FlyingMantis: an Elastic, Modularized and Fast-deployed New Generation Search Engine

Bill He
lihe@seas.upenn.edu

Leshang Chen
leshangc@seas.upenn.edu

Mikael Mantis
mantism@seas.upenn.edu

Steven Hwang
stevenhw@seas.upenn.edu

1. INTRODUCTION

We draw inspiration from the Google paper in implementing our search engine. Our aim is to create a product that could parallel Googles search in performance and appearance. With our implementation, we hope to allow users to make realistic searches to retrieve not only realistic results, but also other useful information such as e-commerce hits, and weather information etc. Key aspects of our system which we would like to focus on are our approaches to development of a system which sought to be efficient in storing and providing information to service search queries.

2. HIGH LEVEL APPROACH

In order to construct an efficient search engine, our optimizations focused on distributing as much of the processing as possible. It was intuitive to distributed the crawler in order to fetch a large number of documents in a timely manner given that the robot.txt files of many websites required delays between queries to their hostname. Indexing the documents by their contents and relevance naturally fit into a streaming Map Reduce-like problem that we felt could solve with a high level of parallel processing on a per-file granularity and storing our indices in Oracle RDS was natural as RDS had a good reputation for being fast efficient in querying and manipulating tables as well as allowing for high levels of per-row based parallelism (given that all tuples outputed by the Indexer were unique on the primary key of word and document ID). In our efforts we took advantage of many of Amazon Web service's tools and resources such as EC2, S3Distcp, HDFS, Elastic MapReduce and RDS to implement our solutions. Using these resources, we found that we could develop a relatively scalable processing pipeline to make search results available and relevant. Furthermore, we took some liberties to explore concepts beyond the scope of web document processing in order to include more related content in UI by tapping into Amazons API.

2.1 Division of Labor

- Foundation (HW1): Stevens HW1 Server
- Distributed Crawler: Bill
- Indexer/TF-IDF Retrieval: Steven
- PageRank: Leshang
- Search Engine/UI: Mikael

2.2 Project Architecture

- Distributed Web Crawler
  - Master Node (1 t2.xlarge EC2 Instance)
    - Maintains the frontier queue
    - Respects the robots.txt delays and restrictions
    - Determines unseen urls
    - Sends urls for Slave Nodes
  - Slave Nodes (StormLight Structure: 7 t2.micro EC2 Instances)
    - Spout that receives urls from the Master Node
    - Bolt that crawls the current url and retrieves the document
    - Bolt that uploads the document and parses/stores the relevant outgoing links (locally)
    - Bolt that sends a response back to the Master
  - Other Nodes
    - Node that dynamically injects urls into a queue residing on the Master for it to consider
    - Node that handles the robots.txt retrieval
- Spark Indexer
  - The indexer was hosted on AWS Elastic Map Reduce
  - Elastic MapReduce configuration
    - 1 Master Node - m4.4xlarge (16 vCpu, 64 Gb RAM)
    - 6 Slave Nodes - m3.xlarge
  - Elastic MapReduce internal Hadoop FileSystem
    - Stored Web document Corpus on EMR Cluster HDFS for fast Elastic Map Reduce processing
  - Spark Configuration

    - Run in Client Mode (See Appendix Figure 1)
    - multi-core allocation to a single Executor on each node with the maximum amount of RAM allocated to processing
  - RDS Database
    - Inverted Index table (Columns: Word, DocID, TFIDF Score)

- Spark PageRanker
  - Run Page Rank Algorithm on EMR for deploy or locally for debug

- Web Search Interface.
  - Search Engine Servlet run on HW1 Servlet container and HTTP server hosted on EC2 instance.

3. CRAWLER

The crawler leverages code from HW2 as a skeleton. As a result, it followed the StormLite structure, with slight modifications at each step of the topography. We seeded our crawler with 250 hand-picked seed websites to be the starting point of the corpus. In choosing seed urls, we selected sites whose robots.txt file would yield sitemaps that build the frontier queue significantly. Special consideration was given not to truncate the sitemaps and the links given, because they point to the most relevant resources. However, regular pages were not afforded this flexibility. We limited the number of outgoing urls that went into the frontier queue to 40 and the number of urls per hostnames to 200 allocated at random. Our crawler adopted the Mercator model, so realistically, we didnt have the infrastructure to maintain or crawl a significantly more number of urls. The start our processing stream was the Spout bolt which got the urls from the frontier queue in a Mercator manner for the crawler bolt while also respecting the robots.txt restrictions as it retrieved the html (or xml) pages. The next bolt extracted the useful outgoing urls and uploaded the retrieved documents onto S3. This Bolt also saved a copy of the outgoing links locally on a text file. The final bolt decided which urls are going back into the frontier queue.

