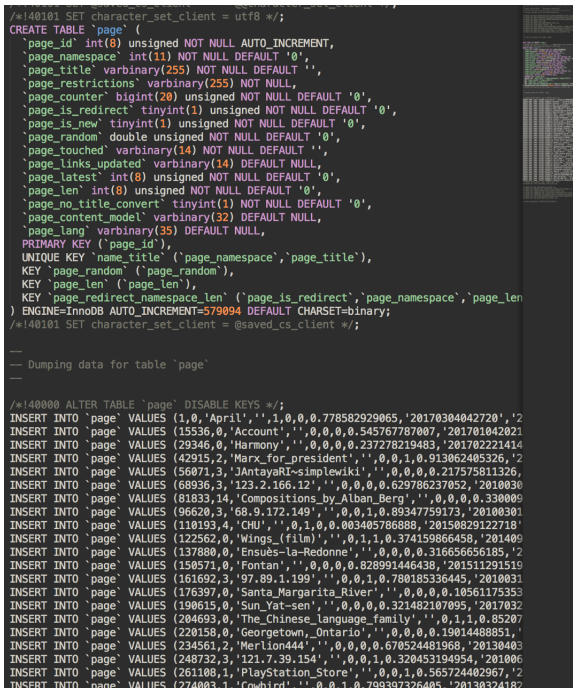

Figure 3: An example from the page.sql metadata file

2.2 PageRank

Where is the Code: The PageRank code for ranking pages is in src/pagerank/pagerank.scala
Where are the Results: The PageRank results are at src/pagerank/pagerank.tar.gz, which can be unzipped with tar -xzvf pagerank.tar.gz on a Linux or Mac system or with a Windows unarchiving tool.
Helper Scripts: The file run_tests.sh will run 25 runs of the algorithm and store the results on disk.

PageRank is an algorithm used on early versions of Google in order to rank the popularity of various pages based on internal references. We implemented a Spark GraphX version of PageRank in Scala using the instructions at `http://mike.seddon.ca/computing-wikipedias-internal-pagerank-with-spark/`. This PageRank algorithm was run on an Amazon Web Services (`https://aws.amazon.com`) cloud computing platform. Rather than running the algorithm for a distinct number of iterations, our implementation runs until convergence to 0.0001, which is the default way Spark's GraphX library does PageRank. We ran the algorithm 25 times for an average of 176 seconds per run, and we recorded the results each time and stored them in our submission directory. Screenshots detailing the exact process can be found below.

2.3 Inverted Index

Where is the Code: The MapReduce code for building the inverted index is in src/invertedindex/InvertedIndexBuilder.java.
Where are the Results: The inverted index results are at src/invertedindex/inverted_index.tar.gz, which can be unzipped with tar -xzvf inverted_index.tar.gz on a Linux or Mac system or with a Windows unarchiving tool.

An inverted index stores mappings from some content to its location in a larger database. Included in this work is the ability to remove stop words based on frequency, but in this assignment we downloaded a stopwords file online and filtered very common English stopwords with the stopwords.txt file in the src/invertedindex directory. In this assignment, we re-purposed our earlier work from Winston (Programming Assignment 2) to create an inverted index mapping words to their locations in the Simple English Wikipedia. This location data consisted of a document ID, line number, and line index (word position), all of which are needed for the later querying ability. This functionality was performed using a MapReduce function in Hadoop on Erebus, a 400+ GB RAM, 58 core, 50 TB research server and took 83 minutes to build the inverted index for all of the English Wikipedia. Screenshots showing this process are below.

2.4 Query

Where is the Code - Query: The Python code for querying the inverted

index is in src/searchengine/searchui/searchui/query.py

The final goal of the project was a querying program on top of the internal analysis, which combines the PageRank and Inverted Index generated earlier. Our query program is a Python 2 program re-purposed from earlier work on our Winston system (Programming Assignment 2), which accepts a user-specified query of one or more words. It obtains keywords hits from the the inverted index over all of Wikipedia, and using a our own ranking algorithm, it determines a good display ranking of the hits for later display. The ranking algorithm we developed is defined by this Python code, which returns a list of results in the form of tuples with the word, document name, line number, and word position (where the word is on the line):

```
1
2  NON_ALPHANUMERIC_CHARS = re.compile('[^a-zA-Z0-9 ]')
3
4  def query(inverted_index, query_string):
5      QueryResult = namedtuple('QueryResult', 'word doc_name line_number
           word_number')
6
7      results = []
8      cleaned_query = re.sub(NON_ALPHANUMERIC_CHARS, '',
           query_string.rstrip())
9      words_in_query = cleaned_query.lower().split(' ')
10
11     for word in words_in_query:
12         word_entry = inverted_index.get(word)
13         if word_entry is not None:
14             doc_names = word_entry.keys()
15
16             ranks = {}
17             for doc_name in doc_names:
18                 rank = pagerank.get_rank(doc_name)
19                 if rank is None:
20                     continue
21                 else:
22                     ranks[doc_name] = rank
23
24             sorted_ranks = sorted(ranks.items(),
                   key=operator.itemgetter(1), reverse=True)
25
26             for ranked_doc_name in sorted_ranks:
27                 word_positions = word_entry[ranked_doc_name[0]]
28                 for line_number, word_number in word_positions:
29                     results.append(QueryResult(word, ranked_doc_name[0],
                           line_number, word_number))
30
31     return results
```

