CIS 555 Final Project: Google-Style Search Engine

Yukai Yang, Yimin Ji, Hongyu Zhang, Yuchen Ding
School of Engineering and Applied Science,
University of Pennsylvania
{yukaiy, sylviaji, hz53, ycding}@seas.upenn.edu

1. Introduction

1.1. Project Goals & High-level approach

The goal of this project is to build a Google-style cloud-based search engine composed of four major components: a web crawler, an indexer, the PageRank algorithm, and the final search engine and web user interface. Extended from previous homework in this class, the crawler will make use of the StormLite paradigm and S3/RDS cloud storage services. We will use the MapReduce paradigm to build our indexer, perform link analysis, and build the PageRank algorithm to rank indexed pages for our search engine results. Last but not least, we will connect every piece together and construct a single web search engine with Spark as the server-side framework and React as the client-side framework. While maintaining a high quality of code, we also ought to make sure the system is reliable, scalable, and well-functioning.

1.2. Milestones

- Milestone 1 (04/22) First, we work on the prototypes of each component (crawler, indexer, PageRank, search engine, and web user interface) based on our project architecture. Next, based on the crawler prototype, we continue developing and debugging the indexer and PageRank.

- Milestone 2 (04/29 - at the end of group check-in) We continue developing and debugging each component. At the same time, we try to integrate the output of the indexer and PageRank into the search engine and web user interface to get preliminary results.

- Milestone 3 (05/02 - code-complete deadline) We perform thorough testing to make sure all basic functionalities are working as expected. We plan to support extra features, like supporting additional types in crawler and indexer, providing location-specific search results, and integrating search results from Yelp or Facebook. We plan to improve the UI design of the web user interface. We will try to implement fault tolerance and AJAX support if time permits. In the end, we conduct the experimental analysis for evaluation and complete the final report.

1.3. Division of Labor

While each of us is responsible for one specific component, we will cooperate to integrate our components, perform debugging, conduct evaluation, and complete the final report together.

- Yukai Yang - Indexer, search engine back-end, ranking algorithms
- Yimin Ji - Web user interface, search engine News/Yelp API integration
- Hongyu Zhang - Web link graph, PageRank
- Yuchen Ding - Crawler

2. Project Architecture

The basic pipeline of the search engine is shown in Figure 1. The distributed crawler on EC2 first crawls around 500,000 documents and their metadata from the web, which are stored persistently on RDS. We then use an indexer to compute a document index for retrieval and build a web link graph for PageRank calculation. The indexing and PageRank results are stored in RDS. The web user interface is hosted on S3, and given a search query, it calls the REST APIs of the search engine server which is hosted on EC2. The server then retrieves the most relevant documents from RDS and returns them to the client as a JSON object, which is rendered in the browser.

3. Implementation: non-trivial details

3.1. Crawler

The Web crawler is extended from past assignments, and it is rebuilt to be distributed in order to meet the limitation