3.1 Distributed Approach

When we began distribution on EC2, we abstracted tasks into the Master and the Slaves. The Master maintained the states (i.e. the frontier queue, the set of urls that have already been crawled, and the robots.txt restrictions). Whereas, the Slaves state-lessly retrieved, uploaded, and parsed the documents and saved the outgoing links on a text file locally. The outgoing links extracted are communicated back to the Master to handle. The Master ran on a t2.xlarge EC2 Instance (2 vCPUs and 8 GiBs of memory). We started smaller instances, but it was quickly obvious that it was simply not enough to sustain our operations. The Slave nodes ran on t2.micro EC2 Instances (1 vCPUs and 1 GiBs of memory). Because of the stateless nature of the slave nodes, we were afforded the flexibility to add and remove instances without affecting the entire operation. We made sure to save the Masters state and upload each slaves repository of outgoing links periodically onto S3 buckets.

3.2 Extra Considerations

To make the crawling operation scalable, sustainable, and efficient, we considered a few optimizations. We started to implement bloom filters to make the infrastructure space-efficient, but it was empirically determined that there wasnt a significant difference, so the idea was abandoned. However, we manually hashed the hostnames into separate buckets, before maintaining a hashset of the frontier queue as well as the crawled urls. This allowed us to limit the size of the set of urls per hostname (for reason mentioned before) as well as offer quicker access to urls.

The documents were stored as objects on different S3 buckets with number identifiers ("DocIds") instead of urls. This made accessing the documents easier for the Indexer which is leveraging the Spark framework. The crawler also generated a list of number identifier to url pairs, which was constantly updated on S3. When we finished crawled, we were maintaining a corpus size of 213,673 documents.

4. INDEXER

The Indexer's purpose is to extract content from the corpus's webpage information and provide a score for every word in the document to represent its overall relevance in the document as well as representing the uniqueness of the document in the corpus. The score used was the TF-IDF score of words in the document, where the TF portion of our version of the score provided information about the relative normalized frequency of the word in the document and the IDF portion provided information about how important the word is to the document. The result of indexing is a set of triples containing a word in a document, a document id and the word's TFIDF score.

4.1 Useful Indices produced

Three Indices are produced in order to assist with servicing search queries: DocID to URL Index, Inverted Index, Combined Score Index. The DocID to URL Index mainly provides the Inverted index an optimization on the grouping operations in RDS when joining the TFIDF table and the PageRank table on the DocId in order to produce the Combined Score Index which is mainly used in processing search queries. The Combined Score index essentially provided a unified table for the relevant TFIDF score and PageRank weight such this processing did not have to be done at the point of procesing a receieved search query. The Inverted Index provides a way to find which Documents contain the search query terms and therefore determine which pages to consider ranking and returning to the client. The Inverted Index is computed in two phases: the first phase involves computing the TF scores of each word (See Formula 1) and the second phase involves computing the TF-IDF score (See Formula 2) though it should be noted that the second stage could only start after the first phase is completed. The Combined Score Index is computed in the final stages of the Indexer and PageRank processing phases are over and a SQL query is used to populate it as a SQL table (see Figure 3).

$$TF(t,d) = \frac{f_{td}}{max\{f_{td} : f_{td} \in d\}} \quad (1)$$

where f_{td} refers to the word frequency of term t in document d.

```
INSERT INTO TFIDF (word, docid, TFIDFSCORE)
    SELECT TF.word, TF.docid, ( TF.tfscore * LN( <corpus_size> / MyCounts.total )) AS TFIDF
    FROM TF,
        ( SELECT word, COUNT(docid) AS total
          FROM TF
          GROUP BY TF.word
          ) MyCounts
    WHERE TF.word = MyCounts.word;
```

Figure 1: TF-IDF Computation SQL expression.

$$IDF(t, D) = \log \frac{N}{|\{d \in D, t \in d\}|} \quad (2)$$

where N refers to the number of documents in corpus D, t refers to a term in document d.

