CIS 455/555

Distributed Search Engine : Grumper

Aayushi Dwivedi | Ankit Mishra | Anwesha Das | Deepti Panuganti

1. Introduction

1.1. Project goals

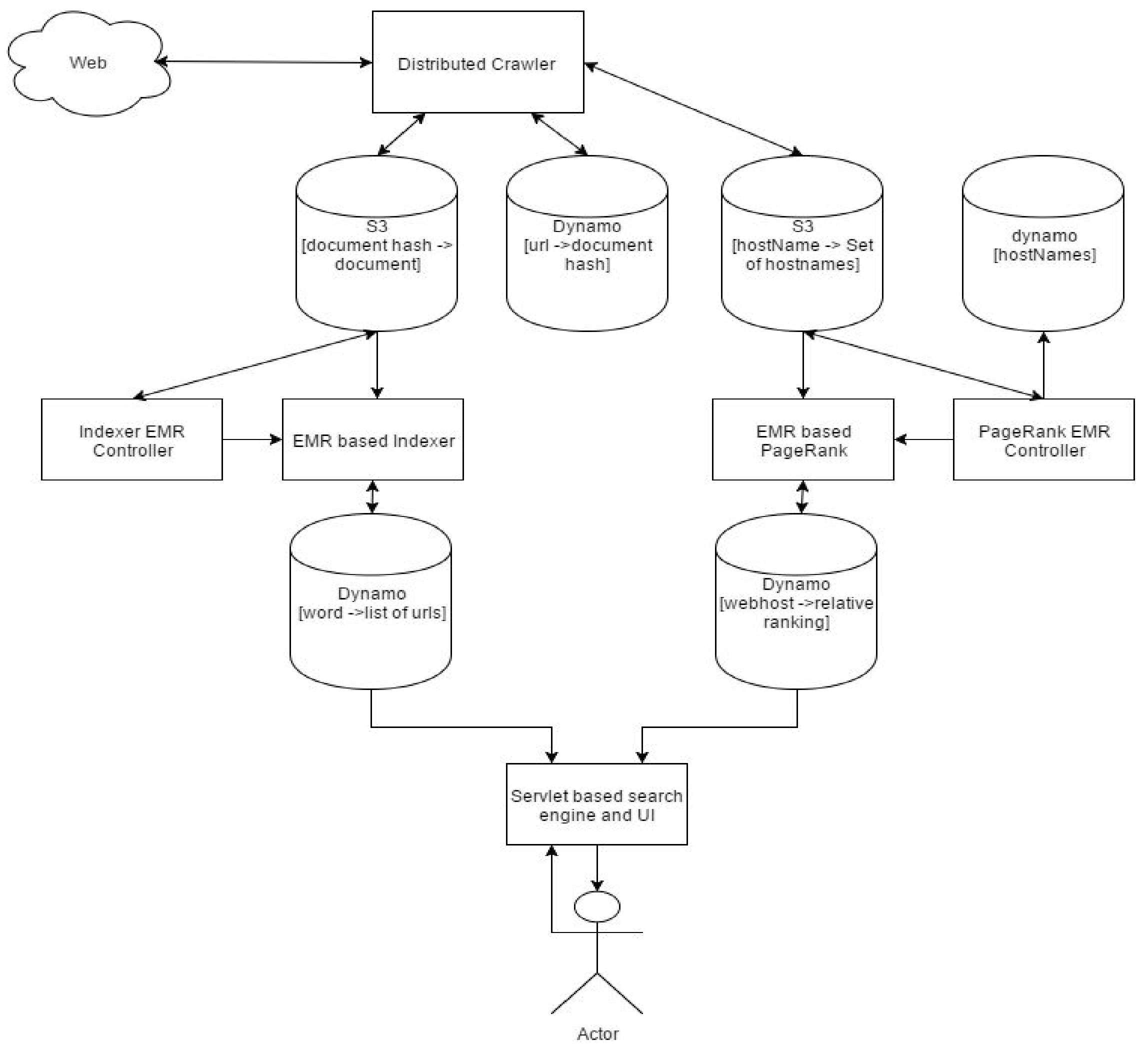
To design, implement and test a distributed web crawler, indexer and ranker to serve the final goal of making a search engine.