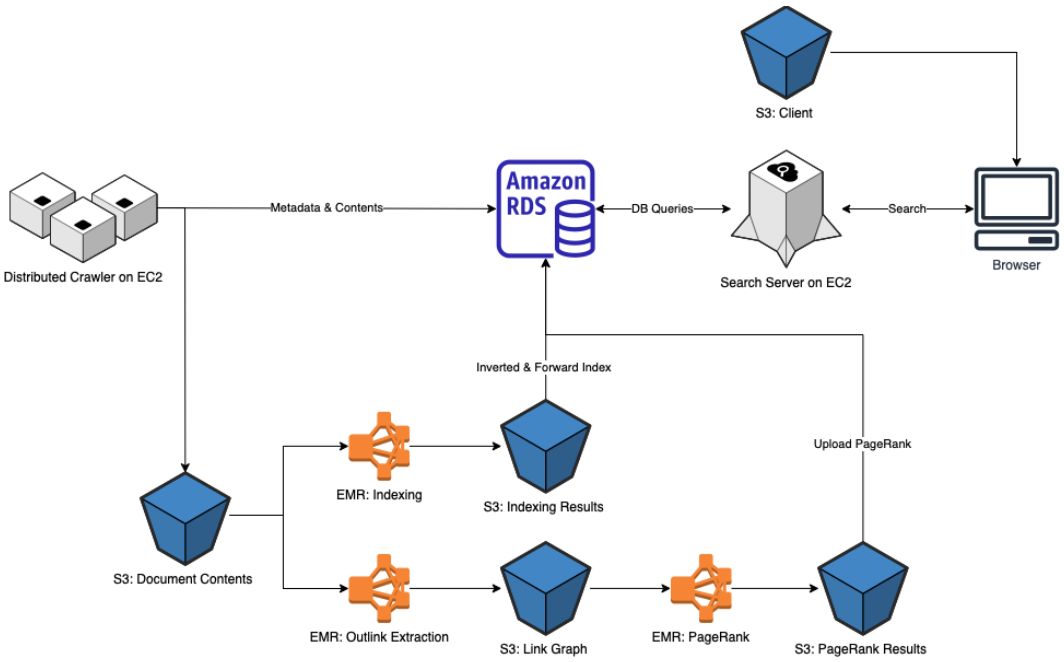

Figure 1. Search Engine Architecture

of time and resource. It is able to retrieve and parse HTML documents, while also following restrictions in robots.txt and well-behaving in terms of making distributed requests to different hosts. The crawled documents and metadata are stored on AWS RDS with a PostgreSQL database engine, and contents are also stored on AWS S3 for better access from EMR.

There are three major components in the crawler implementation, crawl master, StormLite, and storage. The master section sets up the StormLite topology, connects to databases, and initiate the crawling process. The StormLite component is modified from the homework implementation to make it multi-threaded. Each of the fetching bolts is responsible for a set of hostnames to avoid concurrently making multiple requests to one website. Also, the ability of parsing robots.txt is enhanced to adapt to the real-world web environment without using outside packages. As for the storage section, we have set up tables in RDS to record crawled HTML documents and associated information, and we have also made an extra copy in S3 bucket for performance. When crawling, the program is able to track visited pages and not index a page more than once. Beyond from that, we also keep timestamps in case of a crawled page is modified and we want to update the record in the database.

In the interest of getting to all of the regions in a timely fashion, we designed the task queue to be no longer than 100,000, so that the crawler would not go too deep in one domain. Moreover, the crawler is set to restart every 15 minutes in provision of a manual check or an interruption.

Furthermore, we have deployed the crawler on EC2 nodes with different sets of seed URLs. The evaluation of performance will be discussed in later section of this report, as well as tests on different numbers of threads.

3.2. Indexer

Our indexer is implemented as a Hadoop MapReduce job, which consists of a parser (mapper) and inverter (reducer). Each parser processes text in documents and emits a token stream. Text processing includes tokenization, normalization, and stemming with the Snowball Stemmer, which are implemented using Regex and OpenNLP[1]. During normalization, tokens are converted to lowercase, and emojis, URLs, emails, hashtags, phone numbers, diatrics, and repeated characters are removed. We also adopt a language detector to filter nonenglish documents. After tokenization, we remove tokens that only contain punctuations.

Inspired by Google's paper, our indexer aims to create an inverted index with hit lists containing positions of each term occurrence in the documents. Hit lists enable us to display excerpts and potentially support phrase search. To this end, the parser emits token position along with the document ID and the token itself. The emitted tuples are then partitioned by term and sorted by all three fields. Since extracting positions prohibits chunking individual files, we implement our own input format and record reader for reading the entire content as a single record in MapReduce. Lastly, an inverter is responsible for aggregating document frequency (DF), term frequency (TF), postings list, and hit

[1]https://opennlp.apache.org

lists for each term. However, hit lists also substantially expand our index storage. We use Berkeley DB to create an on-disk buffer for aggregating hit lists in each inverter. The output results are written in the following format:

```
term1,df:
<docId1,tf>
position1;
position2;
...
<docId2,tf>
position1;
...
term2,df:
...
```

The inverted index is then uploaded to RDS and stored as 3 separate tables, Lexicon, Posting, and Hit. Lexicon's columns include term ID, term, DF, and posting offset, a pointer into the Posting table that marks the beginning of its postings list. Similarly, Posting has document ID, TF, and hit offset as columns. We choose this scheme to minimize storage cost since no duplicate values are kept in the index. To support fast hit lookup, an additional forward index table is created from the inverted ones using SQL select statement. The forward index maps from document ID, term ID pair to hit offset and hit list size. Further, document lengths are also recorded in a standalone table for BM25 scoring.