Essentially, this code does the following to get the list of page results:

1. First, we strip the query of non-alphanumeric characters.
2. For each word in the query string:
   (a) We grab the entry in the inverted index for that word. For example, if we search for "Earth", then the inverted index of Wikipedia will give us a dictionary where the keys are the page titles on Wikipedia where the word "Earth" occurs and the values are dictionaries of line numbers and word number (i.e. the word position or where on the line the word is) of the occurence of the word "Earth" in the file for the current key.

(b) Once we have that nested dictionary, for each document name (i.e. page title) where this word occurs:

i. We lookup in our pagerank file, which was generated by Apache Spark and our earlier code, the numerical pagerank value for that document name (i.e. page title) on Wikipedia.

ii. Then, we build another dictionary of document names to the rank for that document.

(c) With this map of document names to ranks, we iterate over the document names we have for the current word in the query string in order of the documents ranked the highest (with the largest numerical page rank) to the documents ranked the lowest:

i. At this point, we're going over the results from the inverted index based on our searched word in the query string **in descending order of rank for the page**, and we simply append to the results list we're building up for this query all the results for the document we're on in the ranked documents

3. Finally, we return the custom-ranked results based on the pagerank and the inverted index!

It then returns the relevant location document IDs, line numbers, and line indexes. Currently, our querying functionality is limited to uncomplicated multiple word queries. However, common operations like "and," "or," and "not" would be feasible on our system. Additionally, we have implemented the underlying word-position logic to search for consecutive words using quote specification such as in the example "fare well." In the future this query would only return the lines where these two words appear consecutively.

2.5 Web Portal

Where is the Code: The web user interface code is in src/searchengine/searchui.

We further improved the query by implementing it in our Python 2 web portal, re-purposed from Winston, and similar to a real search engine. It allows the submission of queries, and then displays the results in a returned page. We built the UI as a Flask App (`https://github.com/sloria/cookiecutter-flask`). Screenshots of the Web UI, search engine are below.

You can run the Web Portal by first install the Python requirements with **pip install -r requirements.txt** then running **honcho start**. Make sure you use Python 2.7.X if you run it. Additionally, you must first have the pagerank.txt file from the pagerank.tar.gz file, the inverted_index.txt file from the inverted_index.tar.gz file, and the page_ids_to_titles.csv file from the inverted_index.tar.gz file in the root of the searchui directory at src/searchengine/searchui in order for the querying program to be able to run queries.

3 Screenshots and Process

In the next set of screenshots, we will show you the following:

1. **Setting Up AWS**: Using AWS to setup an Elastic Map Reduce (EMR) cluster with Apache Spark and HDFS to run the PageRank algorithm in a distributed fashion. We did ours with 3 worker nodes (virtual machines on AWS) and 1 master node (the virtual machine on AWS where the driver program for Spark was run from).
2. **Running PageRank on AWS**: Logged into the AWS master node and running 25 runs of the PageRank algorithm.
3. **Extra Credit: Creating the Inverted Index with Winston**: Logged into a large server to compute the Inverted Index for the English Wikipedia based on Winston, our code for Programming Assignment 2.
4. **Extra Credit: Web UI and Custom Ranking**: The final extra credit portion, which is the Web UI using the query function described above.

