junosearch: A scalable and fully functional search engine

Alex Zhao (alexzhao), Akash Subramanian (akashsub) , Victoria Xiao (xiaov), Wai Wu (wuwc)
University of Pennsylvania, CIS, School of Engineering and Applied Science
(@seas.upenn.edu)

Abstract - In the spring of 2017, our team of 4 set out to design a scalable and distributed search engine as our final project for CIS 455/555 at the University of Pennsylvania. We pooled our knowledge about the Mercator crawler, Google Page Rank, Map Reduce, TF-IDF, Ranking, and experimented with technologies such as AWS to fulfill this goal. We present to you, ***junosearch*** *(www.junosearch.fun).*

1 Introduction

1.1 Project Goals

The ultimate goal of this project is to have a fun, rewarding, and edifying experience while developing a scalable, distributed crawler and search engine. We seek to attain this goal via two fronts: achieve mastery of core technical concepts and competence, as well as develop project planning, iterative development and teamwork skills.

On the technical front, we seek to achieve competence in large scale system design, as well as gain experience with distributed crawler, indexing, storage, and ranking design and implementation. We seek to also familiarize ourselves with multiple relevant web technologies, chief among them Amazon's EC2, S3, DynamoDB, and EMR. Our personal goals include develop expertise in at least one core technology and be exposed to working with others, for a well rounded educational experienece.

1.2 Approach

We knew that scalability and integration were key issues we needed to tackle effectively for this to be a successful project. Thus from the onset, we agreed to respect key interfaces that allow for scalability when designing, communicate often through Slack, and have MVP's as early as possible to conduct iterative testing.

1.3 Timeline

April 20th Basic version of crawler complete. Crawled 500 documents. Design of distributed crawler begins.
May 1st Completed interface for each modular component. Begin Indexer.
May 3rd Basic Distributed Web crawler complete. Crawled 100K documents.
May 4th Completed basic integratable components. Begin integration tests. Begin writing search engine and ranking algorithm. Crawled 10K documents.
May 5th All individual components working. Crawled 300K documents. Continued testing.
May 6th Started producing and processing test crawl data. Basic search engine completed. Crawled 900K documents and indexed 400K documents.
May 8th Feature freeze. Finished integrating all components of the project. Dugging and twek ui in prepration for demo.
May 9th Completed project. Demo. Complete report.

1.4 Division of Labor:

What we had planned: **Cawler:**Alex (Primary), Wai (Secondary). **Indexer:** Vicky (Primary), Akash (Secondary). **Page Rank:** Wai (Primary), Alex (Secondary). **Search Engine:** Akash (Primary), All (Secondary). **Project Report:** All.

In reality we developed in a highly integrated manner. Alex and Vicky worked on the Crawler, Vicky and Akash worked on indexer, Wai took the lead on page rank, Akash developed the backend of search engine and searching function, Alex took responsibility for developing the front end, preparing architecture diagrams, and integrating the api hooks developed by Vicky, while Wai and Akash took care of a lot of the database operations.

2 Architecture

On a high level, this is quite a modular project (See Fig. 1 for architecture diagram). The crawler is a fully polite, distributed multi-threaded mercator style web crawler that ran on multiple EC2 instances. We then use EMR to run PageRank and TFIDF jobs in order to build the index, with final results stored in DynamoDb. The ranking function incorporates weighted average of TF-IDF and PageRank. Finally, a servlet backed frontend fulfills user requests by querying the index on the backend and and serving the top n results to the user. The webapp is deployed on both Elastic Beanstalk and

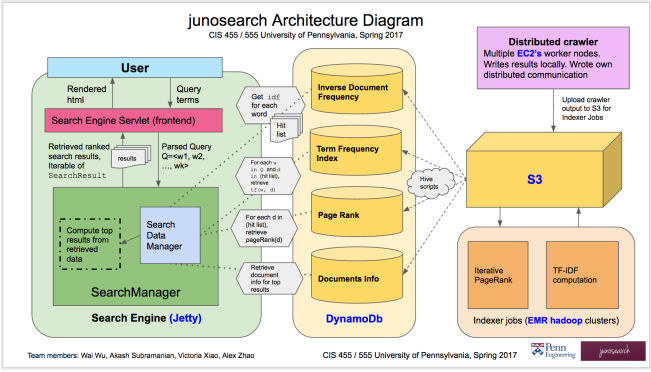


Fig. 1. Overall architecture of the search engine. Larger version may be found in the appendix

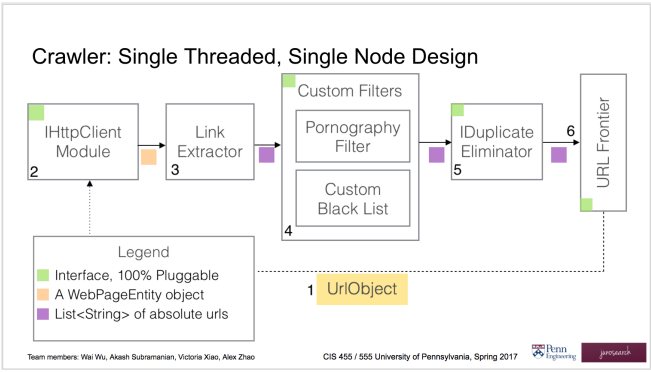


Fig. 2. Single Threaded Crawler Architecture

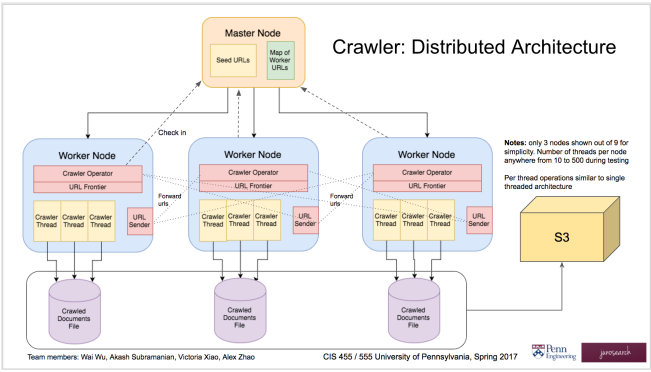


Fig. 3. Distributed Crawler Architecture

on heroku, the latter of which is linked to the domain name "www.junosearch.fun".

2.1 Crawler

Single Threaded Crawler: Fig. 2 The first iteration of the crawler was a modular and polite single threaded crawler.

Distributed Crawler: Fig. 3. The final iteration of the web crawler is a Mercator-style, distributed system that consists of 9 worker nodes and one master node, all running on Amazon EC2. On a high level, the worker nodes consists of 500 worker threads, each of which retrieves documents from the web, parses all URL links from retrieved documents, and sends retrieved URLs to the correct node. URLs are distributed amongst nodes based on a hash function and are sent between worker nodes through an HTTP messaging system similar to the one used in HW3. The distributed communication was written from scratch using spark web framework.