3.3. PageRank

To compute the PageRank values, we first need to construct a web link graph to represent the linking information (out-links, out-degree, etc.) of crawled documents. We build the web link graph as a MapReduce job. Similar to indexer, we read the entire HTML content in each document as a single record using our custom input format. In our mapper, we query RDS to find the URL corresponding to the input file using JDBC Driver[2]. We use Jsoup[3] to extract anchor elements and convert to absolute URLs. This step also removes unneeded hyperlinks like emails and phone numbers. The mapper emits each <URL, out-link> and <out-link, ""> as intermediate key-value pairs to the reducer. In the reducer, for each key, we aggregate the URL and each out-link as a single string. We also sum the total number of out-links and append it to the string. And we append "true" or "false" to the string depending on whether the URL is in the database. Finally, we append an initial PageRank value 1.0 to the string. To avoid parsing errors, a unique separator "::??<2<spli75tt," is added each time we append to the string, and a unique delimiter "::spli75tt." is added to the end of the string. The reducer writes <NullWritable, Text of final string> to the output file. Each record in the output file is in this format:

```
URL<separator>out-link1<separator>
out-link2<separator>...<separator>
out-degree<separator>
true or false<separator>
1.0<delimiter>
```

Next, we implement PageRank as an iterative algorithm using Elastic MapReduce. As discussed in class, the PageRank is computed as follows:

$$PageRank^{(i)}(x) = \alpha \sum_{j \in B(x)} \frac{1}{N_j} PageRank^{(i-1)}(j) + \beta$$

, where $PageRank^{(i)}(x)$ denotes the rank of page x in iteration i, $B(x)$ is the back set of page x, N_j is the out-degree of page j, $\alpha = 0.85$, and $\beta = 0.15$. The damping factor is added in order to deal with sinks and hogs (self-loops). This formula suggests that we need to preserve the out-link information for each page and the output file format in each iteration. Since we do not have any out-link information for those pages that we have not crawled, these dangling links can not contribute to the rank values of other pages. To save time and computation, we ignore those dangling links. In our driver, for each iteration, we create a MapReduce job. To save space and also test for convergence, we only save the output files of the last 2 iterations. In our mapper, we extract the URL, out-links, out-degree, and previous rank value from the input value. We calculate the average vote it can send to its out-links and write this average vote to each out-link as key-value pair if the page has 1 or more out-links. Then, to preserve the out-link information, we write all the other information (URL, out-links, out-degree, etc.) to the current URL as key-value pair. In the reducer, we sum up the votes and apply the damping factor to get the final rank value. Then, we aggregate the information emitted from the mapper and the rank value together, separated by the separator defined above and add the delimiter defined above at the end. We write <NullWritable, Text of final string> to the output file and the format is preserved for the next iteration.

After computing PageRank, we use EMR to write the results to our database. In our mapper, we extract the rank value of the page. Using JDBC Driver, we query RDS to find the corresponding document ID and we insert the entry (document ID, rank value) to the table PageRank. The mapper emits no key-value pairs and the reducer does nothing.

Last but not least, we use EMR to view the total difference between the last 2 iterations to confirm that the PageRank values have converged. The first mapper emits the rank value of the previous iteration as value, and the second mapper emits the rank value of the latest iteration as value. The

[2]https://mvnrepository.com/artifact/org.postgresql/postgresql
[3]https://mvnrepository.com/artifact/org.jsoup/jsoup

reducer simply sums up the rank values of the previous iteration and the rank values of the latest iteration respectively and output these 2 numbers. In this way, we can compute the difference and check for convergence.

3.4. Search Engine

For document scoring, we incorporate Okapi BM25 and PageRank. BM25 improves upon TF-IDF by accounting for document length and term frequency saturation. It can be computed given a query q and document d as follows:

$$\text{BM25}(q,d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{(k+1) \cdot tf(t,d)}{k \cdot (1 - b + b \cdot \frac{|d|}{|d_{avg}|}) + tf(t,d)}$$

The scores are then combined as $\text{BM25} + \lambda \log(\text{PageRank})$, where $k = 1.2$, $b = 0.75$, and $\lambda = 0.5$. The log of PageRank is used because PageRank ranges from 0.15 to hundreds in our setting.

It is imperative for a search server to retrieve results with a latency as low as possible. A search engine that takes many seconds to respond is frustrating from a user's perspective. To this end, we batch SQL queries whenever possible and cache their results in memory.

