FastAndFurious 1.0

by
Aakriti Singla
Krishna Chaitanya Daliparthy
Mainak Datta
Parul Bhalla

University of Pennsylvania, PA

Table of Contents

1. Introduction

1.1. Project Goals

The objective was to build a web search engine (implementation of indexer/crawler/pagerank-er) which is an academic clone of Google Search Engine. While Google presented it as a prototype in the year 1998, there have been many changes to the implementation of The Anatomy of a Large-Scale Hypertextual Web Search Engine over time. We, here , present the search engine to incorporate the latest methodologies favoring performance and scalability.

1.2. High-level Approach

Our design approach involved the following components working together :

a. Mercator Style Crawler to parse HTML documents and crawl a decent portion of web.
b. Indexer to index the web pages crawled in (a) by content (various html tags and tf , df, tf-idf values and positions of occurrences)
c. Run Pagerank algorithm on the output of web graph from (a) using MapReduce job to rank each page
d. An interface for the users to search for the key term(s) to view results ranked on the basis of various inputs from indexer and pagerank based on the ranking algorithm which is highly optimized.

1.3. Division of Labor

All members of the team have taken up the responsibility of creating and managing the infrastructure on which their component was running. While we worked on any component that required attention, the below mentioned are the primary responsibilities taken up by the team.

Team Member	Primary Responsibility
Aakriti Singla	Web Crawler
Parul Bhalla	Indexer
Krishna Chaitanya Daliparthy	Search Engine
Mainak Datta	Page Rank

2. Project Architecture

2.1 Introduction

The whole project was mainly divided into 4 major components which are outlined as follows:

1. Crawler
2. Indexer
3. Page Rank
4. Search Engine

2.2. Crawler

Crawler was a modified version of Mercator with design choice to create a robust and scalable system. URL Frontier was implemented in form of a on-disk queue which was stored in Berkeley DB and the start and end pointer was maintained in order to perform enqueue/dequeue operations, thus removing memory constraints and to allow the application to be scalable. A distributed architecture was implemented wherein we had a total of 16 EC2 (medium) instances (1 Master and 15 Workers) for crawling the web. The master was responsible for co-ordination and communication between the various workers. The workers were multi-threaded with the thread pool of size 20 to crawl and map the extracted URLs via SHA-1 algorithm (hashing based on hostnames) to respective workers. The Map-Reduce framework from Homework3 was tweaked to implement the same.
All possible exception-handling was done to ensure the crawler runs without haulting to crawl 100,000 pages.
The crawler architecture was supported by basic implementation of Master-Servlet which displayed the ip:port , status and number of files crawled. Proper error-logging was done. A Berkley DB table was maintained to store seen-urls, along with tables for Hash-ID to Web-graph mapping (for Pagerank), Hash-ID to Document Url mapping (to have a primary key which was not as huge as urls), Hash-ID to content mapping (data, type, length, last-crawled) and a table to store Robots.txt for each crawled domain to ensure politeness.
Crawler also included a module to transfer data from Berkeley DB to Amazon S3 in the required format (input format required by Indexer and Pagerank-er)

2.3. Indexer

Indexer was another important component of the search engine. The goal was to

create an Inverted Index to make the word lookup faster. The idea was to parse the entire HTML crawled content and associate a set of features with each word, which will help us quickly locate the documents in which the words can be found. The set of features that we decided were such that the resulting set of documents that are returned by the search engine corresponding to the given word are as relevant as possible. Thus each word in my indexer, is associated with a list of Document-Ids(which is nothing but the hash of the url since it was easier to store the document id instead of the complete url everywhere which was very long in length), that is a list of all the documents in which that word appears. In addition to that each word is also associated with the list of positions at which the word was located in the document. Amongst the other features that were used were: Tag Counts, wherein we kept the track of the following tags: h1,h2,h3,h4,h5,h6,bold,italics,underline, anchor, **meta**, and title tag. We also included the normalized term frequency and the normalized tf_idf associated with each word.
The normalized tf that we calculated, we used the following formula for that:
no. of times the word appears in the document/maximum term frequency out of all words

2.4. PageRank (-er)

PageRank was an iterative MapReduce procedure where the output of the reduce stage was used as the input to the map stage of the next iteration. It was run on Elastic MapReduce, reading its input files from S3 and writing its output to data files in S3 which were copied to DynamoDB using a separate Elastic MapReduce job. The input files were generated by the crawler in a “webgraph” format: DocID currentPageRank {List of DocIDs of pages linked from this page}. The final output after multiple iterations was files of the form: DocID pageRank, which was the format used to store in DynamoDb Table.