Upon crawler startup, the master assigns each worker node with its hash value, sends a ⟨ hash, URL ⟩ mapping of all workers to each worker in the system, and initializes the crawl by distributing a list of seed URLs amongst the worker nodes. Each worker node consist of:

- **WorkerServer** - Initializes the CrawlerOperator. Contains Spark Java ports that listen for URL messages, and enqueues received URLs to the CrawlerOperators frontier queue.
- **CrawlerOperator** - Maintains a blocking URL frontier queue that is shared by all threads on the node. Spawns and coordinates the activity of 500 CrawlerThreads. Also spawns a thread that periodically checks for and breaks frontier queue deadlocks.
- **CrawlerThread** - Contains the actual crawling functionality of the crawler.

Each **CrawlerThread** consists of:

- An **HttpClientModule** that dequeues a URL from the URL frontier and fetches the web document corresponding to the URL. Tunable politeness guarantees. Contains a modular **HttpClient**, a robust client that executes a request and returns a response. Contains LRU cache to store robots.txt info. The biggest challenge since HW2 was refactoring the module to be more robust and stable, including introducing tunable politeness guarantees, handling internal and permanent redirects, and If-Not-Modified responses (introducing the idea of history into URL Objects).
- A **LinkExtractor** that uses Jsoup to retrieve a list of URL links from the retrieved web document. A number of pluggable filters to remove unwanted URLs from the list. For the demo, a PornographyFilter was used to filter out a list of 150 pornographic websites.
- A **Duplicate eliminator**. First iteration queried BerkeleyDB but that quickly became a bottleneck. Reimplemented using in memory, Rabin fingerprint based Hash Table.
- A **FileAccessor** that writes retrieved documents to a .txt file on disk.
- A **URLSender** that determines the hashed value of each URL on the list and forwards them to the appropriate crawler node.

The Crawler writes locally to txt files with the format: ⟨ URL: String, Contents: String, Outlinks: List⟨String ⟩ ⟩

Choosing Seed URL's: The top 500 URLs on Alexa's Top 1 Million Websites list were used as seed URLs for the crawler, with pornographic links manually removed.After crawling is completed, .txt documents consisting of ⟨ URL,

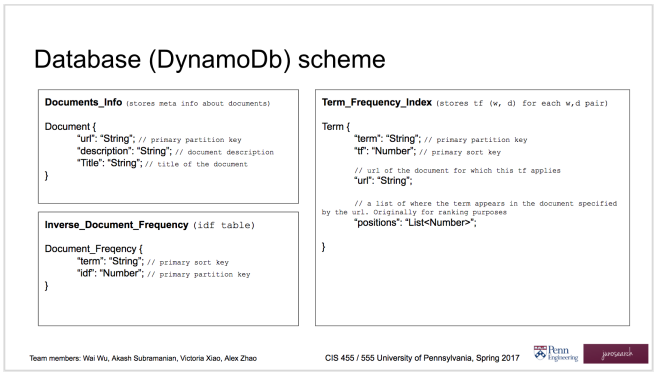


Fig. 4. DynamoDB database Schema

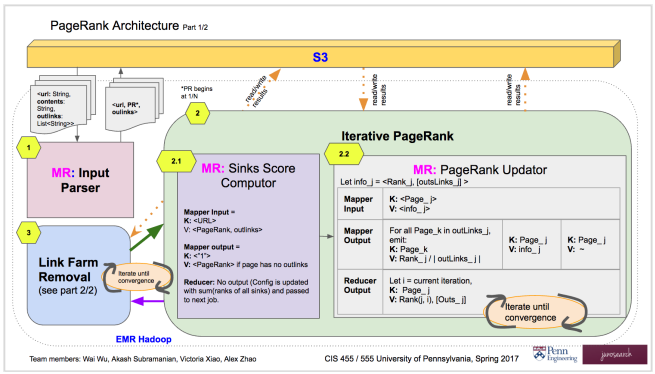


Fig. 5. Page Rank Overall Architecture

web document, list of outlink ⟩ tuples were uploaded to S3 to be used as input for the Indexer and PageRank.

Results: The corpus for the demo consists of ˜900,000 web documents, which were retrieved from 21 hours of crawling, which includes stopping and restarting along the way.

2.2 Indexer

The Indexer consists of 3 MapReduce jobs that are run using Hadoop and Amazon's Elastic MapReduce framework. The final result was run on a cluster of 10 M4.xLarge EC2 instances. The MapReduce jobs are as follows:

- **Term Frequency (TF) MapReduce** - This job takes in the crawldata .txt documents produced by the crawler and emits tuples in the format ⟨ term, URL, tf, position ⟩, where: *term* is a word in the corpus *URL* is the word of the web document that the term appears in *tf* is a weighted count of the number of times each term appears in the document. The weight of each occurrence of a term differs depending on the place in the document that it appears in. Options for term placement included: within a paragraph, within a header, within a title, within the URL, within metadata tags. *position* is a list of positions in the document where the term appears. Care was taken to split the terms properly during this phase.
- **Document frequency MapReduce** This job takes in the output of the TF MapReduce and emits tuples in the format ⟨ term, document frequency ⟩, where document frequency is the number of documents in which the term appears.
- **Page properties MapReduce** This job takes in the crawldata .txt documents produced by the crawler. It emits tuples in the format ⟨ URL, document title, document description ⟩, where title and description are extracted from metadata tags in the document body.

The results of all 3 MapReduce jobs are emitted to S3. A Hive script is then used to transfer the results to DynamoDB. The schema of the database is shown in Fig. 4.

2.3 Page Rank

The PageRank algorithm we used is an extension of the basic PageRank websurfer model. The page rank of page x is given by:

$$PR(x) = \frac{1-\lambda}{N} + \lambda \cdot \sum_{y \to x} \frac{PR(y)}{|outsNeighbor(y)|} + \lambda \cdot \sum_{z \to \emptyset} \frac{PR(z)}{N}$$

And consists of the following three parts:

1. Contribution from websurfer teleporting to random webpage, with probability $1-\lambda$
2. Contribution from websurfer randomly picking links to follow with probability λ
3. Contribution from sinks with no outgoing links, whose page rank is spread out across the rest of the nodes.

We made a few modifications to improve the accuracy of page rank:

Dealing with Sinks (No outgoing links) For sinks, we deviated from the conventional approach of treating them as though they have outgoing links to all other nodes in the graph. We felt that would create a very dense graph with huge sets of links, and slow down the runtime of our pagerank algorithm to $O(N^2)$ per iteration. Instead, we chose to first run a separate mapreduce job that summed up total ranks of all the sinks, and passed it to the pagerank mapreduce job through a shared Configuration object, Fig. 5 phase 2.1.

Self Loops For self loops, we removed them from the total count of outgoing links, to prevent hogging of rank.

Dealing with Nodes with no incoming links: These nodes would not be part of the intermediate keys passed to the reducer in Fig. 5. phase 2.2, because the conventional algorithm loops through the outlinks of the page and emits ⟨ Outlinked Page, Rank Contribution ⟩. So in the mapper phase, we also emitted keys of the form ⟨ Current Page, Page Description⟩.

