Group 19: Douglas Dolitsky, Derick Olson, Sara Weinstein, Kyle Wilcox

1 Introduction

General Approach

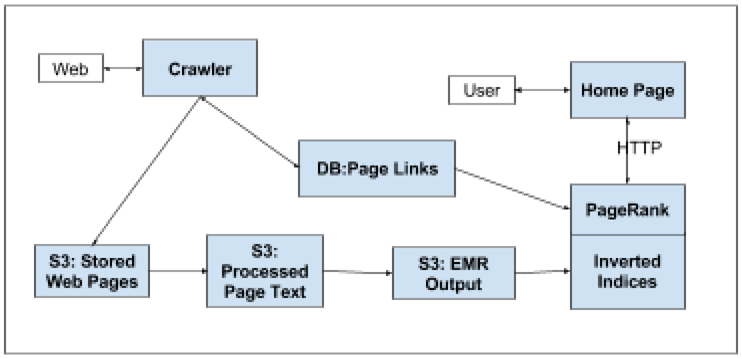
We assigned one person to be the key team member responsible for each of the four major areas, and worked in pairs or as a group as much as possible. We divided the work according to the to the following labor assignments:

Crawler: K. Wilcox, key; D. Olson, secondary
Indexer: S. Weinstein, key; D. Dolitsky, secondary
PageRank: D. Dolitsky, key; S. Weinstein, secondary
Search Engine: D. Olson, key; K. Wilcox, secondary

April 6	First meeting. Organize high-level infrastructure and AWS tools (S3, EMR, DynamoDB)
April 13	Initial implementations. Begin crawling pages.
April 14-24	Independent work. Meetings to maintain interfaces between components. After some time adjusting to using the AWS tools, we each worked independently for about a week, and then began to review our interfaces and make adjustments so that our code worked together.
April 25	Begin indexing on small data sets. Work out issues with indexer pre-processing.
April 30	First full integration test (without PageRank), which was successful Second integration test (with PageRank), successful but queries very slow
May 2	Demo final product

2 Basic architecture:

Below is a simple sketch of the architecture of our search engine:

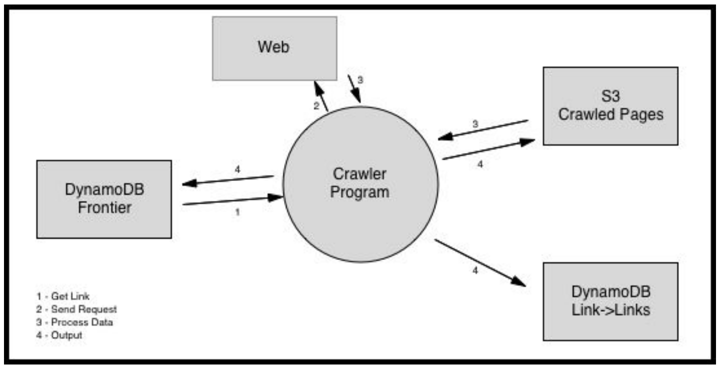


Crawler Data Structure

The crawler downloads web pages, extracts the link on the page, and also obeys robot rules. The link information is saved in a DynamoDB on AWS in two separate tables, one for forwarding to the PageRank function, and one to maintain the frontier information for the crawler. Each page is saved as an object in an S3 bucket, with the URL name as the name of the object. The rate of the crawl is around 300k a day, and needs to be checked on at times to make sure it doesn't get stuck in one site.

Out of Heap Space: When first scaling the crawler to be able to run in the background on its own, it would run out of memory at around 2k pages due to frontier storage running out of memory. We noticed that after about 1000 pages crawled the program drastically slowed because of the size of the data structures.