2.5 Search Engine

The search form had a simple User Interface backed by bootstrap design, it had a searchbox (HTML5) and a checkbox to enable/disable Spelling Check.

Architecture Overview

The submitted query is first tokenized and then lemmatized using MorphaStemmer. The output of lemmatization is a list of words.

The Search Engine is a ThreadPool backed application that has a pool capacity of 20 threads (The number tweaked and finalized according to the response time). Each of the waiting thread is given a word from the list mentioned above.

The worker then connects to DynamoDB instance with table name doc_idFeatures to fetch the corresponding features vector.

3. Design Choices, Data Structures, and Implementation

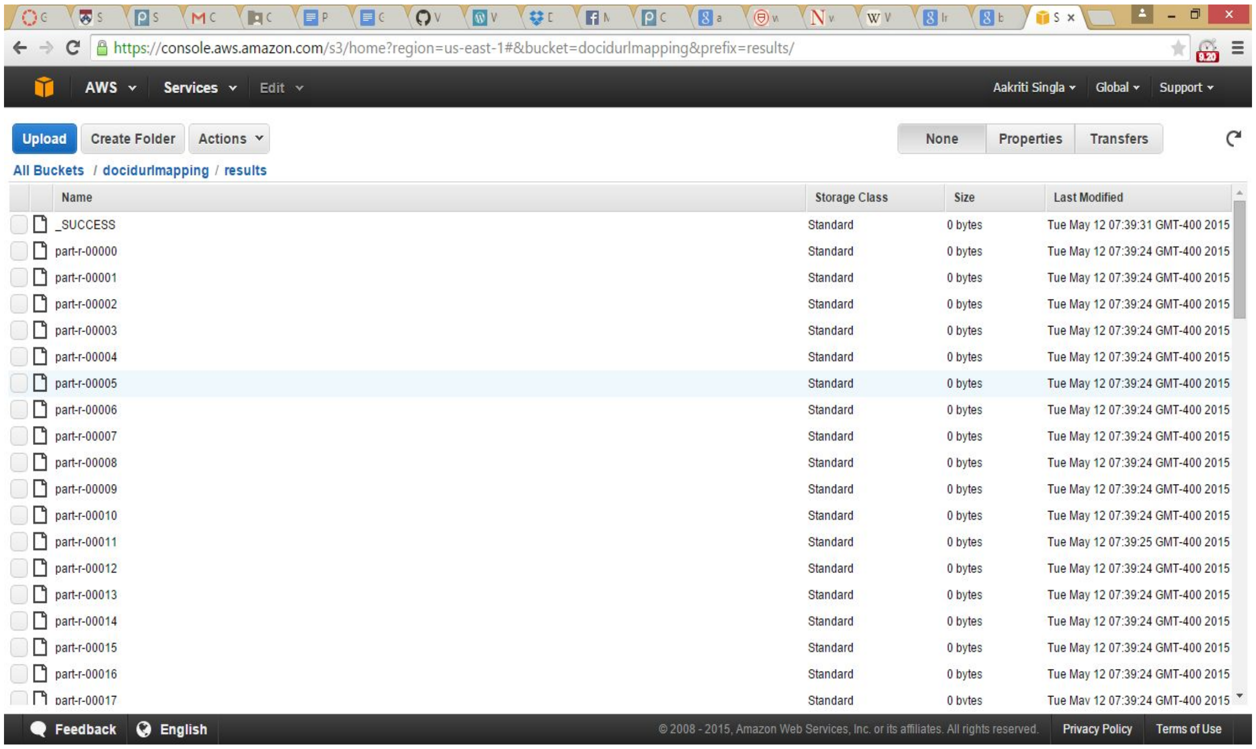
3.1. Crawler

As a modified version of Mercator Architecture , a tweak of MapReduce framework was used. The Workers constitute of a thread pool and accepts crawling jobs one at a time from a on-disk Berkley DB queue. Upon recieving a crawling request, the Crawler Thread checks first for Robots.txt for disallow/allow urls , crawl delays etc and den send a HEAD request to ensure validity of page and then crawls the entire page. The head request checks for the content-type , language restriction ('en'), handle various 3XX response status and server 2XX range response codes. The content of the pages is stored in Berkeley DB along with the Hash-ID of the crawled URL which is later transferred to S3 as it will be an input for Indexer. Along with this a Hash-ID URL mapping is maintained in S3 and later moved to DynamoDB to be used by the Search Engine to display results. Page Downloading and URL normalizations are done well-versed to handle all the cases and get the best possible results and to handle information extracted from each crawled result. The crawled URLs outbound links were stored in Berkeley first and then moved to S3 in a certain format as per the requirements of Pagerank.
The extraction of job assignment for each crawler is a simple dequeue operation performed on a on-disk Berkeley DB which meets the Mercator politeness requirements which are fetched from another Berkeley DB. The objective of storing the Robots.txt on Berkeley DB was to allow persistence, avoid re-fetch operations and also parsing required on the file. Storing a page involves storing the data , last-crawled time and Hash-IDs. Duplicate crawls are avoided by checking another Berkeley DB table that maintains the list of all crawled urls till date.