4.2 Indexer Implementation

The Indexer is written in spark to process all the web pages that were stored in hdfs. The Web pages are processed in three stages: 1) Read in files as Pair of filename and text content, 2) Apply a Map function to process the contents of each document and output a list of triples 3) Collect all the results and store them in a .csv file which we manually upload to AWS RDS Database. In the first stage, the corpus is divided into 40 partitions and the spark application will apply the two later stages on each partition iteratively. The RDD produced by this stage stored tuples of two elements (first element: filename String i.e. "file:/../../120, second element: whole document text as a String). The partitioning scheme was necessary for scalability as the spark executor processes coudld process these partitions in parallel and we passing the results of a transformation around, it was recommended to keep the sets of transformed tuples fairly small. It should be noted, that the decision to have 40 paritions was the result of testing performance and balancing the amount of RDDs shuffle around without overloading the memory on each Slave node. In our case, 40 partitions had fairly good performance and the concern about memory was easily resolved by upgrading the Slave node instances to the next level given that our scenarios did not involve too large of a dataset. The second stage, involved parsing the document text in order to extract the words in the document body. We used Jsoup to extract the body of the html document and split the body by a space. If Jsoup could not find a body in the document, we would just filter this document from the index; we deemed this document not worth searching. After splitting the body, we used a regex to only include strings with alphanumeric characters. If it passed the alphanumeric regex, we also check if the string only contained numeric characters and filtered number-only strings because we only wanted alphabetic strings in the inverted index. Next, the strings were down-cased and stemmed using a porter-stemmer and stored in a hashmap where the key was the stemmed word and the value was the count of the stemmed word in the document. After the words in the document are stored in the hashmap, the TF scores of each word are computed by dividing the word count by the maximum word count in the document (see Formula 1). The RDD produced from this stage contained a Tuple of three elements (first element: stemmed word, second element: DocID (i.e. the filename is the DocID), third element: tfidf score). Finally, in the third stage, after the csv file of the results is created and uploaded to

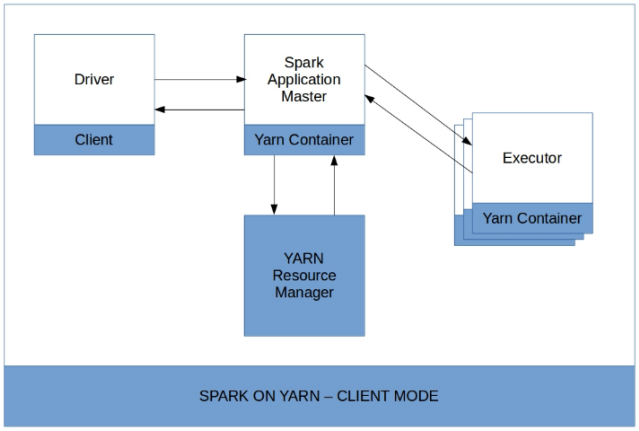

Figure 2: Spark Client Mode Configuration.

S3, we manually update the RDS database using the Import functionality in SQL developer.

After uploading the document tuples onto RDS, the IDF scores are computing for each word and the TFIDF Score column in the TFIDF table is updated to contain the TFIDF scores for each row (see Formula 2). The TFIDF score is computed by multiplying the TF score with the inverse log of the corpus size divided by the number of documents that contain the word (note: the corpus size is determined by an prior query using Select distinct on doc id in the TFIDF table). The main purpose of combining the TF and IDF scores was to determine how well a word characterizes a document by how much information about a document the word provides. Specifically, a highly informative word would have a high TF score (indicating it is probably an idea/topic emphasized in the document) and low IDF score (indicating that the word is a defining aspect of the document). The result of the TFIDF is then store in a TFIDF table and the TF table is discarded to save space in the database. In order to take advantage of optimizations in the RDS and data locality, we used an SQL query (see Appendix 3) to generate the results and store them in the TFIDF table.

4.3 RDS Storage System Based Index

The estimated size of the TFIDF table was 3-5Gb (output generated for our corpus was resulted in $> 50,000,000$ rows where each row in the TFIDF table consisted of 20-byte word, 4-byte number, 8-byte float).