Outgoing link to non-existent node: We also dealt with pagelinks to nodes not crawled or non-existent by emitting a tuple of ⟨Current Page, ˜⟩ in the PageRank map phase, so that in the reduce phase, the node is ignored if " ˜ " is not

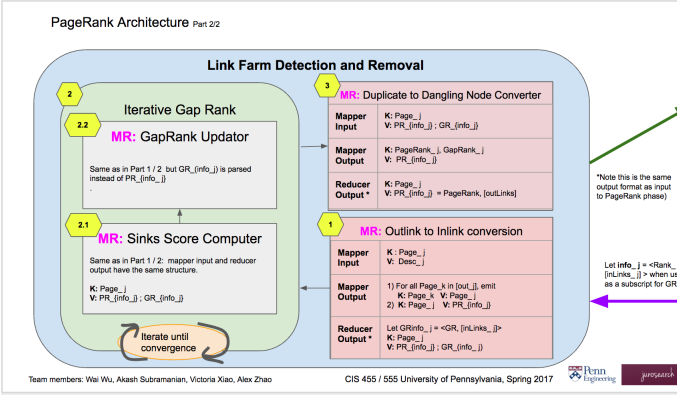


Fig. 6. Page Rank Extra Credit: Link Farm detection, Figure 5 part 3 in detail.

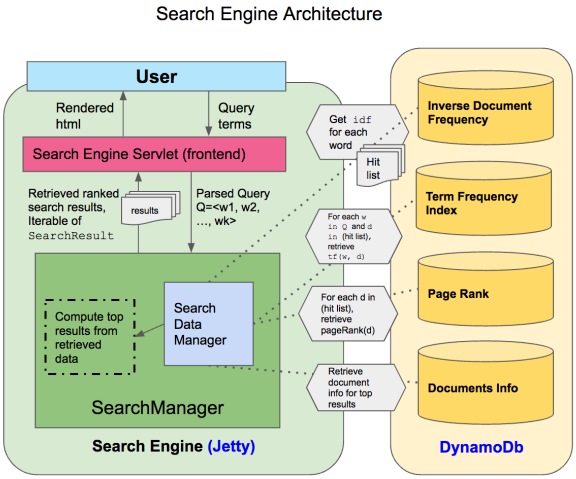


Fig. 7. Search Engine Architecture

part of the set of values.

Link Fram Detection (EC) (Fig. 6) We further extended our PageRank workflow to be resilient against a type of black-hat search engine optimization known as link farms. Links farms are a collection of websites that contain a lot of links to other websites within the cluster, even if a subset of these webpages are not relevant to one another. In graph theory terms, these farms are cliques or dense cycles that increase each other's out and inlinks, therefore artificially increasing their PageRank.

Previous methods include algorithms to find maximal cliques, strongly connected components, PageRank score distributions of in-neighbors, etc. However, we decided to use a method proposed by (Saxena and Rohit, 2014)[1] that uses GapRank - the inverse of PageRank that calculates the rank contributions from inlinks instead of outlinks. In brief, the GapRank score of page x is given by:

$$GR(x) = \lambda \cdot \sum_{x \to y} \frac{GR(y)}{|inNeighbors(y)|}$$

The intuition behind the algorithm is that empirical studies show that cliques will "trap" PageRank such that $\forall x_i, x_j \in$ link farm, $PR(x_i) = PR(x_j)$ in convergence. However, nodes not part of the link farm may just incidentally have the same PageRank. To prevent such false positives, GapRank examines the link structure of the "inverse" graph such that if these potential nodes also have the same GapRank, then they belong to the same clique. By removing outlinks from these nodes and converting them to individual sinks, we can then rerun the global PageRank algorithm and spread the rank from these link farms to the rest of the graph.

We chose to implement the GapRank method for link farm detection as its implementation almost mirrors that of PageRank, which allowed us to extend our code cleanly to implement this extra credit as a separate module, with a boolean switch during actual runs. GapRank also nicely complements our dangling nodes detection module, so we can convert an entire "clique" into a dangling node for the next iteration. Secondly, while other algorithms seemed to constrain link farm detection to specific subsets like cliques and SCCs, GapRank appeared more robust because it detects graphs components that have the same number of outgoing and incoming links, and so applies to many types of cycles and farms.

2.4 Search Engine (Backend and Frontend)

The search engine consists of two components: a user interface and a SearchManager backend.

Users enter queries into the search engine's user interface. The interface passes query terms to the SearchManager backend, which separates and processes the query into a list of distinct terms. The SearchManager queries the DynamoDB database to retrieve, for each term, the document frequency and top 3000 documents for the term, by term frequency, and then retrieves the PageRank values of each document, then ranks each all retrieved documents by the ranking function. The top results are returned to and displayed on the user interface. We chose 3000 to ensure adequat response time (otherwise the hit list would be much too long) but also quality of results.

In addition, the search engine also includes the additional features:

Wikipedia integration for all search terms, a sidebar appears displaying search results of the query terms on Wikipedia will appear.

EBay integration If the search query involves any of the terms "buy", "shop" or "ebay", a window appears displaying the top EBay search results for the other terms in the query. Furthermore if "buy", "shop", "ebay" is typed without any other term, the search engine assumes the user wishes to search for these terms themselves and retrives the results

[1] Saxena, A., & Nigam, R. Page Rank Link Farm Detection. Retrieved from http://www.ijeijournal.com/papers/Vol.4-Iss.1/I04015559.pdf

normally. Please see Appendix B for a demonstration of the search ui.

3 Ranking

The ranking function we ended up using is as follows:

$$Score(d) = PR(d) \cdot \sum_{w \in Query} tf(w,d) \cdot log_{10} \frac{N}{df(w)}$$

Where d is the document being ranked, w is a term that appears in the query and also the document, $PR(d)$ is the PageRank of document d, $tf(w,d)$ is the term freqency of w in d, N is the total number of documents in the corpus, $df(w)$ is the number of documents containing w.

We based our ranking function on this tried and true model for simplistic elegance, with a few twists to improve the search results. First the term frequency takes into account whether a word appears in the header or body of the document, weighting appearances ⟨ h1 ⟩ tags more strongly than appearances in the body. Furthermore, texts in anchor text are associated with the destination document and treated specially. Finally, the search engine stores the list of positions of a term in a document, which may be used to optimize search results.

4 Evaluation and Experimental Analysis

4.1 Crawler Evaluation

We tested the crawler by changing the number of worker nodes while holding the number of threads/worker node constant at n = 500. Our empirical results indicate an exponential relationship between number of worker nodes and crawler speed.

We also tested the crawler by changing the number of crawler threads running on each working node, while keeping the number of worker nodes constant at m = 5. Our empirical results indicate an initially logarithmic relationship between number of threads and crawler speed, reaching a peak script at around 1000 threads, before decreasing slightly. We hypothesize that the initial logarithmic relationship is due to the bottleneck of writing documents to file (since only one thread can write at a time). The decrease in speed may be due to thrashing or excessive thread sleeping due to exceeding the kernel's open file limit. This was characteristic of the "thrashing" behavior of having too many threads.

5 Indexer Evaluation

The indexer was tested on clusters with different number of m4.xlarge EC2 instances. The running times for each MapReduce as a function of the number of nodes in the cluster is given in Fig. 10. As we see performance gains are quite linear.

As a stark comparison, Fig. 11 details the performance of the hive script, demonstrating a clear bottleneck.