3.2. Indexer

The inverted index was built using the Map Reduce Job on EMR. We used 3 large instances for performing the Mp Reduce operations.The following tables on S3 were referred while making this inverted index:

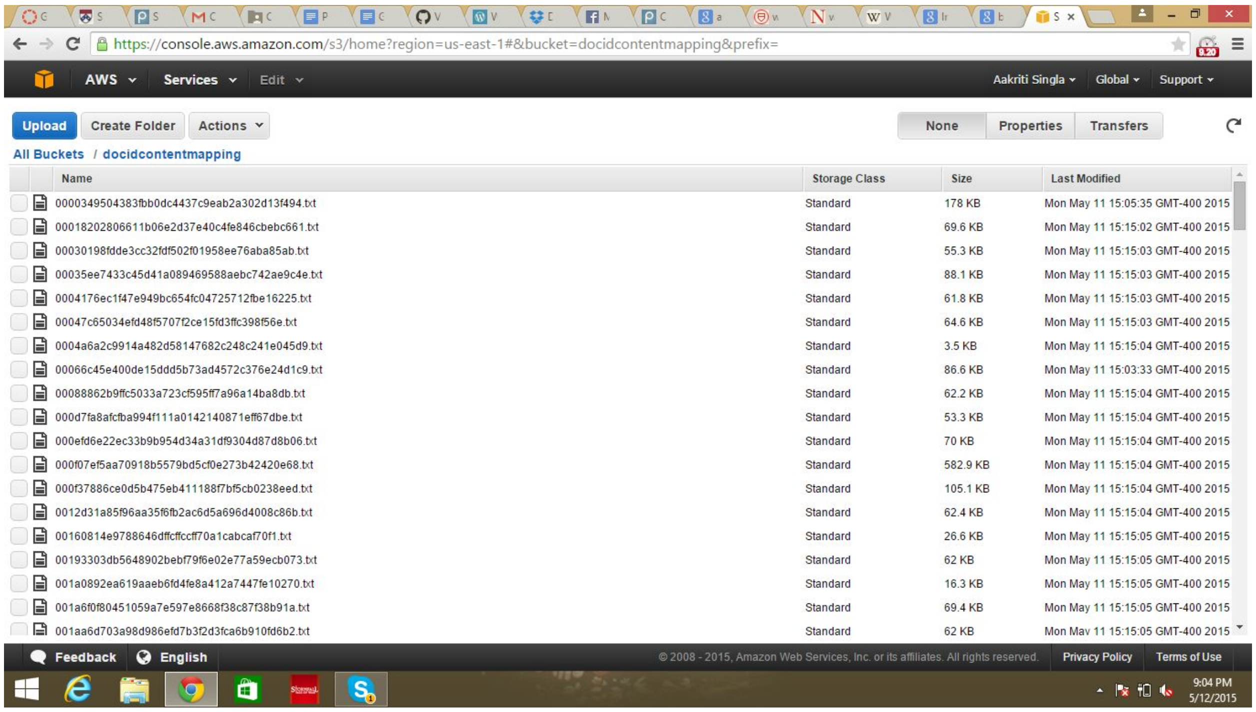
docidurlmapping: It stored the mapping from the document ids to the url, that is each url crawled was represented uniquely by the document-isd which was simply the hash of this url since it was easy to store as compared to the complete url.


https://console.aws.amazon.com/s3/home?region=us-east-1#&bucket=docidurlmapping&prefix=results/

All Buckets / docidurlmapping / results

Name	Storage Class	Size	Last Modified
_SUCCESS	Standard	0 bytes	Tue May 12 07:39:31 GMT-400 2015
part-r-00000	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00001	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00002	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00003	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00004	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00005	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00006	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00007	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00008	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00009	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00010	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00011	Standard	0 bytes	Tue May 12 07:39:25 GMT-400 2015
part-r-00012	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00013	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00014	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00015	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00016	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015
part-r-00017	Standard	0 bytes	Tue May 12 07:39:24 GMT-400 2015

docidcontentmapping: It stored the mapping of my document id and the crawled HTML content corresponding to that document. Thus each document id uniquely represents the crawled content.