To solve this problem, the frontier was moved to a data structure in DynamoDB. Other data structures such as HashMaps of Robot Rules and Host: Last Accessed Time were limited to 1000 in order to keep the heap relatively small, which wouldn't slow the crawler.
Distribution: The crawler was distributed over 9 EC2 instances with the same crawler jar which takes in the following arguments:: max file size in MB, max number of links to crawl in 1K, max depth allowed in a site, worker id, and seed urls. Each worker was assigned an id and processed links from the frontier DB associated with that id. Links were assigned IDs similar to homework 3 (i.e. hashing). This method distributed the links evenly, however testing showed that crawlers performed just as well by letting them pull from the frontier on an "as needed" basis.
Checking for Duplicates: Due to scale of data, keeping an internal data structure was not an option. Since we saved the documents to S3 with the key as actual URL, we were able to just check to see if that object existed in the DB and if it did we could ignore it.
Robot Rules: The robot rules class that was created in the origin HW2 was upgraded slightly to handle the broken formats that it encounters while parsing. The robot rules handles crawl delays by checking (lastAccessed - currenttime > crawl delay), if it is the link is send back to be crawled at a later time.
Crawler Traps: Each page that was added to the frontier was given a depth. The depth was determined by whether the same host linked to another page within its site. For example, foo.com -> foo.com/bar would result in foo.com/bar having a depth of 1. The depth limit was set from the command line. Initially set to 6, after seeing the crawler trapped in one site we set the depth to 2 to have a more diverse corpus.
Language: Initially the crawler did not discern language so the corpus does contain some links in different languages. Wikipedia data was a problem, so language detection was added. To adapt to the existing system, we looked for the first instance of the language tag in the html, which according to w3.org is in the form: `<html lang="fr">`. If these were encountered and it was not some type of "en" we would return no links.
Error in Request: The crawler deals with response codes as follows: For 2xx the crawler attempts to parse the content and output the results to S3 and Dynamo. For 3xx (besides 304) the crawler attempts to look for the "Location" header for the new URL to send back to be rechecked. For 4xx - 5xx the crawler ignores and outputs that there was an error in the request. The crawler also handles timeouts, by logging that the timeout has occurred and then returns no links found. This actually occurred more often than originally anticipated due to dead links being given to the frontier.
Bottlenecks: The bottlenecks happened in two areas, the first being parsing the content to extract the links. Links were originally processed using JTidy, but passing weighty DOM objects was slow. We decided to extract the links by simply looking at the content. The second bottleneck was writing to the S3 and the Frontier table. Due to some pages having 100s of links the crawler at times had to slow down to wait as it wrote to Dynamo. To counter this, we implemented the frontier put requests in batches to speed up the process.
Crawler Workflow:



Inverted Index

Our inverted index is build with a three-stage data pipeline. We first pre-process the HTML document, then run a MapReduce job to aggregate the url data for each word, and calculate the tf/idf for each word-url pair. Results are uploaded to DynamoDB for access by the search engine.

Pre-processing: *Download S3 Crawler output + Upload to S3 EMR input*

The input to our pre-processing is raw html saved as S3 objects by the crawler. The output is an unprocessed inverted index. We pre-process each S3 object by parsing into a DOM object with JSoup to extract a block of text representing the title and body of the original html document.

This text is then stemmed, tokenized, and filtered for common stop-words and non-alphabet characters. The resulting block of stemmed text is written out in the following format:

```
<WORD>        <POSITION>    <URL>         <DOC_LENGTH>        <IS_TITLE>
```

We split words up line-by-line in order to maintain position data at each line because giving the mapper a large block of text would lose position information if the original block of text was partitioned into several map() calls.

The inefficiency of redundant information (url length on every line for every word) was traded to maintain the “state” of each word/url pair in the document (e.g. word attributes: position, is_title and document attributes: length, url), regardless of the order or partition they are processed in.

We repeat this pre-processing stage with bigrams set by adjacent words in the document in the format: term1:term2, term2:term3, … termN-1,termN. This allows us to process bigrams as if they were individual words, using the same data pipeline as below.

Inverted Index: *Read S3 EMR input + Write S3 EMR output*

We calculate the tfidf for each document,url pair by running our pre-processed data through a MapReduce job. This job combines word-url-position triples into word-url-list(position) triples:.

[(W URL POS1),(W URL POS2),(W URL P3)] => (WORD URL P1,P2,P3)

The mapping phase emits the word as a key and passes along the position, length, and is_title information associated with the word. The reducing phase calculates components of the tfidf score. The number of documents mapped to each word WORD_CNT is maintained by mapping each unique url associated with the word key. By maintaining the urls associated with this word, it can calculate the number of urls containing this term, HIT_CNT. We pass the value of TOTAL_DOCS to the Indexer.jar job itself as a context parameter. Finally, the DOC_LENGTH is passed from the mapper from the original file. With this parameters, we can calculate the TFIDF for each document. This information is output into an S3 bucket in the following format:

```
<WORD>        <URL>         <IS_TITLE>      <TFIDF>             <POSITIONS>
```

We initially chose to combine all (URL,IS_TITLE,TF_IDF,POSITIONS) 4-tuples corresponding to a particular word in a single line, but found that the MR jobs ran faster when the words were emitted separately for each url.

Database Transfer: *Download S3 EMR output + Upload DynamoDB table*

The final indexer script uploads the S3 data into DynamoDB, by parsing each EMR’s output S3 bucket storing the values in a database with WORD as primary key and URL as secondary key.