1.2. Division of labor

Component	**Assigned To**
Crawler: design & architecture	**Ankit , Anwesha**
Indexer: design & architecture	**Aayushi, Deepti**
PageRank: design & architecture	**Ankit**
Search Engine: design & architecture **Search Engine: UI**	**Aayushi, Deepti** **Anwesha**

1.3. Project Architecture and High Level Approach

The project architecture is divided into 4 different components i.e. Crawler, Indexer, PageRank ranker and search engine. All of these components run in parallel and independent of each other and communicate through databases.


Web
Distributed Crawler
S3 [document hash -> document]
Dynamo [url ->document hash]
S3 [hostName -> Set of hostnames]
dynamo [hostNames]
Indexer EMR Controller
EMR based Indexer
EMR based PageRank
PageRank EMR Controller
Dynamo [word ->list of urls]
Dynamo [webhost ->relative ranking]
Servlet based search engine and UI
Actor

2. Components

2.1. Distributed Crawler

2.1.1. Architecture

The web crawler we built for this project is a distributed and incremental crawler which respects the robots.txt restrictions. There is one master servlet, which takes the seed url inputs and assigns them to multiple workers based on SHA-1 hashing. Each worker is multithreaded and is responsible for crawling a particular set of domains. If it comes across a domain it is not responsible for, it writes it to file and then periodically sends the data by opening a socket to the other workers.

We have implemented message digesting. We use SHA-1 hashing to digest the content of the web document. Further, we have split our document store between dynamo and S3 to allow url -> hash and hash -> document look up. The crawler workers periodically push all crawled documents to S3, along with hostName -> Set {hostNames} mappings for all the URLs crawled so far (for pagerank) to another S3 bucket.

Since the URL queue may get very big, we limited its size to hold only 5000 urls in memory. The rest are stored in berkeley db and fetched when the in-memory queue is empty. We also used the same method to implement incremental crawling, i.e., when the crawler is shut down, the destroy method of the worker servlets saves the current in-memory queue to berkeley db, which is then fetched and used once the crawler is restarted.

2.1.2. Working

We used Jetty to run our master and worker servlets on EC2 nodes. We implemented a Web UI for us to provide seed urls and also specify the number of threads each worker should use. On one medium EC2 node, we ran the master and two workers, each with ~100 threads, and gave it a wide range of seed urls, ranging from wikipedia to espn. We also constantly monitored the worker status on the Web UI, and incase of any problems, stopped and restarted them with more seeds.

2.2. Indexer

We have built a distributed indexer that takes data collected by the crawler and creates inverted index out of it. This inverted index is used to answer search queries posed at the search engine.

2.2.1. Architecture

Input: Crawled documents stored in S3 bucket.

Output: Tables containing inverted index

Elastic MapReduce framework is used to run the indexer jobs and output of each reducer is stored directly to a table present in Amazon DynamoDB.

The schema of tables used to store indexed data is given below:

Table Name	**Primary Key (String)**	**Attribute (String)**
Unigram / Metadata	Word	Postings (tab separated list)
Bigram	Word (space separated)	Postings (tab separated list)

For each word in the inverted index we store a sorted postings lists. Each of the postings consists of *space* separated values: <URL> <TFIDF> <IDF>. A postings list is a sorted list of *tab* separated postings and is sorted based on TFIDF (descending order) .

When a table is queried for a word, the corresponding postings list is unmarshalled and an ArrayList of Postings object is returned.