3.1 Setting Up AWS

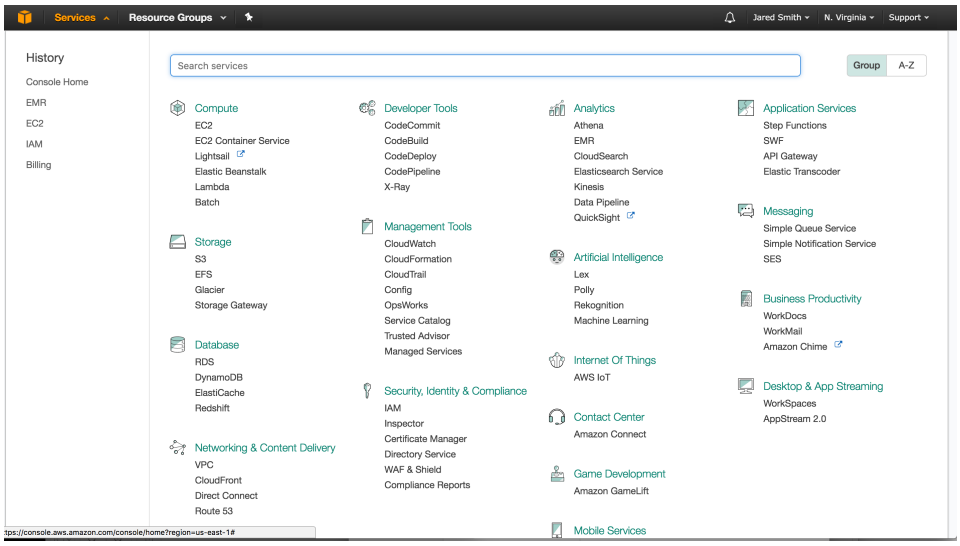

Figure 4: Listing of All AWS Services on the AWS Management Console

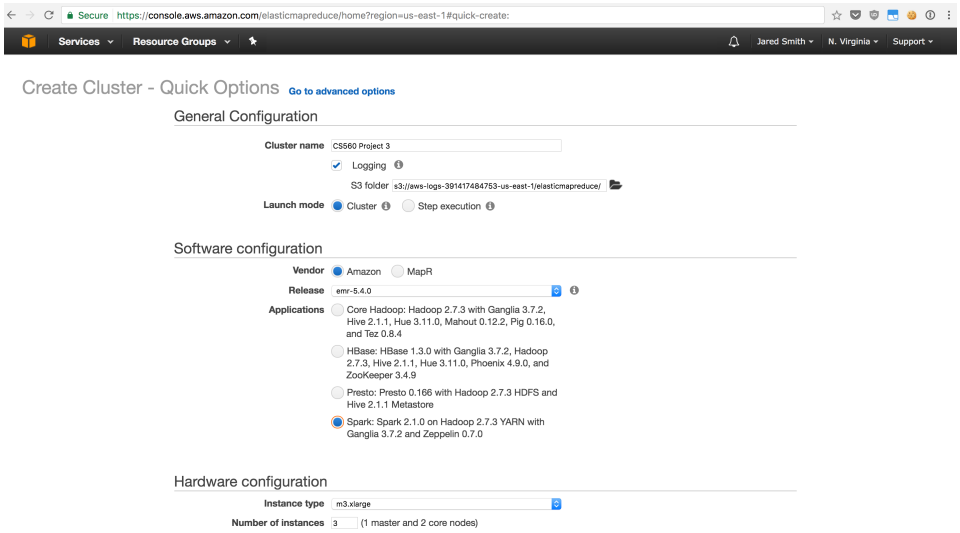

Figure 5: Configuring the AWS EMR cluster, where EMR is Elastic Map Reduce, and is Amazon's pre-built Hadoop/Spark/Kafka/etc. big data service.

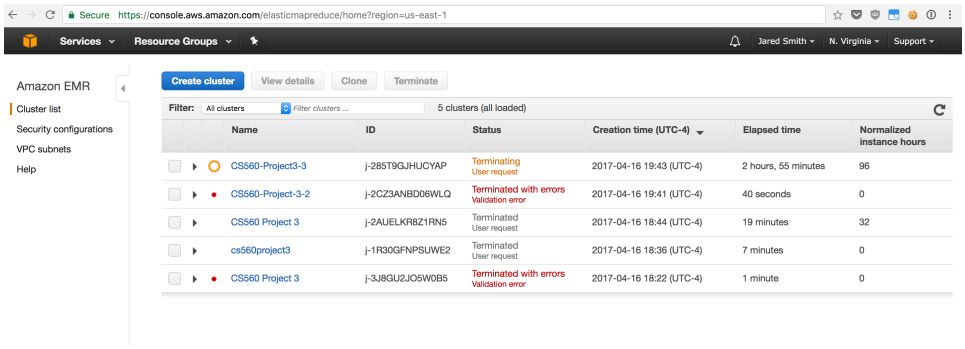

Figure 6: We ran into several jobs where the configuration was incorrectly done. This shows the list of prior failed cluster jobs with bad configurations.