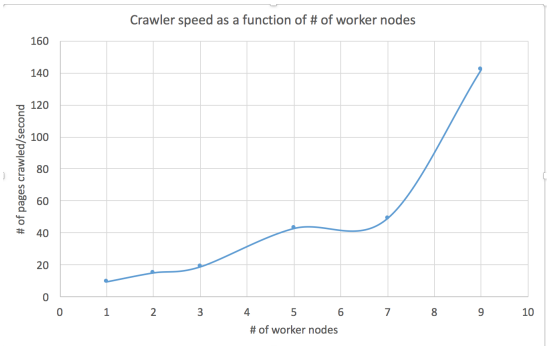


Fig. 8. Crawler Speed as a function of number of nodes

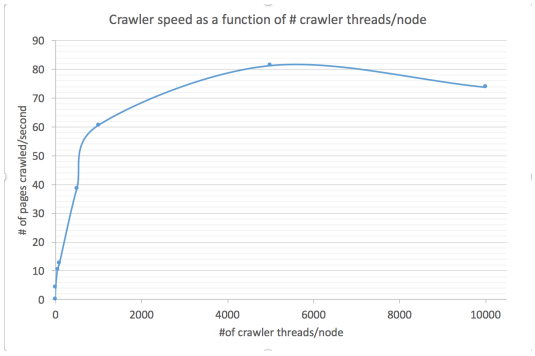


Fig. 9. Crawler Speed as a function of number of threads per node

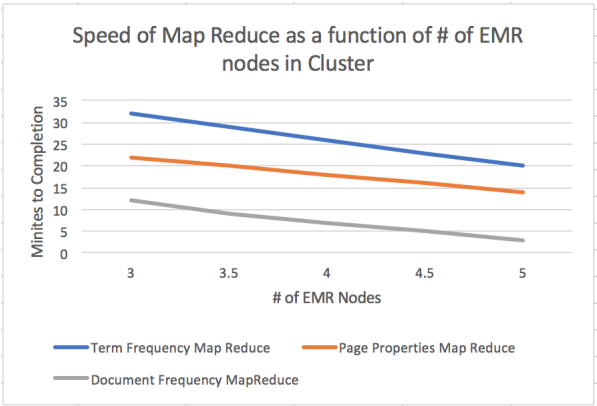


Fig. 10. Map Reduce Performance

6 Page Rank Evaluation

Vanilla PageRank: As expected, achieving more accuracy requires more page rank iterations (Fig. 12). We wanted to find a middle ground between accurate PageRank scores (presumably the more iterations the more accurate) and time to complete. We stopped iterations at 30 in order to save time, that was within tolerance of 0.001. All of these tests were performed using m4.xlarge instances.

We further saw that the time per iteration page ran m4.xlarge decreases sharply when moving from from 1 to 2 nodes, but not as sharply from 2 to 5, potentially due to network overhead.

TF MapReduce output S3 → DynamoDB Hive script.

# Documents Processed	Cluster Size	
	5 nodes (m4.xlarge)	20 nodes (m4.large)
7,000	10 hours	N/A
400,000	7 hours *	6 hours **

* Failed with out of memory error.
** Failed as a result of increasing write capacity on DynamoDB table.

Fig. 11. Performance of hive script to upload documents to DynamoDb

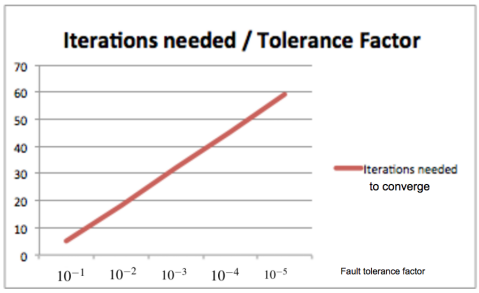


Fig. 12. Number of iterations required for convergence as a function of tolerance required

Vanilla PageRank with LinkFarm Detection: : Running on 1 and 2 nodes resulted in failure (out of memory) before convergence, but the 2 node cluster still managed to reach 14 iterations, achieving acceptable tolerance. Increasing nodes from 2 to 5 drastically improved memory and allowed more iterations to take place.

7 Lessons Learned

General: We learned foremost that dealing with data on such as large scale as that of the internet with distributed, multithreaded architecture is vastly different from processing a small batch of data that fits in memory with a single threaded program. **Crawler:** The primary theme is that the web is a dark, scary and highly irregular place, with a lot of malformed pages. One of the challenges we faced was adapting our crawler to deal with servers that do not comply closely with HTTP/1.1 protocol. For example, the initial design of our crawler expected webpages to specify character encodings, but it soon became apparent that the majority of webpages on the internet do not follow this protocol. Another example of an adaptation that we made to our crawler was handling pages with numerous redirects. After running continuously for 7 hours, the worker nodes will begin to throw occasional "Too many files open" errors, which slowed the crawler down. We were able to mitigate this issue somewhat by putting threads that throw the error to sleep for several seconds. However, if we had more time, it would be nice to completely debug this issue. It would also be nice to implement support for crawling non-html webpages. **Indexer:** We learned that filtering out malformed data received from the web is not trivial. Secondly, splitting text on hashtags, in addition to common punctuation, would have been a good idea. We received a large set of terms starting with a hashtag, which cannot be searched without prepending "#" to query terms. Parsing html contents, and computing tf-idf is fast using AWS EMR, whereas storing this information on DynamoDB was a bottleneck. Hive programs are extremely slow for large amounts of data, and throw unexpected errors. Received a disproportionate amount of useless terms, such as those dominated by numbers and alphanumeric characters, compared to normal frequently queried words. A solution would be to be more selective in the html tags we parsed, instead of parsing every html element that contains text.

8 Conclusion and Reflection

Reflecting on the whirlwind of events surrounding this project, the term "bittersweet" comes starkly to mind. Many all nighters went into the project at the expense of our health, but at the benefit of developing deep friendships that are hard to come by otherwise.

Architecturally, that the hive script to transfer indexed results from S3 to DynamoDb was our bottleneck came as a huge surprise to the entire team. We had expected the bottleneck to be the computationally expensive Indexing and PageRank operations. Since our hive script was buggy in transferring all the data from S3 to DynamoDb and search engine relied on DynamoDb for each query, we were a little disappointed that the actual results served by the search engine did not accurately reflect the hard work we put into the project and the robustness of the rest of the architecture. If we were to do the project again, we will elect to store the index in a disk based database such as MongoDb or BerkeleyDB running locally on the same EC2 instance as our webapp. Not only will this eliminate the need to use Hive Scripts, but will also boost the performance of our search engine by relying on local disk seeks instead of requests over the network.

Furthermore we may elect to use HDFS more effectively instead of leveraging S3 for intermediate storage during the PageRank phase, and use a distributed messaging system like Apache Kafka to make the URL frontier more scalable. Finally, we discovered later in the project that Apache Spark nicely replaces both Apache Hadoop as a computation engine and also as a distributed architecture for the crawler.

That all being said, we firmly believe that this project was a big success. It was certainly a fun and edifying experience and a great bonding experience for the entire team. And since we worked in a closely integrated matter, each team member got exposure to the different technologies used in the project, fulfilling our personal learning goals. Despite the gross amount of coffee consumed in the week leading up to the demo, we are deeply proud of what we have accomplished.