2.2.2. Working

2.2.2.1. Merging Documents:

Before beginning a MapReduce job, we first merge the crawled documents into blocks of 64MBs. We use the IndexerEMRController class for this purpose. It collects all crawled documents and then groups them into blocks of upto 64MBs each. Each line in a given block represents a single crawled document. These merged documents are then stored in the input S3 bucket of the indexer.

2.2.2.2. MapReduce:

We use three different MapReduce jobs to create three different inverted indexes. All three jobs work on the same merged input documents and have same reduce jobs. They differ only in their map phase. Also, we use the default input and output formats for MapReduce.

Input to a Map: Input to the map is a key-value pair with byte-offset in the file as key and a single line of the file as value. The value is a JSON string. We extract the HTML document from JSON and get all the required information from the HTML documents using JSoup.

Pre-processing of text: We use following pre-processing techniques:

- Lower casing the text
- Stemming
- Removing stop words like the, is, etc.
- Removing strings that consist only of numbers
- Filtering all characters not present in English alphabet

Unigram Index Map: Emits a tab separated key-value pair of a word and URL of the HTML document that contains the word.

Bigram Index Map: Emits a tab separated key-value pair of a bigram and URL of the HTML document that contains the word. Words in a bigram are *space* separated.

Metadata Index Map: Emits a tab separated key-value pair of a metadata and URL of the HTML document that contains the metadata. We store content of *title* and *meta (description , content)* tags in the HTML.

Reduce: Input to a reducer is a word and an iterable consisting of all the URLs containing that word. Job of the reducer is to compute TFIDF and IDF of the word and create sorted postings list. The Metadata reducer computes term frequency instead on TFIDF. The output can either be written to a S3 bucket or directly to a DynamoDB table.

2.2.3. Experiments

1. Merging the crawled documents into blocks of 64MBs helped in speeding up the Map phase.
2. We experimented with two different types of word features : n-grams and metadata. We decided to go exclude trigrams from our model since indexing trigrams took too long.
3. We stored a sorted list of postings list to reduce the overhead of sorting at the search engine's end.

2.3. PageRank

2.3.1. Architecture

Pagerank module has been has been implemented as an iterative mapreduce job. The input to the pagerank module was a map of hostName -> Set{hostNames}, which is written to S3 by the crawler. The output is float pageranks stored in dynamodb, at the end of each iteration. This allows us to feed the output of one iteration to the next one very easily. Elastic MapReduce framework is used to run the pagerank jobs and output of each reducer is stored directly to a table present in Amazon DynamoDB.

The schema of tables used to store pageranks is given below:

Table Name	**Primary Key (String)**	**Attribute (Float)**
pagerank	hostName	rank

2.3.2. Working

2.2.2.1. EMR Controller:

The pagerank module is controlled by the PageRankEmrController. The controller starts 3 threads i.e. controller thread, dynamo thread and S3 thread.

The controller thread is responsible for starting an EMR cluster. The dynamo thread is responsible for clearing the previous version of the pagerank table and creating a new pagerank table. The S3 thread is responsible for clearing the output directory of the mapreduce job, merging the hostName -> Set{hostNames} mapping, into groups of 150. Further, it creates hostName entries in the domains table, for each domain that has been crawled, which is used by the mapreduce job to get rid of dangling links. Finally, once the threads have completed, the PageRankEmrController starts the EMR job.

2.2.2.2. MapReduce:

The pagerank mapreduce job is performed iteratively for a fixed number of iterations. The PageRankTool class is responsible for running the mapreduce job iteratively within a single EMR step.

Input to a Map:

Input to the map is a whole file containing the a json array of hostName -> Set{hostNames}. The map is responsible for unmarshalling the json array into individual components and then running the map logic on each entry of the array.

For each hostName -> Set{hostNames} entry, the map calculate the pagerank that a page transfers to it's forward links. The map output is: key - forward link hostName, value - rank received from a host. The map reads the current rank of a hostname from the dynamodb table. By default all pages have a rank 0f 1.0. Further, the map is responsible for removing dangling links and self loops.