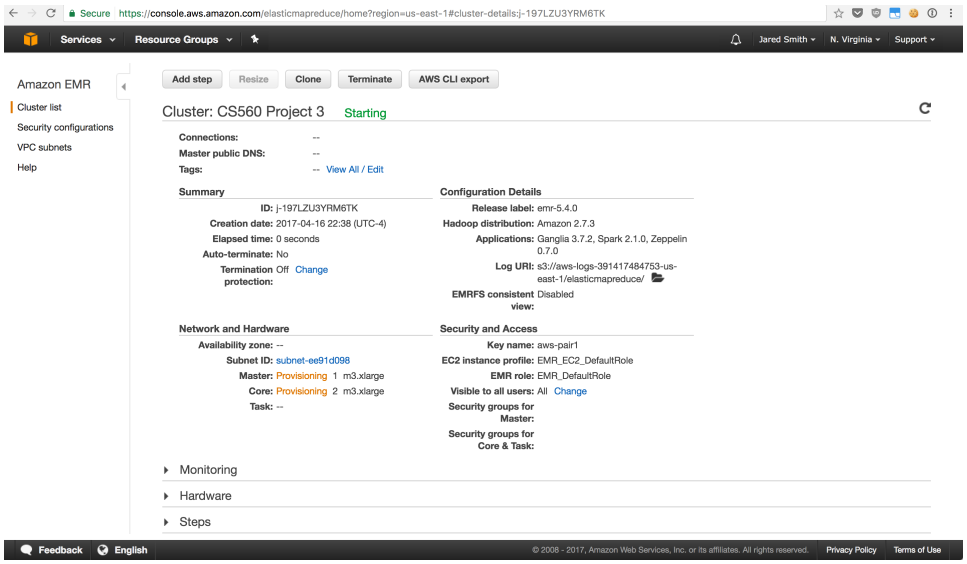

Figure 7: Starting the AWS EMR cluster

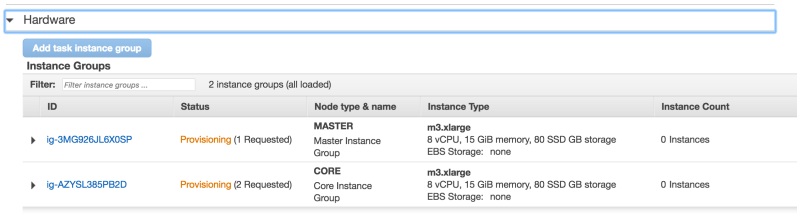

Figure 8: This shows the hardware (virtual machines we requested). You can see it says 1 master node and 3 worker nodes.

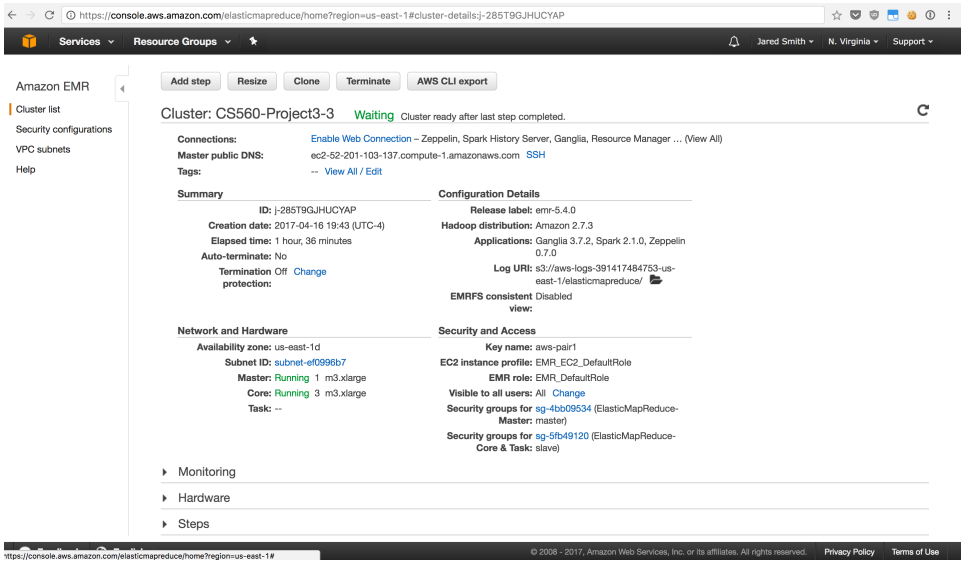

Figure 9: The AWS EMR cluster is now online

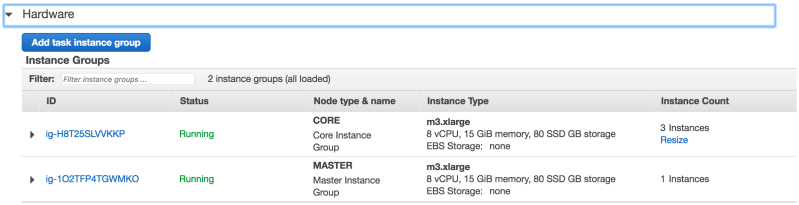

Figure 10: Running the nodes

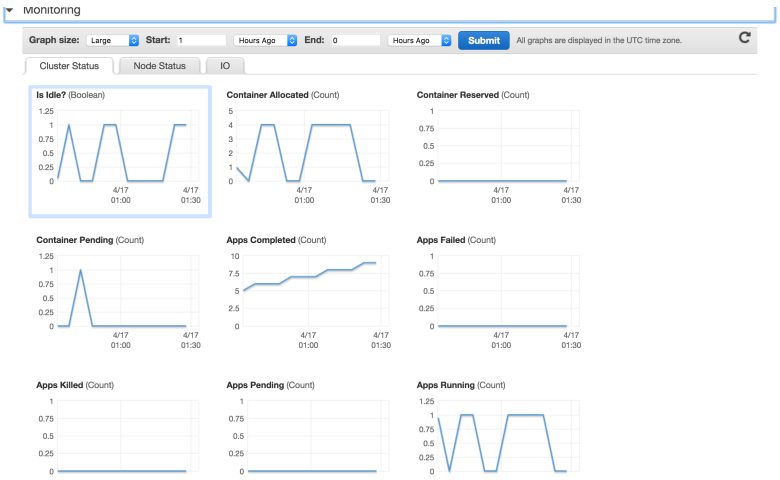

Figure 11: These are some of the graphs for metrics AWS is monitoring on our cluster.