We use HikariCP's database connection pool[4] in our Spark Java server to support concurrent requests. Upon receiving a query from the search route handler, the query is tokenized and processed the same way during indexing. We convert terms to term IDs and create a bag-of-words vector by counting unique terms. We then fetch term occurrences (DF and postings list) for all terms simultaneously. After computing a union of document IDs for all terms, we fetch metadata (PageRank and length) for all document IDs at once. To speed up scoring, documents with a PageRank less than or equal to a cutoff threshold of 0.16 are filtered. We use a cache size of 5,000 for term occurrences and 500,000 for document metadata. Because common terms like "the" are frequently searched among users and from query to query, these occurrences are likely to remain in the cache.

After scoring, top 100 matches are ranked efficiently using a priority queue, which is then sorted before returning to a user. Because extracting excerpts from document contents requires expensive text processing, we fetch URLs and contents for only 10 results at a time and produce excerpts lazily. The client has the option to request any page of 10 results out of 100.

The search engine also allows users to search for the latest news and nearby businesses. For the news search, the most relevant news articles published within the past month will be fetched from News API based on the user query. For the business search, the user's geolocation is determined on the client-side using IP-API, and nearby businesses will be fetched from Yelp Fusion API based on the user query.

[4]https://github.com/brettwooldridge/HikariCP

3.5. Web User Interface

The web user interface is built in React with MUI (Material-UI) as the component library. It has two main pages, the landing page and the results page, and routing is achieved using React Router. When a user visits `http://cis555app.s3-website-us-east-1.amazonaws.com`, they will see the landing page with a search input bar. After entering a search query and clicking on the search button, the user will be redirected to the results page, where the relative web pages will be displayed with their title, link and excerpt, sorted by the score computed by the ranking algorithm. The user can click on the "more" icon next to the title to see the details of the score. The user can also click on the "News" or "Businesses" tab to view integrated search results from News API and Yelp Fusion API. We append some screenshots of search results to the last page.

The web use interface is decomposed into the following collection of reusable components:

- **Search**: user input with autocomplete. To trigger a new search, click on the search icon, hit enter key, or select one of the suggested queries.
- **Index**: landing page with a logo and a Search component.
- **TabPanel**: container of results content based on the current tab selected.
- **Options**: component for switching between different search results (webpages, news, businesses).
- **Main**: results page with a Search component and an Options component.
- **Record**: general search result entry with title, path information, and excerpt with highlighted search terms.
- **About**: popup component with BM25 value, PageRank value, and overall score associated with each Record
- **Results**: container of general search results with pagination.
- **Article**: news search result entry with source, title, description, and date published.
- **News**: container of news search results with pagination.
- **Business**: Yelp search result entry with image, business name, rating, and general business information.
- **Businesses**: container of Yelp search results based on user's IP address with pagination.

For the autocomplete feature, we crawled and scraped an SEO website (https://www.mondovo.com/keywords) and collected more than 20K popular Google search terms from a wide range of categories. We filtered out inappropriate

queries using public available lists like google-profanity-words and human judgment. The remaining queries are used as the corpus for search suggestions.

To deploy the web application on AWS, an S3 bucket is created to host the client side code. The permissions and bucket policy are modified such that anyone with a Penn network IP address will be able to use the search engine. The server is also updated to enable Cross-Origin Resource Sharing (CORS) so that API calls can be made.

4. Evaluation

4.1. Crawler Throughput

The performance of our crawler scales with two crucial factors in implementation: number of nodes and number of threads. As we deploy the crawler on more EC2 nodes, throughput increases proportionally. As we increase the number of threads in StormLite, throughput also increases but with a bottleneck.

The baseline of comparisons is the number of successfully crawled document of a single-threaded crawler on local machine in a 15-minute interval. Details are shown in the chart below.

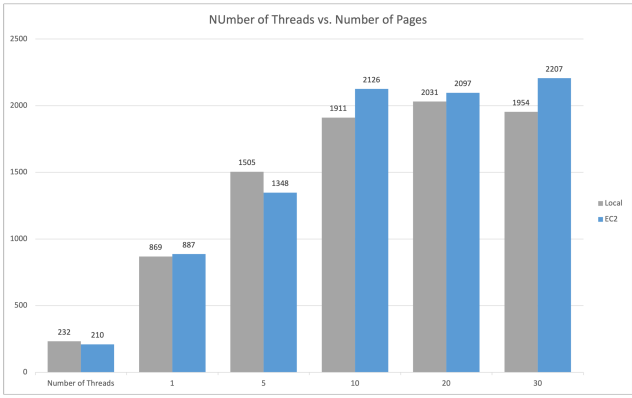

Figure 2. Number of Pages as a function of threads

As indicated in Figure 2, the capability of crawling has increases when we change the crawler from single-threaded to multi-threaded. Nonetheless, when we reach a pool size of 30, there is little marginal benefit in further increase. A possible explanation is that throughput is restricted by database access and internet bandwidth. Also, we have compared performance on local machine and on EC2, and we do not observe significant difference.