Reduce:

Input to a reducer is a hostname and an Iterable over all the ranks that the host has received from it's backlinks. The reducer is responsible for summing up the rank that a host has received and writing the result into the dynamodb table.

2.3.3. Experiments

We experimented with the pagerank module to improve the execution time of the emr job by merging hostName -> Set{hostNames} mapping. The runtime of the emr job then dropped by over 50%.

Further, we experimented with running pagerank as a converging algorithm by keeping the emr output in s3 and then, checking for convergence by using a delta of 0.001, and then writing the new ranks into db. The code for this can be found in EMRController class. However, we found that this resulted in the total runtime of over 7 hours.

2.4. Search Engine

2.4.1. Architecture

Input: Search query as a String.

Output: Top 50 URL results.

The search engine consists of the User Interface and a servlet based engine. The UI consists of 2 html pages, the homepage and the results page. The query is sent to the POST method of the servlet using JQuery. The servlet computes all the required scores and returns the top matches in the form of a string delimited by spaces. The results are displayed 10 per page, and a preview of the page can be seen by hovering the mouse over the URL.

2.4.2. Working

The search engine is hosted using jetty on an EC2 instance. The URL to access the search engine is 52.90.111.118:8080/grumper.html.

Entering the Query

The user enters their query in the form of the Homepage of the search engine. The query is sent to the results page, which in turn sends it to the POST method of the servlet.

Preprocessing

The servlet receives the request, and first preprocesses it. Preprocessing includes tokenizing, stemming, removing all non-alphanumeric characters and converting all letters to lowercase.

Matches in Title and Meta Tags

First, the inverted index is queried to get meta and title tag matches for each term in the query (this data is in the metadata table). Each url is scored as the sum of the individual scores of each query term that matched that URL. Only URLs in which all the query terms match are considered as hits. If there are more than 100 matches, then only the top 100 matches are considered. Results are limited to top 6 from each domain to maintain diversity.

Matches in Content Tags

If there are less than 50 matches in the metadata table, then we look for matches in the content. For this both unigrams and bigrams are considered. 2 threads do the work, one queries the unigram table and the other queries the bigram table. Both the query and the document are represented as vectors, and the value of each term is its tf-idf. The scores for the urls are the cosine similarity between the query vector and the corresponding document vector. Once unigrams and bigrams have been computed, a weighted sum of the 2 scores is taken for each url, with the bigrams weighted higher than the unigrams. Since, the inverted index returns sorted results, only the top 500 are considered if the returned list is larger than that.

Final Scores

The final score of the url is the score computed with title/meta matches or unigram and bigram matches plus the pagerank for that domain. The pagerank is weighted less than the cosine similarity score, because we don’t want domains with a higher pagerank to overshadow more relevant results.

Caching

The user queries are cached, so if the same query is seen again results are returned directly from there instead of redoing all the computation. Berkeley DB is used to store the cached results.

2.4.3. Experiments

1. Title and Meta Tag Matches: We tried giving highest priority to documents that had the query terms within title and meta tags, since it was likely that these documents would be the most

relevant as compared to documents in which the query terms were only present in the content tags.

2. Final Score: We tried calculating the final score as Cosine Similarity x PageRank but this didn't allow us to weight the pagerank, so we used the sum instead. This helped retain the significance of the cosine similarity score.

3. Limiting Number of Matches: A huge number of matched urls retrieved from the inverted index would have slowed down the search engine. Since, the results were returned in a sorted order, it seemed unlikely that results farther down the list would be very relevant. So, only top 500 matches from each table were considered in computing the final score, after which the top 50 are returned.