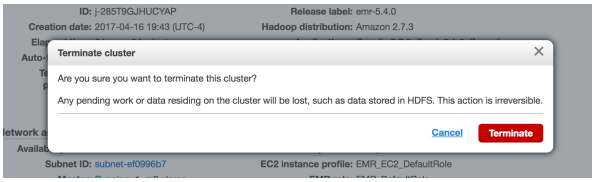

Figure 12: Cluster termination dialog

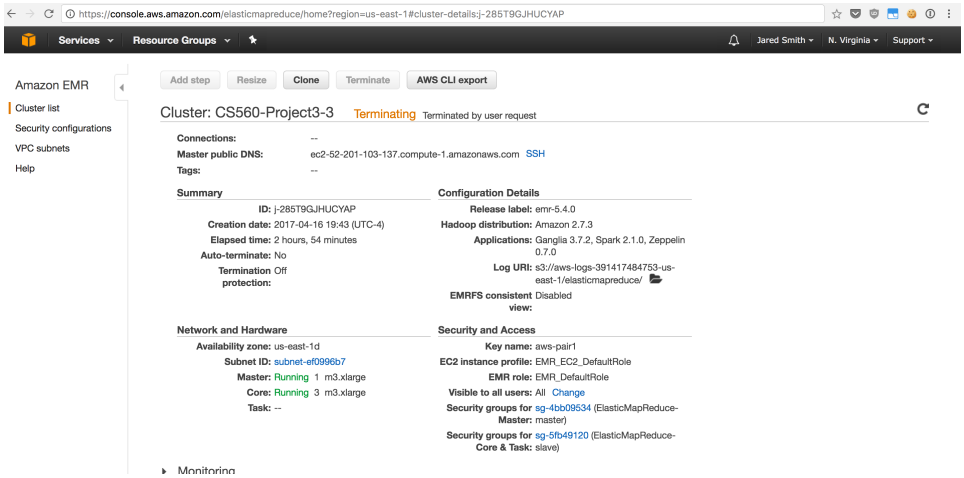

Figure 13: Stopping the AWS cluster

3.2 Running PageRank on AWS

We ran PageRank on AWS with 3 worker nodes and 1 master node. Below are the screenshots of us SSH'd into the master node running the pagerank algorithm. It took on average 176 seconds to compute the pagerank.

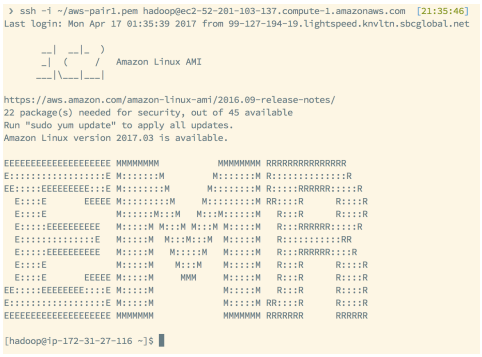

Figure 14: We see this once we log into the master node via SSH, which you can get the address for at the top of the cluster status page above.

Figure 15: The list of input files in HDFS that will be fed to the PageRank algorithm.

Figure 16: Running the pagerank algorithm across all 3 worker nodes, which is a Spark Scala job.

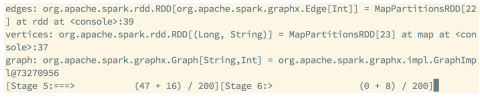

Figure 17: The page rank is running and Spark is at Stage 5 of processing. Recall the Spark paper describes how Spark uses stages to process the Directed Acyclic Graph of tasks it needs to do.

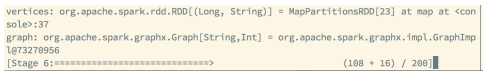

Figure 18: Now at stage 6

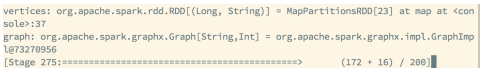

Figure 19: Now at stage 275

```
edges: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]] = MapPartitionsRDD[22
] at rdd at <console>:39
vertices: org.apache.spark.rdd.RDD[(Long, String)] = MapPartitionsRDD[23] at map at <con
sole>:37
graph: org.apache.spark.graphx.Graph[String,Int] = org.apache.spark.graphx.impl.GraphImp
l@73270956
[Stage 1149:>                                                       (0 + 2) / 2]
```

Figure 20: Now at stage 1149

```
scala> titleAndPrGraph.vertices.count()
res9: Long = 174803
```