In a future iteration, we would like to add PageRank scores to each url-document pair at this stage in order to avoid the overhead of fetching the PageRank of each document individually during the ranking phase in the search engine.

PageRank

The PageRank-computing component used the data collected by the crawler to determine the importance of each page. This program uses the “Links” DynamoDB table, which stores pairs of urls and out-links of each url, as input. After running the PageRank algorithm on the data, the pagerank of each url was stored in another DynamoDB table.

Design Choices

Originally, we implemented the program to run on multiple machines using MapReduce. The map function was responsible for partitioning each url's pagerank amongst its out-links, while the reduce function was responsible for combining all of the contributions from each url's in-links into a single pagerank score. However, due to severe technical challenges getting the program to run using Apache's Hadoop framework and AWS's Elastic MapReduce, this approach was eventually rejected.

Due to the relatively small corpus of pages compiled by the crawler at the time that the PageRank component was initially implemented, it was potentially feasible to run the program locally on a single machine. However, the computer being used did not contain enough RAM to store all of the urls and out-links. We first attempted to circumvent this issue by representing the urls in a more spatially-efficient manner. Specifically, each url would be mapped to an integer, which would be used as an identifier by the PageRank program. We decided not to use this technique due to lack of scalability, as the crawler could eventually add too many links for even the integer representations to fit into memory. We ultimately chose not to store all of the links in memory, but rather to retrieve each url's out-links from the database when needed. While interacting with the database so often definitely slowed the PageRank process, this approach seemed to offer the best overall mix of scalability and simplicity.

We then had to figure out exactly how to calculate the pageranks. We decided to use a damping factor of 0.85 (as is done by Google) and to remove all sinks from the graph, in order to deal with pagerank hogs and deliberate use of links to improve pagerank. We also had to figure out when to terminate our PageRank program. Obviously, waiting until absolute convergence was not practical. Therefore, we had to come up with our own definition of convergence. We chose to use the maximum change in pagerank of all pages over a given iteration to access convergence. By limiting the maximum change in pagerank, we guarantee that there can be no single page whose pagerank significantly overstates or understates its importance. Such an occurrence could be detrimental to our query results. We set our maximum delta value to be 1/10N, where N is the total number of pages, which is much smaller than the initial pageranks of 1/N. However, due to time constraints, we were never able to reach our convergence.

Ranking

We calculate the ranking of document results by using a combination of the existing scores in the database (e.g. tfidf, in_title), and our method and order of running term queries to the database (e.g. bigram_match, in_url, multi_term_match, proximity).

Documents are ranked on the following features:

relevance	*is the term in this document's url or title?*	*(bool constant)*
tfidf	*what is the tfidf score for word-doc pair?*	*(double 0.0-1.0)*
page_rank	*what is the page_rank of this document?*	*(double 0.0-1.0)*
bigram_match	*was this document part of a bigram match?*	*(bool constant)*
hitcount	*how many query terms match this doc?*	*(integer >=1)*
closeness	*how close were the positions of multiple matches?*	*(double >=1.0)*

The processes of ranking begins with search terms. Our goal is to end up with a list of RankableDocuments which can be sorted in ascending order. The RankableDocument data structure is essentially a wrapper for the scores above, with a function to combine them into a single comparable score (double) used for sorting:

*DocScore = (bigram_match) * (w1*hitcount + w2*relevance) * (w3*page_rank) * (w4*tfidf + w5*closeness)*

Note that each score is weighted for normalization and relative priority

The *tfidf* and *page_rank* scores are simply accessed from the database. The relevance, bigram_match, hitcount, and closeness scores must be computed on the fly because they cannot be pre-computed.

The *relevance* score is computed by combining title and url information. The database contains a boolean value for whether or not the term was in the title of the document. We add url matches by scanning the database for urls that contain the token.

The *bigram_match* score is computed by making separate queries for bigrams and single terms. We set a boolean *is_bigram* to true for all documents returned in the initial bigram query. During the combining phase, these values are OR'd so that the final documents receive a boost for being a bigram match.

The *hitcount* and *closeness* scores are computed during the combining phase because they take into account multiple document hits for each token in a query. The hitcount is simply the sum of documents returned for a multi-token query. The closeness is computed by checking if the positions of the various tokens in the document are within a certain range of each other, with higher values for closer tokens.

Each search term is first processed with the same tokenizer-stemmer used on the crawled pages. Cleaned tokens are then used to query the database. We then build the document results by storing each document returned in a DocumentMap of Urls to Lists of RankableDocuments (DocumentMap data structure structure is a wrapper for Map<Url, List<RankableDocuments>> with additional functions for flattening the list of RankableDocuments into a combined RankableDocument). After combining the lists of RankableDocuments for each url, we sort them by their combined score.