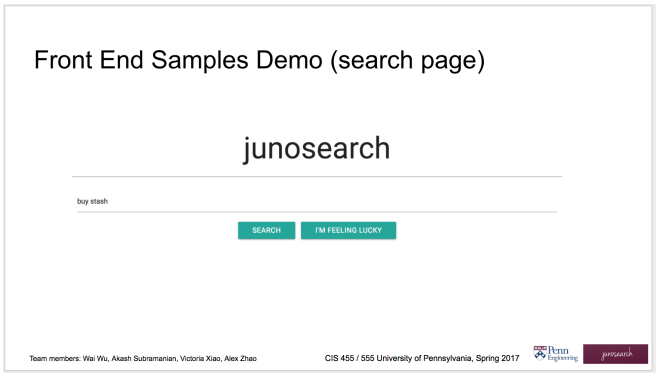


Fig. 13. Search User Interface

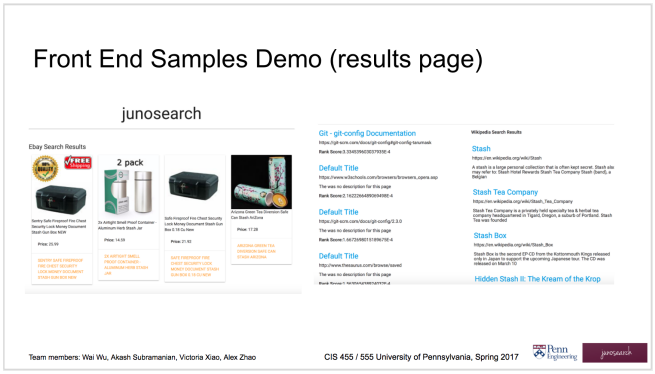


Fig. 14. Search User Interface, shopping results and wikipedia integration

Acknowledgements

We thank Amazon for generously supporting this project with aws student credits. We thank Harshal Turner Lehri for being a fantastic and always patient advisor. Finally, we thank Dr. Andreas Haberlen for an excellent semester, for being a tireless mentor and champion for all of his students.

Appendix A: Architecture Diagrams

Each of the diagrams included in the body of this paper is also attached as a full sized architecture diagram to support printing and more comfortable reading. Please reference the attached document “Junosearch Architecture Diagrams”.

Appendix B: Search UI exhibit

Please see Fig. 13 and Fig. 14.

Appendix C: Running instructions

Please review the README in the project root for detailed running instructions.

junosearch Architecture Diagrams

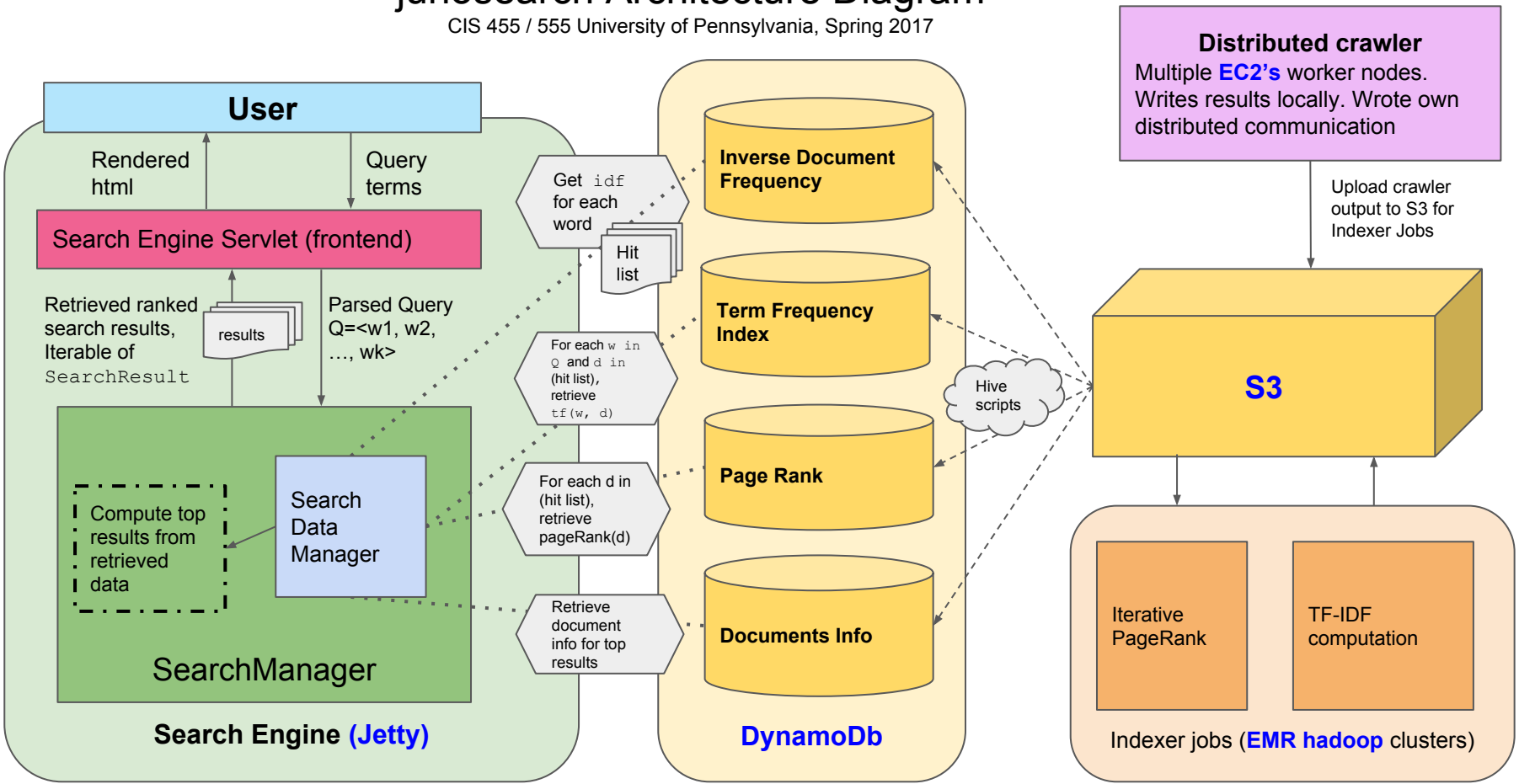
University of Pennsylvania, CIS, School of Engineering and Applied Sciences
CIS 455 / 555 Final Project