Additionally, we have performance evaluation on how number of nodes influences crawler throughput. There is a linear trend in performance when we deploy on more nodes.

Number of Nodes	Number of Pages
1	2207
2	4338

4.2. Indexer Performance

To efficiently access small documents in S3, we combine documents into chunks using Hadoop's CombinedFileInputFormat before feeding them into MapReduce.

We vary the number of cluster nodes (m5.xlarge) in EMR and run indexer on our corpus of 500,000 documents. In Figure 3, increasing from 2 nodes to 6 nodes significantly improves indexing performance, decreasing time taken by 2.8x. Further increasing nodes to 10 reduces time taken by 4.2x; performance gains are diminishing. As we increase the number of nodes, indexing workload likely becomes dominated by communication and Disk IO during the MapReduce shuffling phase.

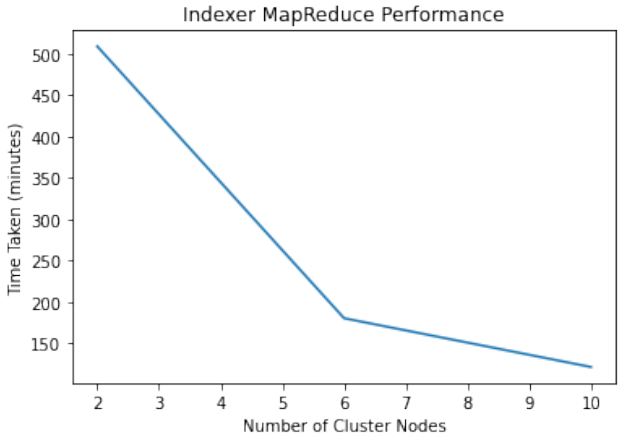

Figure 3. Time to run indexer as a function of nodes

4.3. PageRank Performance

The evaluation of our PageRank performance is divided into 2 components.

Firstly, we vary the number of input documents (293,000 vs 500,000) and compare the convergence rate. In this evaluation, we sum the PageRank values together and compute the total difference from previous iteration at iteration 10, 20, 30 and 40 for each input size. The results are shown in the table below and in Figure 4 for better visualization. As the input size increases by 1.7x, we can see that the difference also scales by that factor approximately. In both cases, we can see that our PageRank converges really fast under 40 iterations. For input size 293k, at 40 iterations, the total difference is merely 0.09. Considering the vast number of input documents, such a small difference suggests that our PageRank values have converged. Similarly, for input size 500k, at 40 iterations, the total difference is merely 0.14, which is safe for us to say that our PageRank values have converged after iteration 40. These results provides strong evidence that our PageRank is well-functioning and scales well over the input size.

Input Size	293k	500k
Iteration 10	626.51	1177.67
Iteration 20	15.82	29.83
Iteration 30	0.92	1.62
Iteration 40	0.09	0.14

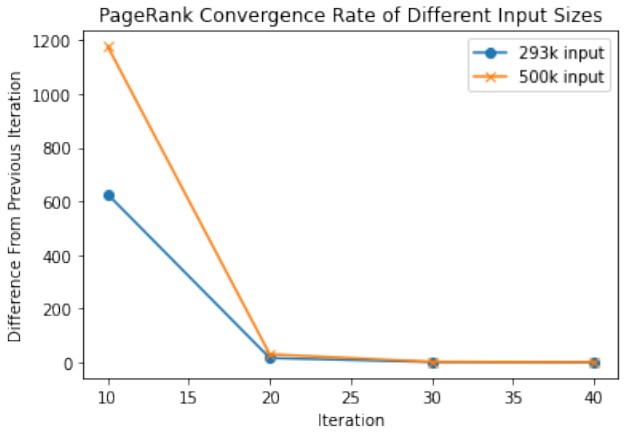

Figure 4. PageRank Convergence Rate on Different Input Sizes

Secondly, we vary the number of cluster nodes (m5.xlarge) in EMR and compare the average time spent on a single iteration of PageRank on input size 500k, as shown in Figure 5. Increasing from 2 nodes to 4 nodes improves the performance by 2x as expected. Increasing from 2 nodes to 6 nodes, however, only improves the performance by 2.7x. And increasing from 2 nodes to 10 nodes only improves the performance by 4.2x. Similar to the performance of our indexer, we can see that the performance gains are diminishing. This suggests that as we increase the number of nodes, we faces a bottleneck that limits the our PageRank performance, which is possibly due to Disk IO and communication in the MapReduce shuffling phase.