Figure 21: Now we can see the number of vertices in the pagerank generated by Spark the GraphX library on the Wikipedia data.

```
prGraph: org.apache.spark.graphx.Graph[Double,Double] = org.apache.spark.graphx.impl.Gra
phImpl@57e14b
titleAndPrGraph: org.apache.spark.graphx.Graph[(Double, String),Int] = org.apache.spark.
graphx.impl.GraphImpl@42d6568e
timestampEnd: Long = 1492393192292
duration: Long = 177298
Duration: 177298 ms (~2 minutes)
United_States: 818.2641259896291
Multimedia: 816.6600942196424
France: 395.8886873670224
Definition: 338.43461234428815
English_language: 319.8528823227395
United_Kingdom: 313.98604609382176
International_Standard_Book_Number: 284.0803868582328
Country: 250.93384509869472
Geographic_coordinate_system: 234.02814100972452
England: 233.6676732723399
```

Figure 22: The PageRank job finished with top 10 pages of the pagerank

```
#!/bin/bash

for run in {1..25}
do
  spark-shell -i pagerank.java
done
```

Figure 23: We used this script to automate running 25 runs of the pagerank algorithm and collecting results.

```
[hadoop@ip-172-31-27-116 sombra]$ ls results/
pagerank-174-seconds-10  pagerank-175-seconds-17  pagerank-177-seconds-12
pagerank-174-seconds-18  pagerank-175-seconds-19  pagerank-177-seconds-2
pagerank-174-seconds-25  pagerank-175-seconds-20  pagerank-177-seconds-23
pagerank-174-seconds-8   pagerank-175-seconds-21  pagerank-177-seconds-24
pagerank-174-seconds-9   pagerank-175-seconds-22  pagerank-177-seconds-3
pagerank-175-seconds-1   pagerank-175-seconds-4   pagerank-178-seconds-16
pagerank-175-seconds-11  pagerank-175-seconds-5   pagerank-178-seconds-7
pagerank-175-seconds-13  pagerank-175-seconds-6
pagerank-175-seconds-14  pagerank-176-seconds-15
```

Figure 24: Output files from all iterations where the number in the filename is the number of seconds to finish.

3.3 Extra Credit: Creating the Inverted Index with Winston

We acquired the inverted index with the document name, line number, and word position for every word in the English Wikipedia using Winston, our code from Programming Assignment 2. It took 83 minutes to collect all of the data.

To build the inverted index, run the following commands:

- `python WikiExtractor.py --json -it abbr,b,big,blockquote,center,cite,em,font,h1,h2,h3,h4,hiero,i,kbd,p,plaintext,s,span,strike,strong,tt,u,var ../../data/simplewiki-20170401-pages-meta-current.xml`

Then move the output from wiki extractor to the data directory.

- `./clean_data.py ../../data/text/`
- `number_input.py --i cleaned_pages --o numbered_cleaned_pages`
- `mv numbered_cleaned_pages input`
- `javac -classpath "/hadoop/etc/hadoop:/hadoop/share/hadoop/common/lib/*:/hadoop/share/hadoop/common/*:/hadoop/share/hadoop/hdfs:/hadoop/share/hadoop/hdfs/lib/*:/hadoop/share/hadoop/hdfs/*:/hadoop/share/hadoop/yarn/lib/*:/hadoop/share/hadoop/yarn/*:/hadoop/share/hadoop/mapreduce/lib/*:/hadoop/share/hadoop/mapreduce/*:/contrib/capacity-scheduler/*.jar" -d ./build *.java`
- `jar cvf InvertedIndex.jar *`
- `mkdir output`
- `hadoop jar InvertedIndex.jar elliotjared.programmingassignmenttwo.InvertedIndexBuilder input output`

Figure 25: This shows the steps we took to generate the inverted index.

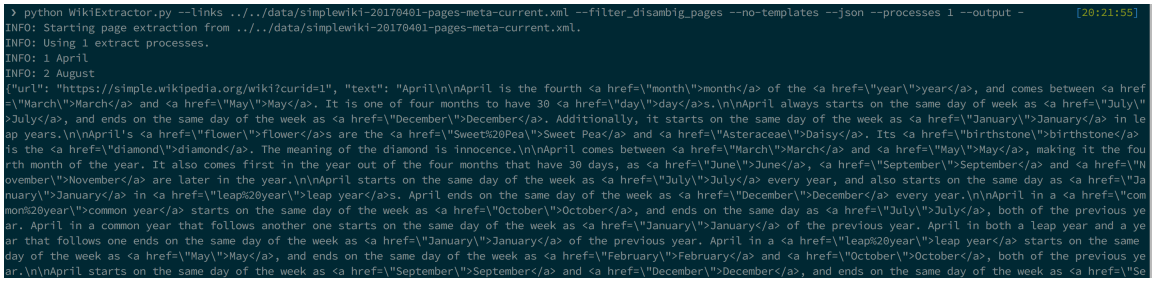

Figure 26: Running WikiExtractor to get the page data as JSON to pass to our inverted indexer.