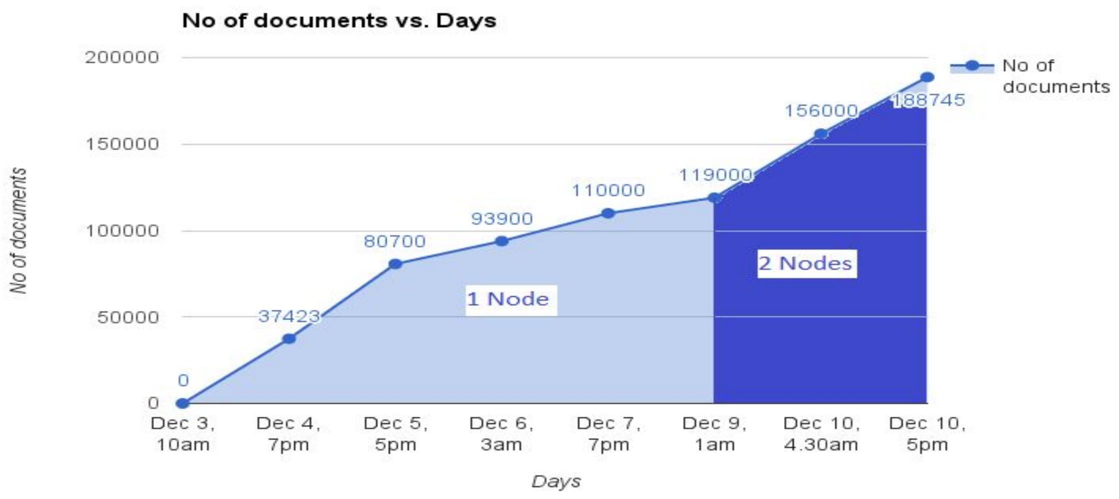
3. Evaluation

3.1. Crawler

3.1.1. Number of EC2 nodes

We crawled ~115K documents using one medium EC2 instance with one master and two workers, each with ~110 threads in around 3 days. For the remaining documents, we created another medium EC2 instance with the same master-worker setup and ran both in parallel for around a day. This gave us the remainder of our corpus. Each of the EC2 instances were started with seed urls in different domains, to eliminate overlap in crawled domains between the nodes.

3.1.2. Crawl Throughput



The chart above gives an approximate representation of the crawler throughput. We crawled around 60% of our corpus using 1 EC2 node with two worker instances with an average crawl rate of around 10K per worker per day. With two nodes, we averaged around 4000 documents per hour.

(Note: The crawler did not run continuously over the time range given above, as it was started and stopped multiple times to deal with bugs etc.)

3.1.3. Quality of Crawled Documents

We crawled a total of 188,745 documents from different domains including news, sports, entertainment etc. 99% of the crawled documents were in English, with the remaining few (~2000) being in Chinese, Japanese, French, Spanish etc.

3.2. Indexer

Following table shows the statistics for indexing 188,745 documents and writing the output directly to Amazon DynamoDB:

Indexer	No. of Mappers	No. of Reducers	No. of indexes	Size of index	DynamoDB Write Capacity	Build Time
Unigram	15	15	4,456,409	4.38 GB	800	4 hrs
Bigram	15	15	30,257,929	14.10 GB	800	15 hrs
Metadata	7	7	91,651	78.29 MB	500	24 mins

Following table shows the statistics for indexing 188,745 documents and storing the output to a S3 bucket:

Indexer	No. of Mappers	No. of Reducers	Build Time
Unigram	15	15	20 mins
Bigram	15	15	24 mins
Metadata	15	15	7 mins

As can be seen from the above tables, indexer spends majority of its time in writing the data to the DynamoDB tables.

