PHIIGO - A Distributed Web Crawler

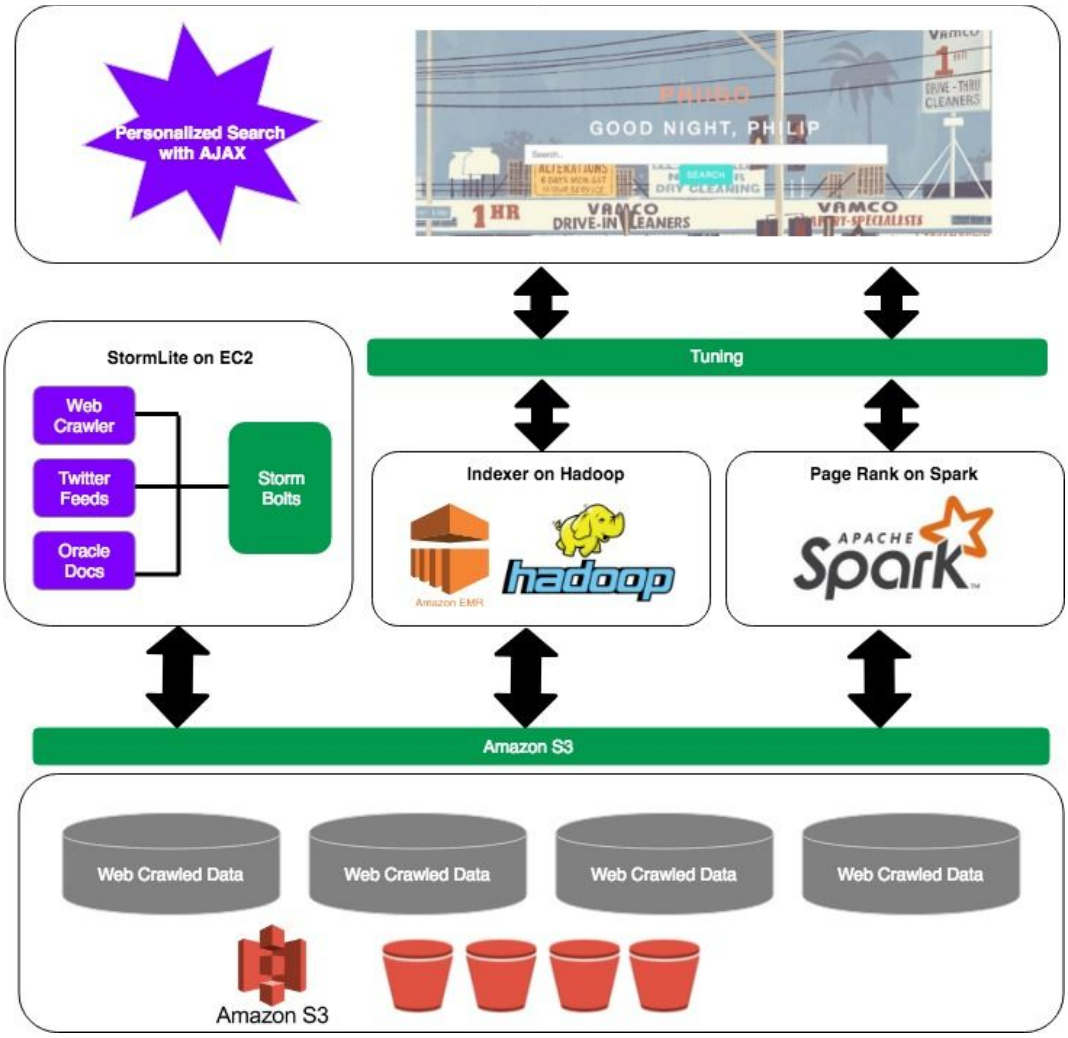
CIS 555 Final Project
Group 2: Yecheng Yang, Peng Yan, Hugang Yu, Linghan Zheng

I. Introduction

This project aims to construct a large-scale distributed system that allows efficient and comprehensive search functionality. The system is powered by four main components: crawler, indexer, page-rank calculation and search engine which builds upon Stromlite, Hadoop and Spark frameworks respectively. One crucial aspect of the system is to exploit parallelism to achieve maximum efficiency while assuring search with high accuracy through inverted indexing and page-rank calculations.