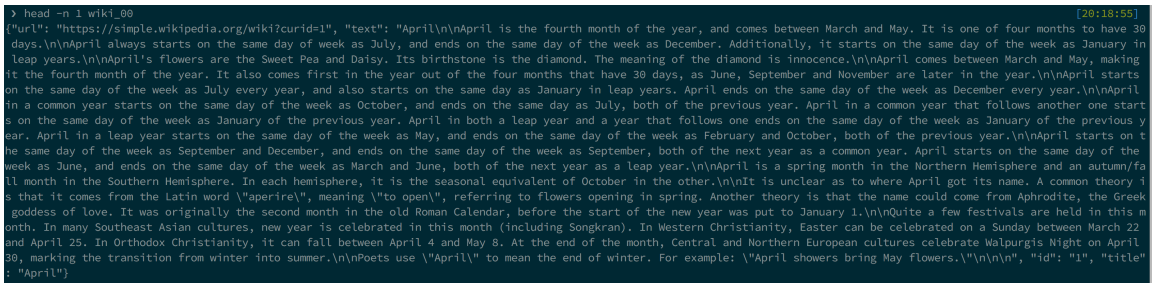

Figure 27: Sample output from Wikiextractor

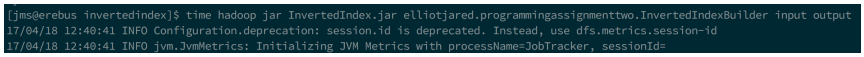

Figure 28: Now we're running the inverted index

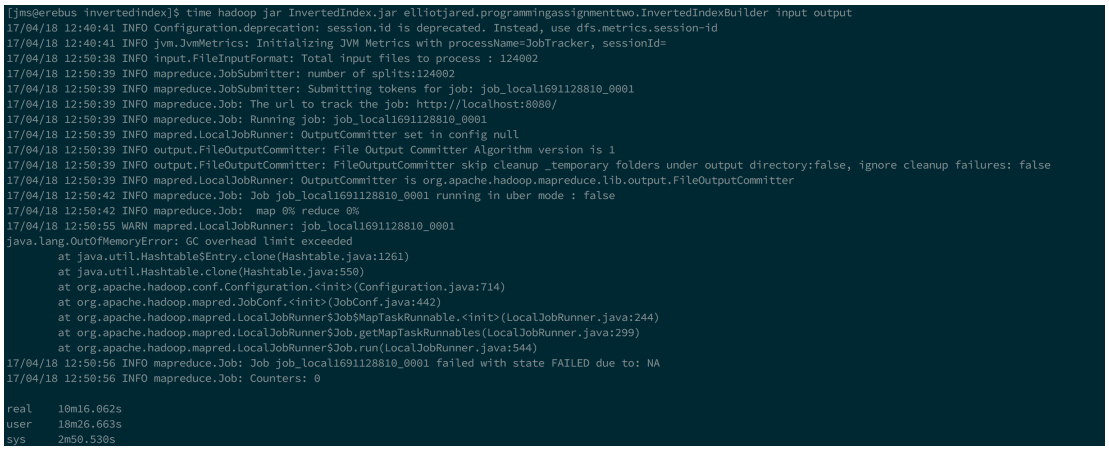

Figure 29: However, sometimes we had not enough memory issues for the inverted index. We increased the memory by raising the maximum heap size for the JVM and it took off fine from there.

Figure 30: Here is the job in the middle of it running.

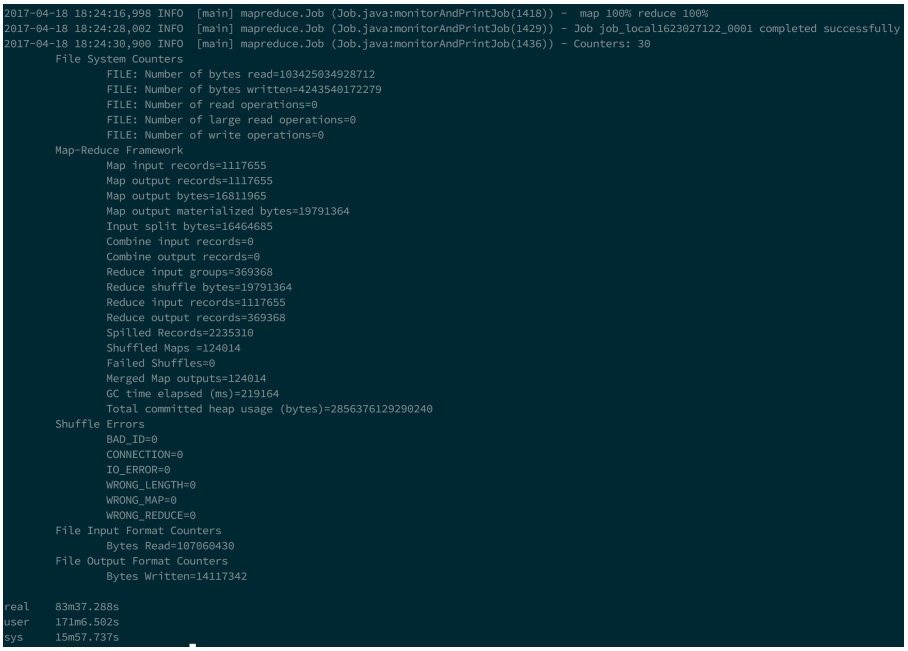

Figure 31: Finally, it's completed in 83 minutes.