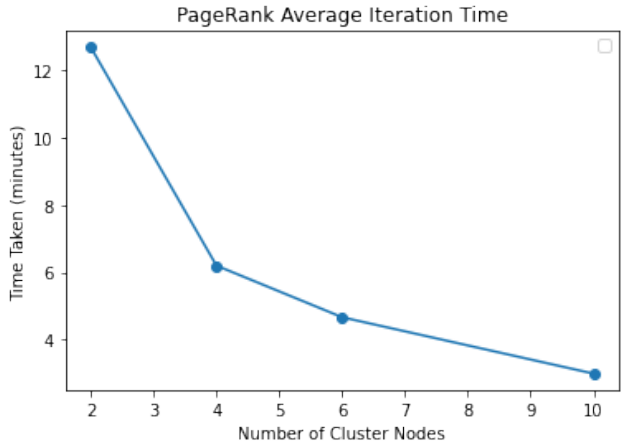

Figure 5. PageRank Average Iteration Time

4.4. Query Response Latency

To benchmark query response latency of the search server, we wrote a Python tool for submitting requests sampled from a list of 100 queries. To stress test the server, we use mostly multi-word queries up to 20 terms and vary the number of concurrent requests. We run the server on m5.xlarge with 4 vCPUs and 16 GB memory.

As shown in Figure 6, our server can handle up to 2 concurrent requests reasonably fast. On average, responding to 1 and 2 concurrent requests takes 756 ms and 898 ms, respectively. Our SQL batching and caching strategies pay off. As the number of concurrent queries increase, the respond time grows almost linearly. With more than 4 concurrent requests, the average latency rises above 1.5 seconds; BM25 and excerpt extraction become the bottleneck. The m5.xlarge instance with only 4 cores is unable to keep up with these CPU-intensive operations. Increasing the number of cores or distributing requests across multiple server nodes are potential ways to scale to large number of requests. Due to limited resources, we are unable to experiment with these options.

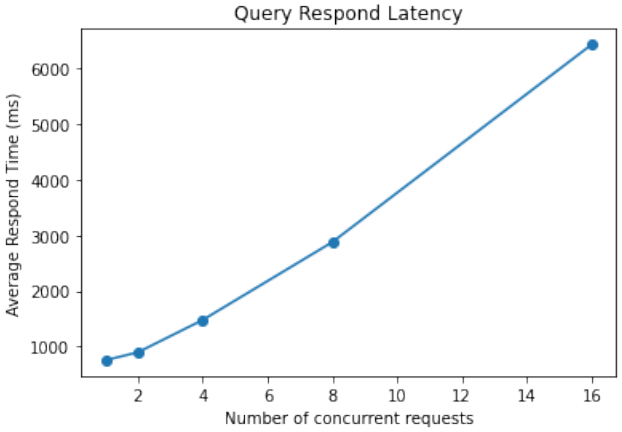

Figure 6. Response latency as a function of concurrent requests

5. Conclusions

In this project, we build a Google-style cloud-based search engine and evaluate its performance. The crawler adopts the StormLite paradigm and crawls half million web documents. The indexer builds an positional inverted index using MapReduce. The PageRank constructs the web link graph and we build and run the iterative PageRank algorithm with damping factor using MapReduce. Our search engine incorporates improved TF-IDF (BM25) and PageRank to compute the final ranking. Our web UI supports search auto-complete, shows excerpts with highlighted hits on the search result page, and integrates search results from News and Yelp webservices. The entire project is cloud-

based and after thorough testing and performance evaluation, we have confidence in our search engine that it is well-functioning, reliable and scalable.

Future work still lies ahead to further improve our search engine. For our indexer, we plan to support indexing title and anchor text. And we want to incorporate the matrix representation of PageRank algorithm to our MapReduce paradigm and explore other effective ways to deal with dangling links, sinks and self loops for our PageRank. For our search engine, we plan to tokenize content when extracting excerpt is expensive and cache tokenized content in memory or store in RDS to reduce latency.