Wu Wai, Akash Subramanian, Victoria Xiao, Alex Zhao

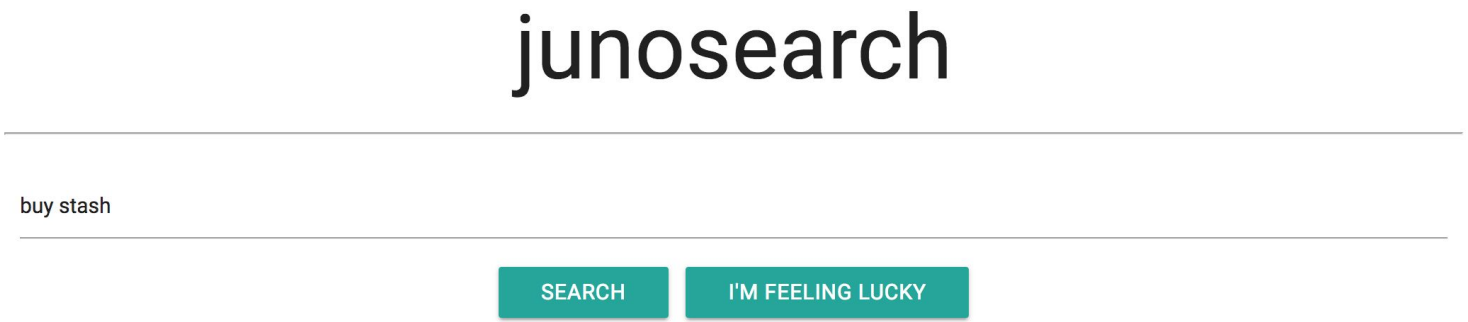
I. Overall Architecture

junosearch Architecture Diagram

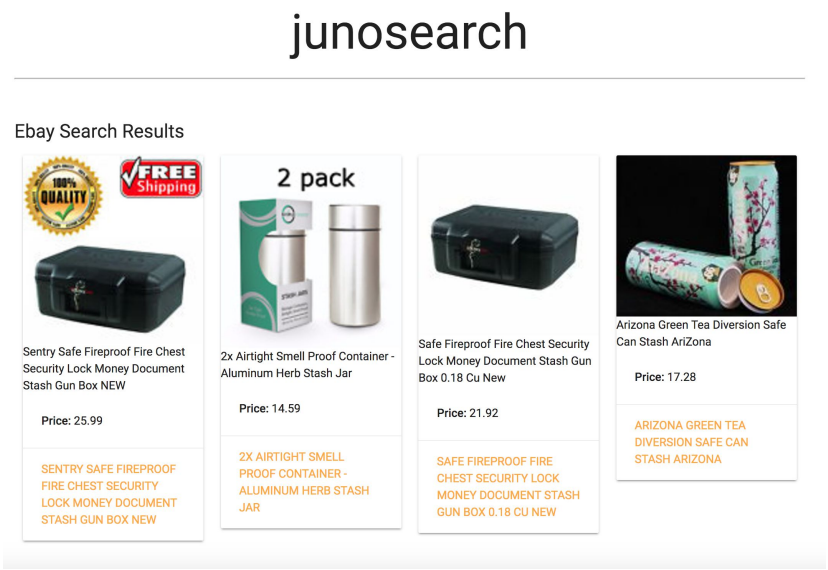
CIS 455 / 555 University of Pennsylvania, Spring 2017

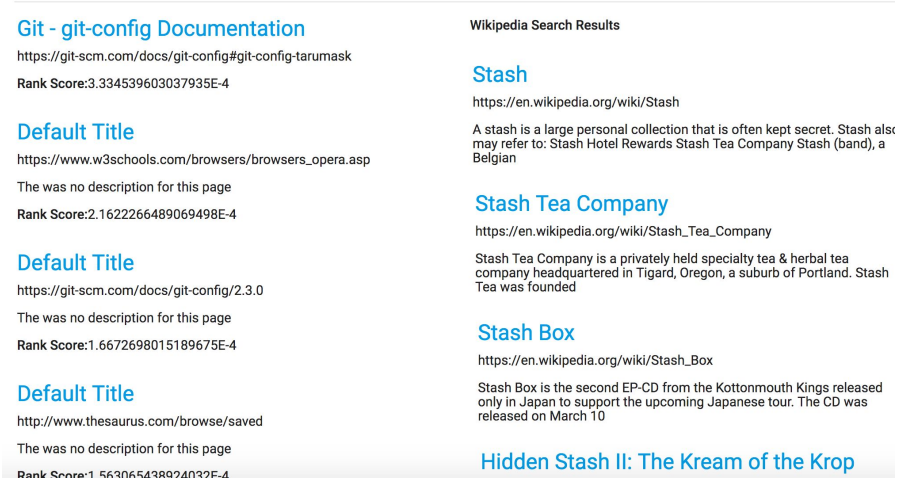


Front End Samples Demo (search page)



Front End Samples Demo (results page)





Database (DynamoDb) scheme

Documents_Info `(stores meta info about documents)`

```
Document {
        "url": "String"; // primary partition key
        "description": "String"; // document description
        "Title": "String"; // title of the document
}
```

Inverse_Document_Frequency `(idf table)`

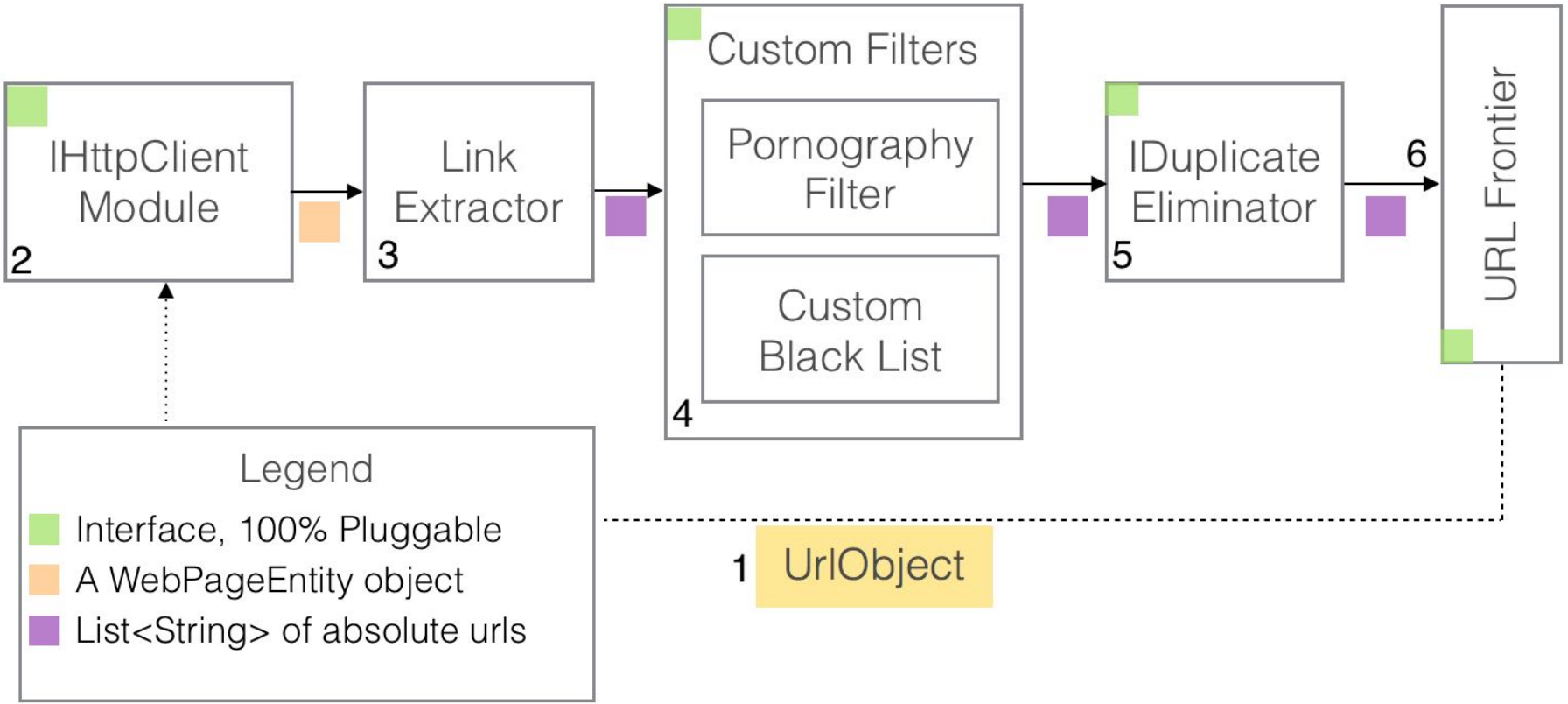
```
Document_Freqency {
        "term": "String"; // primary sort key
        "idf": "Number"; // primary partition key
}
```