EVALUATION

Crawler:

The crawler downloaded 300k pages a day, or about 12.5k an hour. Measured using aws cli tools:
Command: aws s3 ls s3://crawler-data032293041316-t4/ --recursive | wc -l

Page Rank:

Each iteration calculates pagerank of approximately 9 pages/second. With a 40,000 url corpus, this takes approximately 74 minutes. (Does not include time to store to DB, which does not have to be done after every iteration)

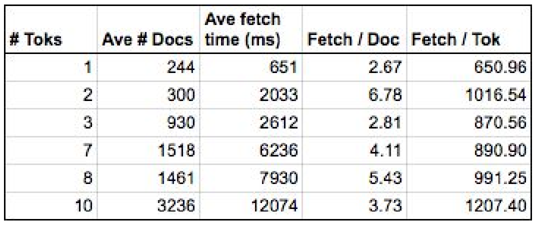
Search Engine:

To evaluate our search engine, we ran queries on a set of terms and measured the number of milliseconds for the database fetch (*fetch_time*) and document sort (*sort_time*), which sum to the *total_time* metric. For each search query, we ran three trials and took the average running times in milliseconds for our analysis. In addition, we tracked how many tokens the query was parsed into (e.g. account for dropped stopwords) and record the number of documents returned. We chose queries ranging from one to ten words, focussing on common words and current events.

We analyzed our data based on two hypotheses: (1) longer search queries take longer to run, and (2) more documents results take longer to run. We believe that longer queries will result in more fetches from the database for each term, as well as a longer combine phase in order to calculate scores per document. We believe that a higher number of documents returned will scale approximately linearly (O(nlogn) to be precise) because we are using a priority queue to sort them.

The vast majority of time is spent in the fetch phase. However, since the number of documents returned ranges only from 10-3000, we do not have the data to tell how fast the sort_time would be on a larger document set. Sort time appears to increase linearly with the number of documents returned, so as our corpus grows it is likely that sort time would take a larger percentage of the overall time.

The following table shows that even though the range documents returned changes dramatically, the time to fetch scales linearly with the number of tokens entered. The average fetch_time per token remains constant at around 1000ms for all queries we tested.



# Toks	Ave # Docs	Ave fetch time (ms)	Fetch / Doc	Fetch / Tok
1	244	651	2.67	650.96
2	300	2033	6.78	1016.54
3	930	2612	2.81	870.56
7	1518	6236	4.11	890.90
8	1461	7930	5.43	991.25
10	3236	12074	3.73	1207.40

Table 3: Results by documents, by token

Aggregate results across all fetches show that the average percentage of time spent fetching is 99.5%, where the average time for sorting is 0.50%. Currently, it takes about 1 second per token in the search query. Since search queries are rarely longer than short to medium-length phrases, this is acceptable. Furthermore, we are limited by the time it takes to access DynamoDB, so best way to improve on this metric would be to query the database for multiple tokens at once.

The percentage of sorting time would likely change with scale. For sorting, we averaged about 2ms per 100 documents, but 20ms per 100 documents in the worst case. Assuming that the sorting time does increase approximately linearly by the number of documents returned, we can hypothesize that sorting a result on the order of 10k documents would take 200ms. With a corpus of around one million documents, the largest results would be on the order of 100k documents. The sort time for these would be on the order of 2 seconds for the average case. We think that 2 seconds is an acceptable sort_time, but we could improve on it by eliminating irrelevant results from the initial sort. If we specified rules for the removal of candidate documents, such as meeting a document quota with certain high-enough scores, then we would eliminate the sorting step and improve our runtime.

LESSONS LEARNED

Bottlenecks

Our primary bottleneck was indexing. Our data pipeline was not designed to be parallelizable, so its runtime grew linearly with our corpus.

We've learned that hadoop does not handle a large number of small files very well--it is much better for it to handle a small number of large files. Our preprocessing step output one file for each document file of its input, so the number of files was equal to the number of documents crawled. Had we run our job on a large enough corpus, we would likely have run out of memory on the Master node, which stores filename metadata in memory. A better approach would be for the preprocessor to concatenate all lines of processed information into a single file, or into a small number of files which would allow for better parallelization.

Another option to speed up indexing would be to break down the indexing into several simpler EMR processes and run them consecutively, for example calculating the TF/IDF independently of aggregating the data and joining the databases afterwards.

CONCLUSION

Overall this project was a success. With more time to refine code, the indexing could be sped up and the crawler could easily reach 1 million pages, and this would be an excellent search engine.