https://console.aws.amazon.com/s3/home?region=us-east-1#&bucket=docidcontentmapping&prefix=

All Buckets / docidcontentmapping

Name	Storage Class	Size	Last Modified
0000349504383fbb0dc4437c9eab2a302d13f494.txt	Standard	178 KB	Mon May 11 15:05:35 GMT-400 2015
0001820280661 1b06e2d37e40c4fe846cbebc661.txt	Standard	69.6 KB	Mon May 11 15:15:02 GMT-400 2015
00030198fdde3cc32fdf502f01958ee76aba85ab.txt	Standard	55.3 KB	Mon May 11 15:15:03 GMT-400 2015
00035ee7433c45d41a089469588aebc742ae9c4e.txt	Standard	88.1 KB	Mon May 11 15:15:03 GMT-400 2015
0004176ec1f47e949bc654fc04725712fbe16225.txt	Standard	61.8 KB	Mon May 11 15:15:03 GMT-400 2015
00047c65034efd48f5707f2ce15fd3ffc398f56e.txt	Standard	64.6 KB	Mon May 11 15:15:03 GMT-400 2015
0004a6a2c9914a482d58147682c248c241e045d9.txt	Standard	3.5 KB	Mon May 11 15:15:04 GMT-400 2015
00066c45e400de15ddd5b73ad4572c376e24d1c9.txt	Standard	86.6 KB	Mon May 11 15:03:33 GMT-400 2015
00088862b9ffc5033a723cf595ff7a96a14ba8db.txt	Standard	62.2 KB	Mon May 11 15:15:04 GMT-400 2015
000d7fa8afcfba994f111a0142140871eff67dbe.txt	Standard	53.3 KB	Mon May 11 15:15:04 GMT-400 2015
000efd6e22ec33b9b954d34a31df9304d87d8b06.txt	Standard	70 KB	Mon May 11 15:15:04 GMT-400 2015
000f07ef5aa70918b5579bd5cf0e273b42420e68.txt	Standard	582.9 KB	Mon May 11 15:15:04 GMT-400 2015
000f37886ce0d5b475eb411188f7bf5cb0238eed.txt	Standard	105.1 KB	Mon May 11 15:15:04 GMT-400 2015
0012d31a85f96aa35f6fb2ac6d5a696d4008c86b.txt	Standard	62.4 KB	Mon May 11 15:15:04 GMT-400 2015
00160814e9788646dffcffccff70a1cabcaf70f1.txt	Standard	26.6 KB	Mon May 11 15:15:05 GMT-400 2015
00193303db5648902bebf79f6e02e77a59ecb073.txt	Standard	62 KB	Mon May 11 15:15:05 GMT-400 2015
001a0892ea619aaeb6fd4fe8a412a7447fe10270.txt	Standard	16.3 KB	Mon May 11 15:15:05 GMT-400 2015
001a6f0f80451059a7e597e8668f38c87f38b91a.txt	Standard	69.4 KB	Mon May 11 15:15:05 GMT-400 2015
001aa6d703a98d986efd7b3f2d3fca6b910fd6b2.txt	Standard	62 KB	Mon May 11 15:15:05 GMT-400 2015

The input to the Map function of the EMR that the indexer used was the key,value pair