3.3. PageRank

Following table shows the statistics for ranking 2380 hosts and writing the output directly to Amazon DynamoDB and running the job for fixed, 10 iterations:

Job	No. of Mappers	No. of Reducers	No. of hostnames	DynamoDB read/Write Capacity	Build Time
Pagerank	10	10	2,380	1000 (domains table/read) 500 (pagerank table/write)	15 minutes

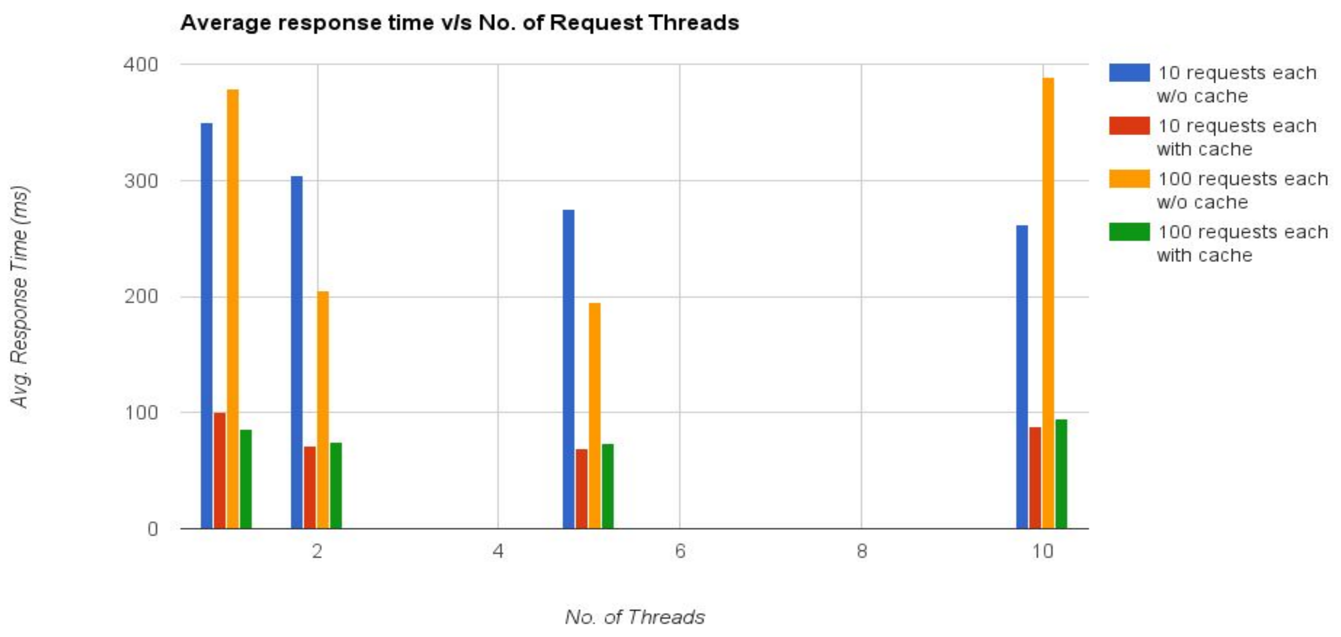
Following table shows the statistics for ranking 2380 hosts using a convergence of 0.001:

Indexer	No. of Mappers	No. of Reducers	Build Time
pagerank	10	10	7+ hours

We decided to go for fixed number of iterations as, we realized that running the job for 10 fixed iterations gave a clear indication towards the relative ranking of the hosts without the heavy overhead. While running the convergence based jobs, the system spent most of its time processing the mapreduce output from S3 and then moving it over to dynamo.

3.4. Search Engine

3.4.1. Response Time



3.4.2. Stress Test

The search engine was stress tested with 1000 threads sending 1 request each in parallel. The latency went up but all the requests were processed correctly aside from a few http exceptions.

3.4.3. Concurrent Requests

We tested concurrent requests with 500 threads each sending 10 requests. There were some HTTP exceptions but the job completed. Towards the end the latency became pretty high , around 4 seconds. We also tested with 1000 threads each sending 1 request. There were some HTTP exceptions but the job completed. The latency went up even higher , around 18 seconds. We believe, the latency rise is due to limitations in the number of threads spawned by Jetty and the runtime of each request.

3.5. Bottlenecks

3.5.1. Time taken to transfer indexed data to DynamoDB

3.6. Possible Improvements

Incremental Indexing

Including proximity score to compute results