Acknowledgment

We would like to thank Professor Zachary Ives and all teaching assistants of CIS 455/555 Internet and Web Systems for their support and guidance.

Not Secure — cis555app.s3-website-us-east-1.amazonaws.com

Penn

PageRank

All | News | Businesses

en.wikipedia.org › wiki › PageRank

PageRank - Wikipedia

PageRank - Wikipedia **PageRank** From Wikipedia, the free encyclopedia Jump to navigation Jump to search Algorithmic used by Google Search to rank web pages Mathematical **PageRanks** for a simple ...

www.wikidata.org › wiki › Special:EntityPage › Q184316

PageRank - Wikidata

PageRank - Wikidata **PageRank** (Q184316) From Wikidata Jump to navigation Jump to search algorithm for calculating the authority of a web page based on link structure. PR ...

web.archive.org › web › 20160703031514 › http: › › searchengineland.com › what-is-google-pagerank-a-guide-for-searchers-webmasters-11068

What Is Google PageRank? A Guide For Searchers & Webmasters

What Is Google **PageRank**? A Guide For Searchers & Webmasters 808 captures 25 Dec 2008 - 19 Apr 2022 May JUL Aug 03 2015 2016 2017 success fail About ...

en.wikiquote.org › wiki › PageRank

PageRank - Wikiquote

PageRank - Wikiquote **PageRank** From Wikiquote Jump to navigation Jump to search **PageRank** (PR) is an algorithm used by Google's search engine to rank web pages. The ...

doi.org › 10.1093%2Fbioinformatics%2Fbtq680

When the Web meets the cell: using personalized PageRank for analyzing protein interaction networks | Bioinformatics | Oxford Academic

the cell: using personalized **PageRank** for analyzing protein interaction networks | Bioinformatics |

Not Secure — cis555app.s3-website-us-east-1.amazonaws.com

Penn

DuckDuckGo

All | News | Businesses

Engadget

DuckDuckGo removes search results for major pirate websites

DuckDuckGo's crackdown on dodgy content now extends to digital bootleggers. TorrentFreak has discovered that the search engine no longer lis...

Apr 15, 2022

The Verge

DuckDuckGo insists it didn't 'purge' piracy sites from search results

There were reports that privacy-focused search engine DuckDuckGo may be suppressing search results from piracy sites like The Pirate Bay. Th...

Apr 18, 2022

Gizmodo.com

DuckDuckGo's Super Private Browser to the Rescue (For Mac)

Data privacy company DuckDuckGo has released its super secure web browser in beta for Mac—finally. The move comes years after the company re...

Apr 12, 2022

Not Secure — cis555app.s3-website-us-east-1.amazonaws.com

Penn

taco

All | News | Businesses

Don Barriga Mexican Grill

4.5 · 112 reviews

Mexican
Service options: pickup · delivery
Address: 4443 Spruce St, Fl 1st, Philadelphia, PA 19104
Phone: (267) 292-5741

TacoTaco Mexican

4.5 · 52 reviews

Mexican
Service options: pickup · delivery
Address: 261 S 44th St, Philadelphia, PA 19104
Phone: (215) 921-2140

Tacos Don Memo

4.5 · 204 reviews

$ · Mexican, Food Trucks
Service options:
Address: 270 S 38th St, Philadelphia, PA 19104
Phone: (610) 529-2039

Rosy's Taco Bar

4.0 · 300 reviews

$$ · Tacos
Service options: pickup · delivery
Address: 2220 Walnut St, Philadelphia, PA 19103
Phone: (267) 858-4561

Taqueria Morales

4.5 · 62 reviews

Mexican
Service options: pickup · delivery
Address: 1429 Jackson St, Philadelphia, PA 19145
Phone: (215) 645-9392

Distrito

4.5 · 1093 reviews

$$ · Mexican, Cocktail Bars
Service options: pickup · delivery
Address: 3945 Chestnut St, Philadelphia, PA 19104
Phone: (215) 222-1657

El Taco

4.0 · 73 reviews

$ · Mexican
Service options: delivery
Address: 3233 Powelton Ave, Philadelphia, PA 19104
Phone: (267) 820-8226

El Rincon Latino

5.0 · 2 reviews

Latin American, Mexican
Service options: pickup · delivery
Address: 469 N 10th St, Philadelphia, PA 19123
Phone: (215) 982-1082