4.4 Elastic MapReduce Configuration Details

The Spark Application was run in AWS Elastic MapReduce in Client Mode with one Master Node and 6 Slave Nodes. In Client Mode, the client is an external machine submitting a job to the Master node which coordinates the distribution of tasks to the Slave Node (see Figure 2). The cluster was given several steps to define the Indexer processing pipeline. Initially, we transfer from several S3 buckets onto the local EMR Hadoop filesystem using a built in EMR function called S3Distcp to have faster access to the files in the cluster, given that reading the files directly from S3 in the distributed fashion had some rate limitations in the amount of files we could read into the system per second. We noticed that reading files in from hdfs, significantly outperformed the method of reading the files directly from

S3. The run configurations we provided for setting memory allocation and cores for the Spark Driver and Spark Executors are as follows: Master node running on m4.4xlarge (16 vCpu, 64 Gb RAM) and the Driver process running on it was given 3 cores to use and 10 Gb RAM; Slaves nodes were running on m3.xlarge (4 vCpu, 12 Gb RAM) and were given 3 cores to use and 10 Gb RAM. The total number of Spark Executors was 6 such that there would be one executor per Slave node.

5. PAGERANK

PageRank is an algorithm for rating Web pages objectively and mechanically, effectively measuring the human interest and attention devoted to them. The idea is from an naive thought the link refers the direction of interest where the attention is paid from current page to the linked page. The page rank can be used to determine whether a page is authoritative and it can also be used for other purposes like sorting based on backlink numbers. The idea was originally proposed by Googles founder, Larry Page, and thus given Page's name.

5.1 PageRank Algorithm

Pages and links on the Internet are considered as elements in a directed graph. The calculation is basically iteratively calculating the weighted average for linked pages until convergence. The formula can be written as:

$$R(u) = \sum_{v \in B_u} \frac{R(v)}{N_v} \tag{3}$$

where R refers the rank and N_v is the number of links from u.

5.2 Special Cases and Treatment

Notice that this project is a naive realization for a general search engine and has only a small amount of data, the crawler is thus somewhat naive in the way it retreives data; at the point where we stop crawling there exist pages that we have not crawled yet but exist in our frontier queue. This fact causes a problem and must be treated separately. The links that don't come back to the cluster, which Page calls "dangling links"; need to be removed in the calculation. The intuition here is that they won't affect the result as they do not contribute to weights of other pages as they are link loops. Thus they are ignored and calculated after the core graph weight has converged. Another way of dealing with self loops is to delete them, but this will not affect the result too much if many pages are crawled.

For the sinks constructed like a loop, or the self loop as a whole, it is not sensible to ignore all of them. Instead, Page has a modification on calculation formula where he add a source for every node to counterbalance the loss of weight:

$$R(u) = \alpha \sum_{v \in B_u} \frac{R(v)}{N_v} + (1 - \alpha)E(u) \tag{4}$$

For iteration stop point, a small positive threshold ϵ can be settled to determine whether the weight of one page is converged. If each of the differences between iterations is below the threshold, it can be inferred that the pagerank comes to an end.

Algorithm 1 PageRank

Input: orgURL x_i, destURL y_i
Initialize $converged = false.$
$danglingURL = destURL - orgURL$
$legalNodes = (orgURL - danglingURL) \wedge links$
while $differenceSum > 0$ **do**
  Map links to (orgURL, (destURL with weights))
  Reduce calculate (URL, weights)
  Count weights dif for each node based on threshold
end while
Recompute weights for dangling links
Collect weights and emit

5.3 Implementation Details on Spark

Our implementation of PageRank is based on MapReduce framework and utilizes Apache Spark as development platform. The mechanism follows: firstly the page URLs are provided by the crawler and stored on Amazon S3. The program will fetch those things in one bucket and process it by algorithm iteratively. Finally it will collect things together and upload the result to RDS. The simply gives the structure.

The detailed functions are given in algorithm table 1. The key problems are: dealing with dangling links, graph sinks, and setting up proper intermediate data throughout iterations. Based on spark, it is very easy to perform operations like getkey, getvalue, distinct, subtract, join and filter. Comparing Hadoop or Storm, we have to manually write every Map and Reduce function and that is not so convenient. As a result, we can use JavaPairRDD as the abstract form of data and use them as intermediate data for iterations to go on. To get rid of the dangling link, simply do a subtract from destination nodes to origin nodes to get the dangling pages. After that do a join between links and dangling pages on the key (or to say in another way, the origin nodes). This will give us the set of links that composed of nodes with both forward links and backward links. Using these core links and applying PageRank algorithm, we can do iterations to calculate the weight for these pages. The way to calculate the difference is to use the subtract function and use filter with threshold to count the total number of nodes that are not converged. With all core nodes converged to a set of weights, calculate the weights for the rest of nodes by applying average weighted index for another time.