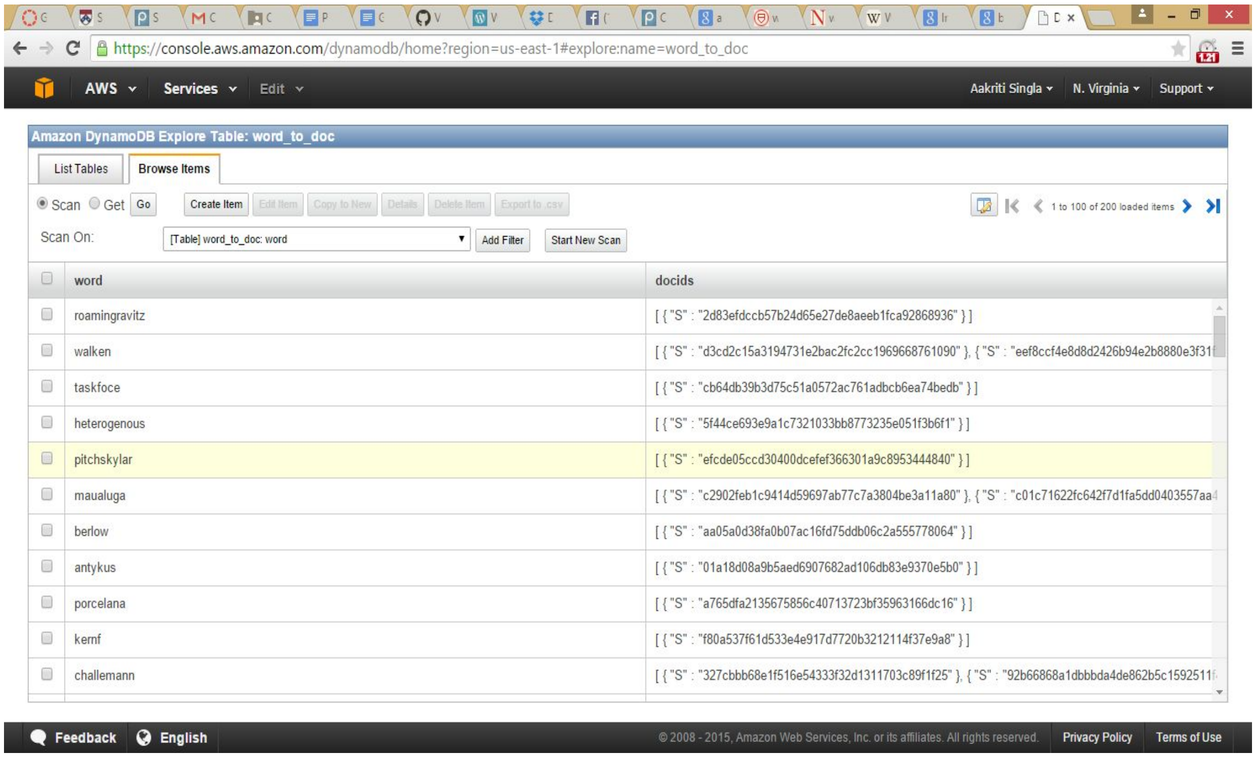
which included document-id as the key and the url as the value. Inside the map function, we fetch the contents of the given document id from the corresponding bucket name in S3. Once the contents are fetched, we use JSoup to parse the HTML content, and build the inverted index associated with that word. While parsing the HTML content, we perform proper tokenization of all the words encountered, as in we are removing all the leading and trailing special characters, moreover we are allowing only a few special characters to be there in my words, also we are removing the stop words, and using a Morpha Stemmer to do stemming of the given words. Thus the output of map is another intermediate key value pair example: word
docid:document-id&tagCounts:b=3,a=4,h1=1&tf:0.25&positions:1,9,10

This output of map goes as input to the Reducer which simply aggregates all the values associated with the given word(key). Thus the final output of the reducer is:
word   <list of docids><list of features associated with the word><list of positions where that word appears in that given document>

We are using Array List to store the document-ids, Hash Map to store the different features associated with that word, and another arraylist to store the positions at which the word was found in that given document. We are also simultaneously writing to the DynamoDB as well. Thus at the end of the completion of the Map-Reduce Job of the indexer, the followinfg tables in DynamoDB are made:

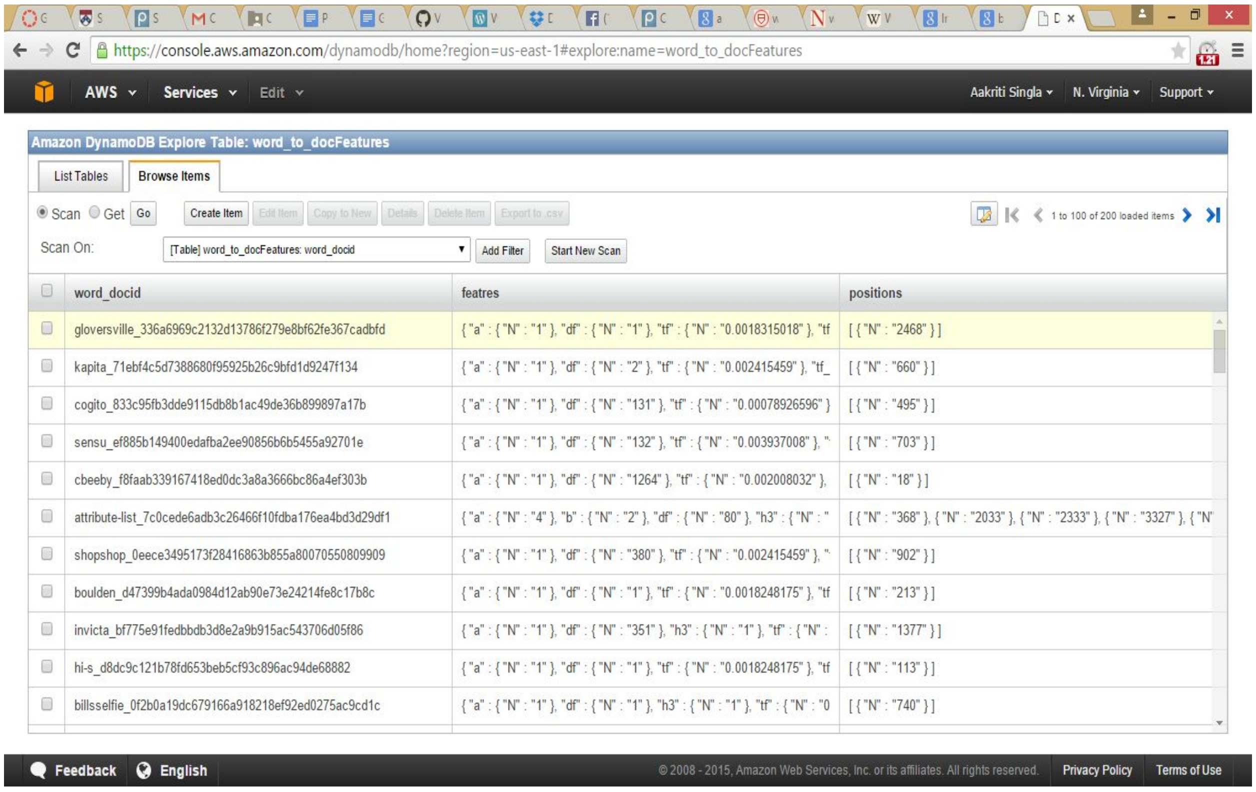
word_to_doc: This table has the mapping of word to the list of all the documents in which that word was found. Hence, in this table my word acts as a Primary Key, and we put a Hash Index on word for fast lookup.

word_to_docFeatures: This table has the mapping of word_docid to the list of features and list of positions that the word was found in that given document. The list of features is a hash map which holds the value of various features like tag counts, tf, df, tf_idf.



word	docids
roamingravitz	[{ "S" : "2d83efdccb57b24d65e27de8aeeb1fca92868936" }]
walken	[{ "S" : "d3cd2c15a3194731e2bac2fc2cc1969668761090" }, { "S" : "eef8ccf4e8d8d2426b94e2b8880e3f31
taskfoce	[{ "S" : "cb64db39b3d75c51a0572ac761adbcb6ea74bedb" }]
heterogenous	[{ "S" : "5f44ce693e9a1c7321033bb8773235e051f3b6f1" }]
pitchskylar	[{ "S" : "efcde05ccd30400dcefef366301a9c8953444840" }]
maualuga	[{ "S" : "c2902feb1c9414d59697ab77c7a3804be3a11a80" }, { "S" : "c01c71622fc642f7d1fa5dd0403557aa
berlow	[{ "S" : "aa05a0d38fa0b07ac16fd75ddb06c2a555778064" }]
antykus	[{ "S" : "01a18d08a9b5aed6907682ad106db83e9370e5b0" }]
porcelana	[{ "S" : "a765dfa2135675856c40713723bf35963166dc16" }]
kemf	[{ "S" : "f80a537f61d533e4e917d7720b3212114f37e9a8" }]
challemann	[{ "S" : "327cbbb68e1f516e54333f32d1311703c89f1f25" }, { "S" : "92b66868a1dbbbda4de862b5c1592511

© 2008 - 2015, Amazon Web Services, Inc. or its affiliates. All rights reserved.


https://console.aws.amazon.com/dynamodb/home?region=us-east-1#explore:name=word_to_docFeatures