II. Architecture & Implementation

On the architecture level, the crawler reads inputs from millions of web pages and save page

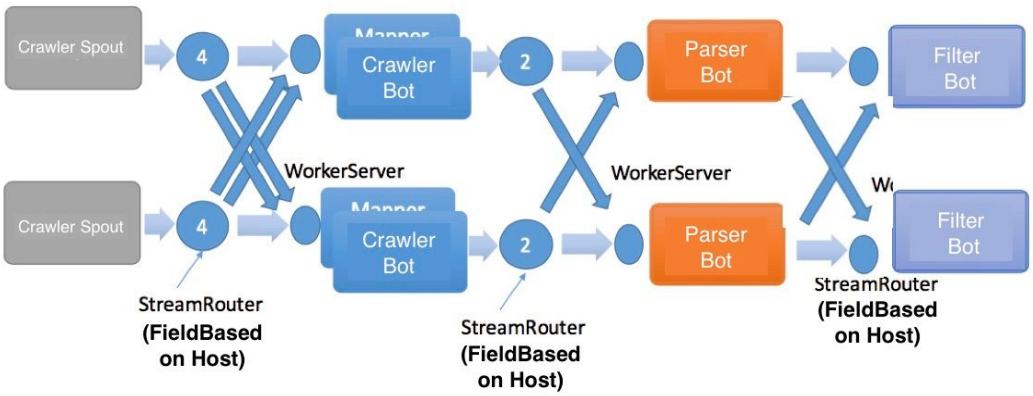


contents with URL to S3 instance on AWS. Later, the indexer and page-rank independently iterate through files to accumulate information necessary for their own purposes. The results of indexer and page-rank calculation are then stored into separate tables of the same MySQL instance, which is again hosted on AWS. Lastly, after users submit queries through our web interface, the search engine will convert them into SQL queries and return ones with the highest scores. The architecture of our solution is highly scalable since every single component is distributed.

**Please refer to subsections for implementation details and performance analysis.*

i.Crawler

Aiming to maximize efficiency of the system, we implemented a Mercator-style crawler on Stormlite framework. Each worker is hosted on EC2 by AWS. There is a separate master node which is responsible for initiating all crawling jobs and pausing, ending and monitoring every ongoing job.



On a high level, the crawler has four components, where are crawler spout, crawler bolt, parser bolt and filter bolt. There are a few improvements in the system. Spout reads from a URL queue which will be periodically saved to the disk. Upon initialization of a job, states about the URL queue can be restored from the previous queue if necessary. If the crawler is paused or stopped for any reason, the remaining URL in the queue will be also saved to disk. In addition, Only parser bolt will have access to Berkeley DB to maximize throughput. Apart from 200 response, the crawler handles all 300 responses well and will skip pages that returns 400/500 responses. In addition, each tuple is passed to next bolt by field hostname, which is the same for each domain. The idea is to keep all URL's from the same host to help reduce redundancy since different workers no longer need to download and store robots.txt as well as last accessed time for each domain. When new URL's are extracted from a page, they are passed to the filter bolt based on its own hostname and thus they will be sent to the filter bolt (and worker) responsible for the hostname. From there, these URL's will be put into the correct URL queue.

When we actually ran the crawler on EC2 instance, we noticed the speed of crawler was reducing over time. Specifically, the number of crawled pages decreases with more hours into the program. In addition, there is sporadic termination of the program as time accumulates. To address such problems, we, firstly, periodically cleared the collections of robots.txt for memory issues and secondly, enabled a periodical uploading of files on S3. The change in performance was quite clear as shown above.

Bonus Features

1. Content seen: one important feature of Mercator is added in our implementation to reduce disk space utilization. This feature uses SHA-256 hashing on each of the pages downloaded and compares with hash values of previously stored page to prevent multiple copies of the same file on disk.