The program runs on Amazon Elastic MapReduce platform. Notice that we exchange the data by S3 and RDS. This means that we don't have to consider the working status of other parts. As soon as the crawler finishes, it can start the PageRank instance automatically or by hand. The code would not interfere and can be fully serialized.

5.4 Associate PageRank with Other Components

We may notice that the result of PageRank should be used by search engine along with the TFIDF score generated by the indexer. Since the RDS also has some limitation on special characters, we should take care of the matching between the URL strings. A better method to do this is to use document IDs or hash names instead of simply using string as the component of the links.

With the matching guaranteed, we can use relational operations provided by RDS to do matching and sorting and

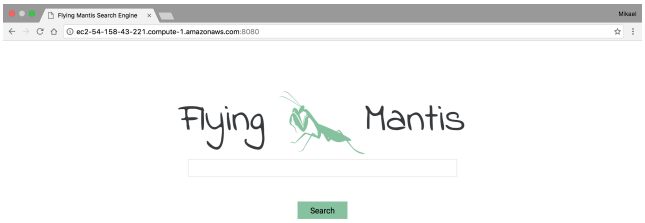

Figure 3: Home page

thus reduce the query time for search engine.

6. SEARCH ENGINE AND WEB USER INTERFACE

For every application, usability is one of the key factors in measuring its quality. Thus we wanted our search interface to be simple and seamless while also delivering results that are as meaningful as possible. Using a Java Servlet, a series of sql queries to our AWS RDS database, AJAX, and Amazon's RESTful Product Advertising API we created a search engine that handles concurrent requests, delivers data asynchronously, and displays relevant results from our index.

6.1 User Interface

The main goal of the user interface was to mimic Google's minimalist theme. This involved using HTML, CSS, and JavaScript (AJAX/JQuery) to display our logo[1] along with a form to perform search queries. Using AJAX, we added a listener on the searchbar so that upon receiving input a post request would be sent to our servlet and the body would be updated asynchronously before a search is made so that the search bar is moved to the top of the screen. Then another AJAX request is made when a user submits their query to add the results underneath the search bar that has already been moved. By updating the html dynamically we are able to make the process of searching more visually appealing as well as more accurate to Google's functionality. The results that are added to the body of the page include the number of results found, the time in milliseconds it took to find these results, the first 10 links from the result set, and the first three results from a call to Amazon's Product Advertising API.

6.2 Search Algorithm

Our process of making a search upon our index involved two major steps. The first, was precomputing the result of intersecting the TFIDF and PageRank tables in our database and taking the harmonic means of each matching tf-idf and

[1]logo created by parkjisun from the Noun Project, https://thenounproject.com/search/?q=mantis&i=205407

```
insert into results
      select word, docid, url, (2 * (weight * tfidfscore) / (weight + tfidfscore)) as partial_rank
      from (
          (
                select tf.word, tf.docid, tf.tfidfscore
                from TFIDF tf
                inner join QUERYTABLE q
                on q.word = tf.word
          )
          inner join
          (
                select d.id, p.url, p.weight
                from PAGERANK p
                inner join DOCID d
                on p.url = d.url
          )
          on docid = id
      );
```

Figure 4: Intermediate Results SQL Expression

page rank score on url. The harmonic mean [2] of two numbers is defined to be

$$H = \frac{2x_1x_2}{x_1 + x_2}$$

the purpose of using the harmonic mean is so that we can mitigate the impact that large scores would have on our search results. The second part of our search involves making a database query when a user submits a keyword search. Upon submission the query is split by spaces into its separate keywords. Then each word is stemmed using a porter-stemmer implementation so that the word can match the stemmed words on the precomputed results table. From there, a sql query is made to group by, sum, and order by the results for each webpage that is found. Finally, the first 10 results become the result set that is displayed on our page. Using jsoup and our stored data on s3 we get the metadata of each result so that we can display the title and description of each result.