Amazon DynamoDB Explore Table: word_to_docFeatures

Scan On: [Table] word_to_docFeatures: word_docid

1 to 100 of 200 loaded items

word_docid	featres	positions
gloversville_336a6969c2132d13786f279e8bf62fe367cadbfd	{ "a" : { "N" : "1" }, "df" : { "N" : "1" }, "tf" : { "N" : "0.0018315018" }, "tf	[{ "N" : "2468" }]
kapita_71ebf4c5d7388680f95925b26c9bfd1d9247f134	{ "a" : { "N" : "1" }, "df" : { "N" : "2" }, "tf" : { "N" : "0.002415459" }, "tf_	[{ "N" : "660" }]
cogito_833c95fb3dde9115db8b1ac49de36b899897a17b	{ "a" : { "N" : "1" }, "df" : { "N" : "131" }, "tf" : { "N" : "0.00078926596" }	[{ "N" : "495" }]
sensu_ef885b149400edafba2ee90856b6b5455a92701e	{ "a" : { "N" : "1" }, "df" : { "N" : "132" }, "tf" : { "N" : "0.003937008" }, "	[{ "N" : "703" }]
cbeeby_f8faab339167418ed0dc3a8a3666bc86a4ef303b	{ "a" : { "N" : "1" }, "df" : { "N" : "1264" }, "tf" : { "N" : "0.002008032" },	[{ "N" : "18" }]
attribute-list_7c0cede6adb3c26466f10fdba176ea4bd3d29df1	{ "a" : { "N" : "4" }, "b" : { "N" : "2" }, "df" : { "N" : "80" }, "h3" : { "N" : "	[{ "N" : "368" }, { "N" : "2033" }, { "N" : "2333" }, { "N" : "3327" }, { "N"
shopshop_0eece3495173f28416863b855a80070550809909	{ "a" : { "N" : "1" }, "df" : { "N" : "380" }, "tf" : { "N" : "0.002415459" }, "	[{ "N" : "902" }]
boulden_d47399b4ada0984d12ab90e73e24214fe8c17b8c	{ "a" : { "N" : "1" }, "df" : { "N" : "1" }, "tf" : { "N" : "0.0018248175" }, "tf	[{ "N" : "213" }]
invicta_bf775e91fedbbdb3d8e2a9b915ac543706d05f86	{ "a" : { "N" : "1" }, "df" : { "N" : "351" }, "h3" : { "N" : "1" }, "tf" : { "N" :	[{ "N" : "1377" }]
hi-s_d8dc9c121b78fd653beb5cf93c896ac94de68882	{ "a" : { "N" : "1" }, "df" : { "N" : "1" }, "tf" : { "N" : "0.0018248175" }, "tf	[{ "N" : "113" }]
billsselfie_0f2b0a19dc679166a918218ef92ed0275ac9cd1c	{ "a" : { "N" : "1" }, "df" : { "N" : "1" }, "h3" : { "N" : "1" }, "tf" : { "N" : "0	[{ "N" : "740" }]

© 2008 - 2015, Amazon Web Services, Inc. or its affiliates. All rights reserved.

3.3 PageRank

3.3.1 Mapper

PageRank was accomplished using an iterative MapReduce procedure. To accomplish this, the reducer output from one iteration of the procedure served as input to the mapper in the next iteration. The input to the map stage was of the format:

DocID currentPageRank {List of DocIDs of pages linked from this page}

e.g: Page1 1.45 Page2,Page3,Page4,Page5,Page6

Given this, each mapper would output three things:

1. For each OutputDocID in the list of outbound links, emit:

   OutputDocID currentPageRank DocID Total number of outbound links

   e.g: **Page2 Page1 1.45 5**

   This is in a way emitting the documents PageRank contribution to each of the documents it is linked to
2. DocID with a special symbol to indicate that the current document has been crawled. This will be used in the reduce phase to avoid assigning PageRanks to pages that have not been crawled

   e.g: **Page1 ***
3. The DocID with the same list of outbound links, and a special symbol to indicate that the list is beginning. This is primarily to propagate the outbound list through the reduce process, and cause its output to be something that can be used as input to the mapper in the next iteration.

   e.g : Page1 !Page2,Page3,Page4,Page5,Page6

3.3.2 Reducer
Due to the shuffle stage between map and reduce putting together everything with the same key, the reducer would receive the something like this as the input:

Page2 {*, Page1 1.45 5, Page3 1.05 3, !Page3,Page5,Page6}

Given this, the reducer would first check for the existence of the ***** character to ensure that Page2 was indeed a page that has been crawled. If the ***** does not exist, that means that the was page doesn't actually exist or hasn't been crawled yet, and we just return without calculating the PageRank in that case. Otherwise, we calculate the current pageRank of Page2 by summing up PageRank "contibutions" from each page that linked to it by dividing the PageRank of that by the number of outbound links from that page. The final pageRank is computed with this formula:
(1 - d) + (Sum of PageRank contributions)*(d)

d was the "damping factor" used to reduce the impact of "sinks", pages that have no outbound links, on the final pageRank calculations. With the PageRank value computed, the reducer would then output it along with the list of outbound links (if any) to have the output of the reducer be in the same format as the input to the mapper:
DocID currentPageRank {List of DocIDs of pages linked from this page}

3.3.3 Information Flow
The initial input to the PageRank module was multiple "web-graph" files generated when doing the crawling, with each line formatted to: **DocID 1.0 {List of DocIDs of pages linked from this page}** with 1.0 being the starting rank for every page. These web-graph files were transferred from the BerkeleyDB instances of each node to S3 files. From that point, EMR came into play and ran the MapReduce algorithm described above, with its output stored in S3, and eventually transferred to DynamoDB for use by our search engine

3.3.3 Challenges
Figuring out the number of times PageRank should be iterated to reach convergence was a slight challenge. For smaller datasets used for local testing, convergence (to the level of 10^{-4}) was reached at around 5 iterations. For the full dataset, 20 iterations were used to be safe even though most of the observable results had very little difference after 10 iterations.