2. Extra types - PDF files: it became clear to us that PDF files will likely include important information such as handouts, user guide and etc. Thus, the system will download PDF files in addition to others to provide extra information for the query.

ii. Indexer

Data Structures

In our inverted index job, we designed some “Writable” structures as below:

HitWritable: the data structure that represents the information about one occurrence of one word in a document. It contains fields including: *isCapital* – whether the word occurs as all uppercase form; *isTitle* – whether the word occurs within <title> element of the document; *isWithLink* – whether the word occurs within <a href=”…”> element of the document; *isEmphasis* – whether within <b><i><mark> element; *isMetaTags* – <meta name=”description”> or <meta name=”keywords”> element; *isTitle* – <title> element; *headingNo* – for <h[1-6]> element, record the number; otherwise, 0; *position* – the approximate position or the occurrence.

WordUrlWritable: the data structure that contains a pair of word and url. The intermediate <key, value> pair of the inverted index job is <*WordUrlWritable, HitWritable*>.

CountWritable: this is the data structure that used to count some scores of a <word, url> pair. It contains fields that are closely related to HitWritable: *termFreq (TF*) – the total number of occurrences of the word in this document; *capitalPer* – the percentage of occurrences that are in uppercase form; *titlePer, linkPer, emphaPer* and *metaPer* – the percentage of occurrences with corresponding flag set true;*headingScore*

– we computed this score by assigning <h1> a weight of 1.0, <h2> 0.8, <h3> 0.6, <h4> 0.5, <h5> 0.4 and <h6> 0.3; *positionScore* – we computed this score by assigning the first 500 words a weight of 0.8, the next 500 words 0.5, and the rest 0.2; *First five positions* – we also recorded the first five occurrence positions of a word in a document in order to implement phase searching such as “computer science”.

Document Parsing

The parsing step is used to split document into words, and compute a “HitWritable” for each specific occurrence. We used Jsoup library to parse the html document. Then we traversed each parsed element: for important element, such as <title> or <meta>, we would set corroding Hit field true, which may lead to higher score in later computation.

We used Apache OpenNLP (https://opennlp.apache.org/) to tokenize and stem words, so that each document stream can be chop into separate words and derivationally related words can be stemmed into same format. We created a stop word lists based on (https://kb.yoast.com/kb/list-stop-words/). As for numbers, we kept records of all numbers between 1000 and 2500, which are more likely to refer to years; while for numbers below 1000, we only recorded numbers with specific meanings such as “911”, “76” and “666”, and we just ignore numbers larger than 2500.

Map Reduce

Master: The job of master is to get input and output paths from arguments, split input documents into map tasks and assign jobs to workers.

Mapper: The input of mapper is a document tagged with its url. Then, the mapper will parse the document, extract useful words and emit <WordUrlWritable, HitWritable> information to reducers.

Reducer: The reducer will analyze <WordUrlWritable, iterator[HitWritable]> input and compute CountWritable information for the specific <word, url> pair.

We implemented the map-reduce job in Hadoop framework and used AWS EMR service to run the job. The job took 593 large files to process and them split these files into 592632 independent documents as map inputs.

Bonus Features

1. ***Off-content information:*** We took the context of the word into consideration, such as <title> and other tags. We also extracted words from <meta name="description"> and <meta name="keyword"> to get important metadata of the document.

iii. Page Rank

Parameter Setting

Our PageRank algorithm runs over 60 million edges graph and getting a 400 thousand node pagerank under spark environment with 10 iterations. The run time is 1min 53 seconds. To alleviate the pain brought by lots of PageRank sinks, we assume that every pagerank sink points to every node in the graph, thus when updating pageranks, that portion is a Constant and easy to compute.

Bonus Features

1. ***Spark:*** Our PageRank algorithm runs under Apache Spark environment.