Term_Frequency_Index `(stores tf (w, d) for each w,d pair)`

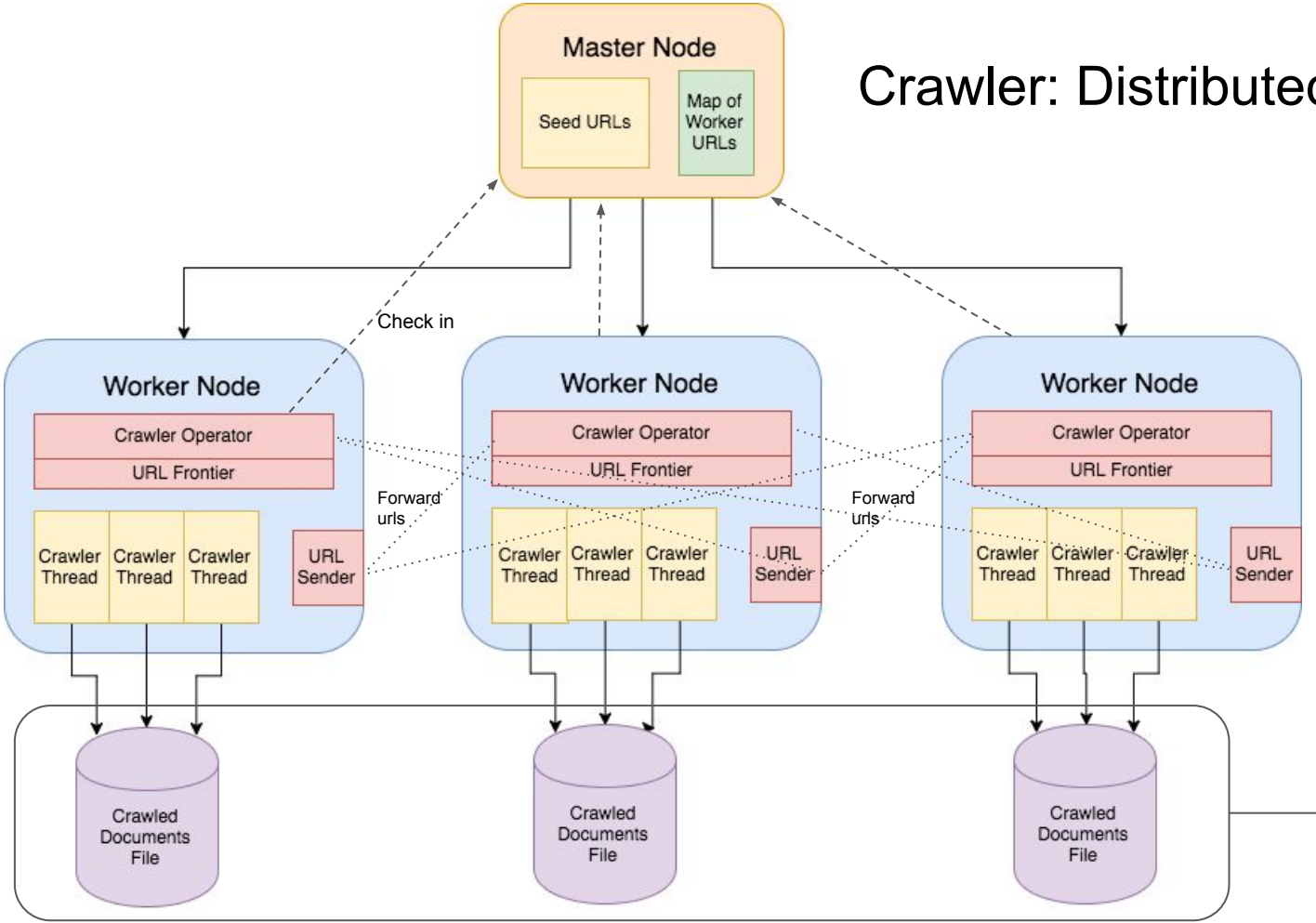
```
Term {
        "term": "String"; // primary partition key
        "tf": "Number"; // primary sort key

        // url of the document for which this tf applies
        "url": "String";

        // a list of where the term appears in the document
specified by the url. Originally for ranking purposes
        "positions": "List<Number>";

}
```

II. Crawler Diagrams

Crawler: Single Threaded, Single Node Design

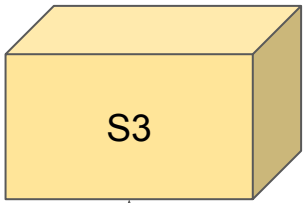


Crawler: Distributed Architecture



Notes: only 3 nodes shown out of 9 for simplicity. Number of threads per node anywhere from 10 to 500 during testing

Per thread operations similar to single threaded architecture



III. Indexer

MapReduce Job I - Computing Term Frequency

Mapper:

Key
line number

Value
<url>\t<content>\t<outlink1,2,...>\t<metadata>

Emit
(<word>\t<url>,
<term_frequency>-<position1,position2,...>)

Reducer:

Key
<term>\t<url>

Values [<term>-<position1,position2,...>]

Emit
(<term>\t<url>,
<term_frequency>\t<position1,position2,...>)

MapReduce Job II - Computing Document Frequency

Mapper:

Key
line number

Value
<term>\t<url>\t<tf>\t<position1,position2,...>

Emit
(<term>, “1”)

Reducer:

Key
<term>

Values
[“1”,..., “1”]

Emit
(<term>, <values_length>)

MapReduce Job III - Document Title and Description

Mapper:

Key
line number

Value
<url>\t<content>\t<outlink1,2,...>\t<metadata>

Emit
(<url>, <title>\t<description>)

Reducer:

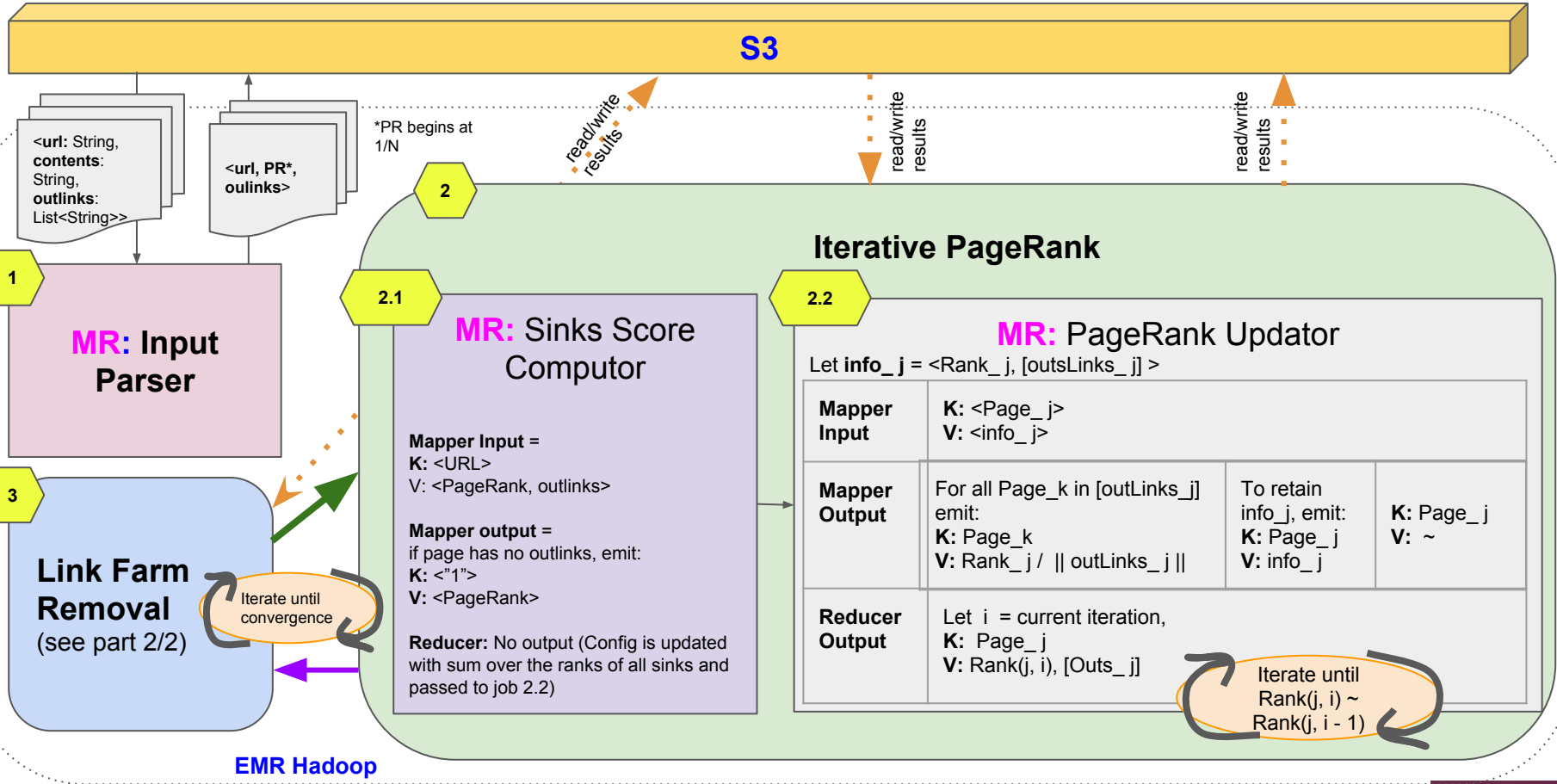
Key
<url>

Value <title>\t<description>

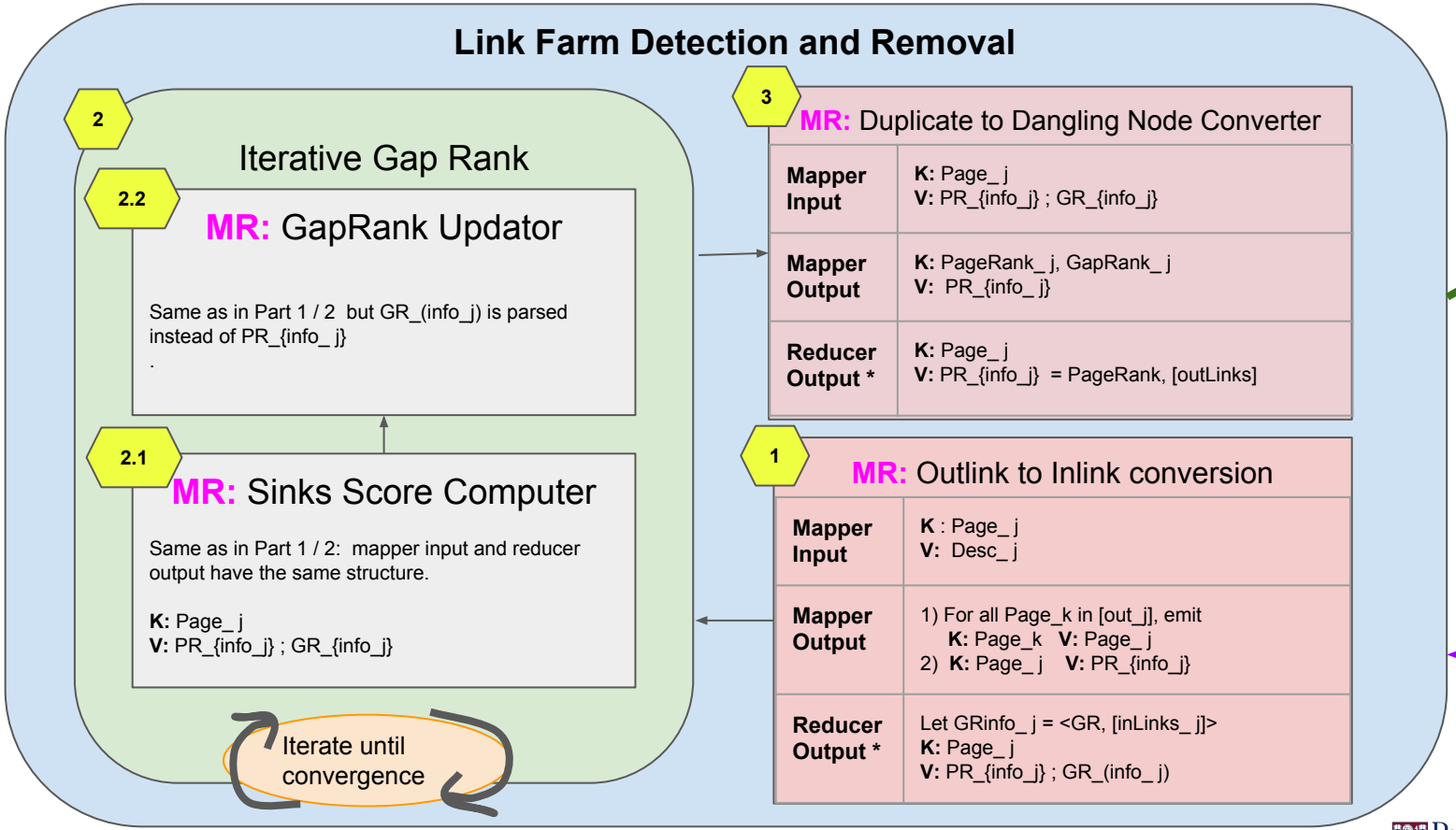
Emit
(<url>, <title>\t<description>)

IV. Page Rank

PageRank Architecture Part 1/2



PageRank Architecture Part 2/2



*Note this is the same output format as input to PageRank phase)

Let **info_ j** = <Rank_ j, [inLinks_ j] > when used as a subscript for GR.

V. Search Engine

Search Engine Architecture