The two other main challenges encountered were dealing with dangling links and sinks. These problems were dealt with the implementation details specified above: the decay factor of 0.85 (Used from the lecture sildes) and the "exists" symbol emitted by the mappers

3.4 Search Engine

Functionality:

The search engine maintains a list of weights assigned to feature vectors. After executing the getItems on DynamoDB, the search engine then multiplies the corresponding weights to features and performs a summation of them to arrive at an IR score. This score is then multiplied with the pagerank obtained my mapreduce and stored in DynamoDB. The final score is passed back to the servlet which sorts it in descending order of overall score and displays.

We have implemented Pagination to display only specific number of results per page for convenience of visibility.

FEATURE WEIGHTS

Title	Meta	H1, H2 <headings>	H3, H4, H5, H6 <headings>	Anchor	TF-IDF
3	3	3	1	6	50

Also note that the tf-idf weights are normalized and the rest parameters were taken as is.

3.5. Extra Credit

SPELL CHECKER

We have implemented the spell checker utility in the search engine. We have taken a traditional bag of words approach to this by populating hashmap with the words and their corresponding count values. Then search for a word, if not present is permutated with all alphabetic combinations until it has a match with highest number.

One quick observation is that it has taken a significant time to load the results, ~1.5 sec with this utility on. But the results were promising.

INCLUDING DOCUMENTS AND WORD META-DATA

The indexer has made use of the meta tag while creating the inverted index associated with the word. This helped in improving the rankings.

4. Experimental Analysis and Evaluations

We will begin with some performance statistics of Search Engine with respect to time.

The results obtained by the Search Engine were very much relevant to the query and were organized according to the weights we assigned in feature vectors and were satisfactory.

In terms of quickness, We used the chrome extension of PageLoad time to determine average response time of search results.

PAGE LOAD STATISTICS

Query	Response time (Seconds)
Without Spellcheck	0.35
With Spellcheck	2.67
Less Indexed Pages (<10)	0.3
More Indexed Pages (>30)	1.8

Now moving towards the crawler:

The total number of documents crawled: 85,550

The total number of HTML documents crawled: 65,578

The indexing was performed on the 65,578 HTML documents for which we used 3 large ec2 instances which took around 3 hours to complete the job.

Also, mentioning about the infrastructure. The following configurations were finally deployed after experimenting with small and medium instances. This configuration has given us the best performance by far.

INFRASTRUCTURE

Component	Infrastructure	No. of Instances	Instance Size

Web Crawler	Ubuntu 14 EC-2 instances (stored Berkeley DB)	16 (1 Master and 15 workers)	Medium
	S3 Buckets	3	Standard
	Dynamo DB	1	Standard
Search Engine	Compute Optimized EC2	1	8xlarge EBS Storage, 10 Gigabit N/W performance
Indexer	Compute Optimized EC2	3	Large
Page Rank	S3, DynamoDB, EMR	11 (1 master, 10 workers)	2xLarge

5. Conclusion

The success story described throughout the report is an outcome of sleepless nights of coding, integrating, error debugging and bug fixing. The project started on a simplified infrastructure with lots of issues of memory leaks, time-outs in crawler, storage style for indexer, time out and configuration issues with EMR, optimization issues in search engine and basic ranking algorithm for IR calculations. However, the same was followed by a Distributed and Multithreaded clone of Mercator for Crawler with all data store on-disk in Berkeley, a highly data driven Indexer, full implementation of Pagerank and a highly optimized multi-threaded search engine with well designed IR ranking algorithm. We started with a NOT AT ALL FAST BUT only FURIATING engine which eventually reached to FastAndFurious 1.0 which is highly scalable, persistent and fault tolerant. Till the time data base is not corrupted, the system can be started, closed and restarted at any time without loss of state, with error debugging helping us ensure a robout system delivered. The machines used used the maximum processing power due to multi-threaded system and ensured high performance. We believe that we not only achieved the initial goals of our application and exceeded the highly fault tolerant robust system as the final implementation