2. Sparse Matrix Multiplication: With Sparse Matrix Multiplication technique, we werea able to gain a 100 times faster performance boost.
3. Crawled URLS: Not only using links of crawled page, but also utilizing 60 million queued links in crawler, which expanded the PageRank input data by 100 times thus getting a more accurate result.
4. Domain-level Rank: We only rank domain names but not the exact url, which under our setting making the pagerank more reliable.
5. Caching: Cached Pagerank Result in a SQL Database to boost performance from roughly a few seconds (3 - 5 seconds) to a blink of eye.

iv. Search Engine

Front-end Structure

Used servlets, Bootstrap, and Jetty to implement the front-end of the search Engine. Structure-wise, we have a login page, a register page, a home search page, and a result showing page with a search bar on the top of the page. Each page communicate with one another through GET and POST requests, and the data transfer is backed by servlets.

Some Design Details

1. Have a page navigation bar at the bottom of the page. The bottom showing the current page number would be unclickable.
2. Each search result section has three components: a result title containing the hostname and the matching words, a url of the page, and the major calculated tuning parameters including Total Score, Neighbor Score, and Page Rank. And the coloring of these components are purposefully matched with the color scheme of Google.

The equation used is as following:

```
if(historySet.contains(getHost(r.url)))
        r.score = r.score * USER_PROFILE_WEIGHT;
      r.score = r.score + r.score * r.neighborScore * WORD_NEIGHBOR_WEIGHT;
      r.score = r.score + r.score * (float)Math.log(2+r.pageRank) * PAGE_RANK_WEIGHT;
```

Tuning

We used more than ten tuning parameters calculated in the Indexing step to fine tuning our search results. These parameters include:

1. Final score: a score calculated using these parameters which is used to rank the page
2. Page rank: page rank score
3. Neighbor score: a score based on the averaged distance between the query words in all the documents they all appear.
4. Related words: the query words that appear in a result page
5. TF: term frequency
6. IsCapital: whether the word is in capital in the page
7. IsTile: whether the word is in the title of the page
8. IsInLink: whether the word is in the link of the page
9. IsEmphasized: whether the word is emphasized in the page
10. Meta Percentage: how many of the matched words are in the meta data

11. Heading Score: a score calculated on the number of times the words appears in a title.
12. Position Score: a score calculated on the relative position of a word in a page

With these parameters, we came up with the following ideas:

1. A page with a url containing the query words are more likely to be official websites, such as www.apple.com.
2. A page with a title containing the query words is a clear indication that the page is about the requesting query.
3. Since there is significant discrepancy between the TF and PageRank between the pages, we reduced the weight of them by a log factor.

And here is the final equation:

```
preparedStatement = connect.prepareStatement("SELECT *,
((LOG(2+tf)*6+headingScore+5*positionScore+" +String.valueOf(TITLE_IMPORTANCE)+
")*((2*capitalPercent+2*titlePercent+5*(url like '%"+word+"%') " + "+20*(url like
'%"+word2+"%') "+"+10*(ur
like'%"+word+".%')+2*emphasisPercent+2*metaPercent)+"+String.valueOf(TF_WEIGHT)+ ")) as
scoreSum from Indexing where word = ? ORDER BY scoreSum DESC LIMIT
"+String.valueOf(400));preparedStatement.setString(1, word);
```

Bonus Features

1. JQuery & AJAX: We implemented autocomplete search bar using JQuery and Ajax. The words source file contains 1000 common English words. We pre-loaded the 1000 words into the html file to reduce the autocomplete time.

2. Serving User Specific Content: Implemented a user login system to store the browsing history of users in mySQL. We keep track of the hostnames a user has visited and boost the the score of the websites with those hostnames accordingly. Such adjustment is done on each query request.

3. MySql: We also applied MySQL to Boost Search Speed. Stored the output and also some pre-calculations into a SQL database, There are more than 200 million records in the Database with index set up.

III. Conclusions

Overall, this has been a successful project for all of us. We were able to integrate knowledge from the course and beyond. We were able to practice our knowledge in a situation very close to real life. Each of us completed the project under expectation and the search engine ran well. If only we had more time in the end, it would be a good idea to include images for queries. If there is something we want to improve in the future, we could have put more details into the indexer and further extract information to optimize the queries as well support images.

IV. Division of Labor

Crawler & Database: Yecheng Yang Indexer: Linghan Zheng
PageRank: Huguang Yu Search Engine and Web User Interface: Peng Yan