3.4 Extra Credit: Web UI and Custom Ranking

We also ran a few basic queries through our web portal for the extra credit portion to show functionality. Here are those results.

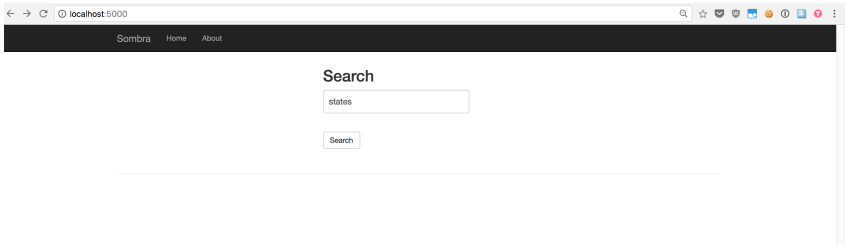

Figure 32: Here is the main search page, which is borrowed from the extra credit for Programming Assignment 2.

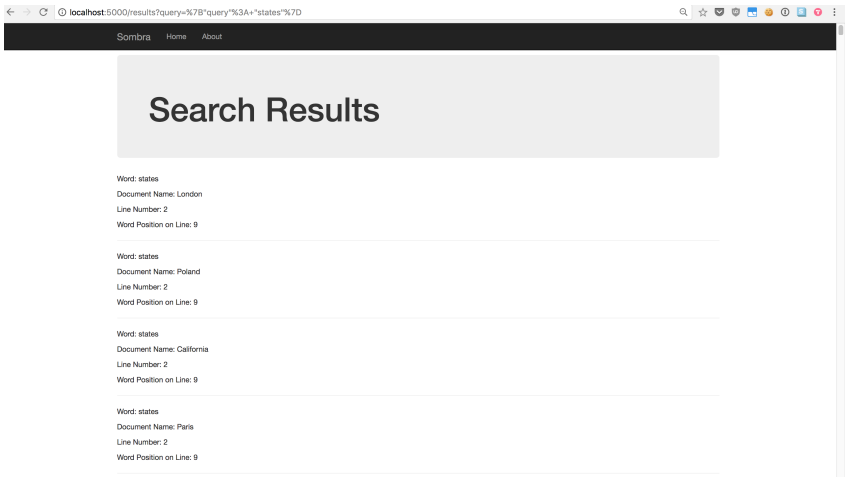

Figure 33: Here is the search results page for the query "state", showing the highest ranking results at the top based on the pagerank and the underlying results based on the inverted index.

4 Test Results

As part of our testing process, we ran both the PageRank algorithm by itself, and the querying and webpage functionality as a while. For the PageRank test, we produced a sorted list of Simple English Wikipedia pages based on their PageRank values in descending order on three virtual machine instances. This test took 176 seconds to converge on average across 25 runs. The top results for one iteration are show below, and one set of the full results are available at /results/pagerank-176-seconds-15. This ranked list makes perfect sense for our dataset. An English speaking country and Wikipedia creator, the United States, tops the list, followed up by other countries. All of these would be linked to in many cultural references across simple pages.

```
United_States: 818.2641259896291
Multimedia: 816.6600942196424
France: 395.8886873670224
Definition: 338.43461234428815
English_language: 319.8528823227395
United_Kingdom: 313.98604609382176
International_Standard_Book_Number: 284.0803868582328
Country: 250.93384509869472
Geographic_coordinate_system: 234.02814100972452
England: 233.6676732723399
Japan: 231.42446809588768
Association_football: 230.94232313662533
Wikimedia_Commons: 229.64370286147533
Germany: 216.65665828527426
Europe: 214.7473503397607
```

Figure 34: Top 15 results of PageRank with article title and value.

5 Challenges

We faced several challenges while building Sombra.

- The largest challenge for this project was the switch to using Amazon Web Services. The benefits of this system are its high effectiveness once setup, and commonality in the industry. Most individuals in the technology industry has, does, or will use AWS at their company. However, setup for AWS is challenging, usually requiring multiple extensive courses to master.
- On the inverted index side, we struggled with garbage collection and available memory, especially with the small file sizes used in this project, which required many configuration changes. Hadoop works well on larger file sizes, but not the smaller files sizes. This was eventually remedied by upping the JVM memory limit on the heap for the garbage collector.
- One challenge we overcame quickly was the preprocessing steps for the inverted index. Wikiextractor, an open source technology, really saved the day with parsing and sanitization of the Wikipedia markup, something that usually would have taken a large amount of time to implement ourselves.

6 Conclusion

In summary, we built Sombra, a web portal capable of querying results a ranked version of the Simple English Wikipedia dataset. To support this web portal, we implemented a scalable Spark PageRank algorithm, and incorporated our earlier work on Winston, which ran an inverted index calculator. This allowed for queries to return word location by page, ranked by importance using the PageRank algorithm. Our implementation was successful, allowing for quick researching of Wikipedia entries in Simple English, a system that would be useful for anyone non fluent in the English language.