6.3 Amazon Product Advertising API

To augment our results we used Amazon's Product Advertising API to display their top three results matching the users query. This involved sending a REST request to their API's endpoint and then displaying the necessary results in html format. A slight drawback from using Amazon's API was that they throttled the number of requests per API key to 1 request per second. This made it particularly difficult to run concurrent requests since the API would throw a 503 response code and we would have to send the request again to get the desired results. Thus when testing with Apache Bench we would often disable the use of the API but with our final tests we included the API. Upon further testing we realized this throttling was negligible for search speeds.

7. EVALUATION

7.1 Crawler

The StormLite structure didn't give the crawler the flexibility that it needed. As a course sandbox it is very neatly formatted and we didn't run any problems in the beginning. However, as we began to crawl for documents from bad hosts, ones that were either outdated or missing information, we discovered that the system would necessarily stall at different points (either waiting indefinitely for more data, or fall into traps). This forced us to adapt a structure to maintain persistence when these stalls happen. However, we quickly discovered that it was still a problem because of

[2]Source for harmonic mean:

the operations to store the large number of crawled urls and the associated information. At this point, we were only averaging 1,000 urls per hour crawled and stored. This was done on local machines. One way we tried to handle the stalling problem was by creating threads on each crawlerbolt, but we would then run into memory issues because of threads that have not been released. As a result, we developed the distributed approach mentioned in the implementation section for the crawler. When we had 2 Slave instances, we were averaging 2,000 links per hour, and this number was consistent. When we upgraded to 7 Slave instances and a more powerful Master, we got around 10,000 links per hour.

7.2 Indexer

By Observation, the Indexer was able to parse the entire corpus and produce the document word tuples on average within 30 minutes, which amounts to a rate of 7000 pages processed per minute. However, the subsequent processing of the TF-IDF scores would take 45 minutes. We also determined that a bottleneck arose when we loading data into our RDS Database. Initial implementations of the Indexer attempted to increase performance by making distributed updates to the database in batches of 5000 Inserts per update and was able to achieve 20000 rows per minute. However, given that the total number of rows in the TFIDF table is 50 million rows, this approach would have exceeded 41 hrs. In the end manually uploading the data to the database deemed to be the fastest solution. One aspect of loading data into RDS that we did overlook (as per Prof. Ives' suggestion) would be to disable transactional logging which potentially added significant overhead to updates made to the database.

7.3 PageRank

With one machine of 4 vCPUs and 4GB memory, the PageRank progarm is able to calculate weights for 200k downloaded pages and 3000k dangling nodes in less than 20 minutes. By speculating the URLs with weights, the result is proved to be legal and reasonable under intuition of importance and authoritativeness.

7.4 Search

Throughout the process of developing the search engine we evaluated our servlet's performance by using apache bench to send numerous concurrent requests. Initially, on a small corpus our servlet processed requests on average of 2.4 requests per second with an average speed of 733 milliseconds across concurrent requests. These were the results from a test of 1000 requests with a concurrency of 200 requests at a time. When we tested the same smaller corpus with 10,000 requests with a concurrency of 100 the results were 1.01 requests per second and an average of 1018 milliseconds across concurrent requests. However, once we ran our server with access to the full database we noticed a huge drop in speed such that running apache bench would have been useless. Because our servlet would output the time it took to complete a search we could see the time per search and it ranged from 20-60 seconds depending on the size of our result set. At first we thought that the use of Amazon's API was throttling our speed but after disabling the servlet's access to the database we saw that the API's throttling was negligible. Upon further examination we realized that by processing each row in the result set to then output only the top 10 results was a very naive approach. One approach we might consider given more time would be to maintain an instance of the result set class and simply retrieve the top 10 pages when desired rather than retrieving all the pages at once. Another approach, assuming we only care about the first 10 results, would be to only retrieve those 10 then make another query to just count the number of rows from the final results table so that we can present the size of the result set to the end user.

8. CONCLUSIONS

We witnessed the benefits of distribution in many aspects of the implementation of our systems' components, in both the Crawler, Indexer and PageRank we were able to achieve significant throughput given the safeguards and fault tolerance put in place. Furthermore, we were exposed to many of the AWS technologies and were forced to consider throughput limitations and possible alternatives. In dealing with significant bottlenecks in S3 and RDS, we also gained insight into how to approach operating these systems in future contexts where we deal with increasingly larger datasets.

9. ACKNOWLEDGMENTS

We would like to thank Prof. Ives for teaching this semester and guiding us through the concepts in Web Systems and scalability. We all felt that we learned a lot of practical knowledge and skills from this course!