NPMdex: A Distributed Engine for NPM Packages

Alyssa Cong, Calvin Eng, Spandan Goel, and Alexander Zheng
Brown University
https://github.com/Calvineng72/m6

Abstract

This paper presents the design and implementation of a distributed search engine tailored specifically for npm (Node Package Manager) packages. Our system comprises three key components: the crawler, indexer, and query subsystems, each running on custom implementations of serialization, node-to-node communication, distributed key-value storage, and a MapReduce infrastructure. The crawler subsystem efficiently traverses the npm package ecosystem (npmjs.com), gathering package information including URLs and additional metadata. The indexer subsystem organizes and indexes this data to prepare the engine for fast and accurate retrieval during queries.

Leveraging distributed communication protocols, our system coordinates data exchange and task delegation across nodes. Furthermore, our custom MapReduce infrastructure enables parallel processing of indexing and querying tasks, optimizing performance across distributed environments. Through experimentation and benchmarking, we demonstrate the effectiveness and efficiency of our distributed search engine in facilitating comprehensive npm package discovery and retrieval.

1 Introduction

Efficient package discovery and retrieval can be a useful tool for developers seeking to harness the power of external dependencies. Existing search solutions often fall short in effectively navigating the expansive npm (Node Package Manager) ecosystem, characterized by its sheer volume and dynamic nature.

The paper is structured as follows. It starts by introducing the necessary background on/example of using NPMdex (§2). Sections 3–6 highlight key contributions:

- §3 outlines the iterative MapReduce that created the main functionality for the system.
- §4 outlines the crawler subsystem and specifically the map and reduce functions used to implement it.
- §7.3 outlines the indexer subsystem and specifically the map and reduce functions used to implement it.
- §6 outlines the query subsystem and the user interface.

After NPMdex's evaluation (§7) and comparison with related work (§9), the paper concludes (§8).

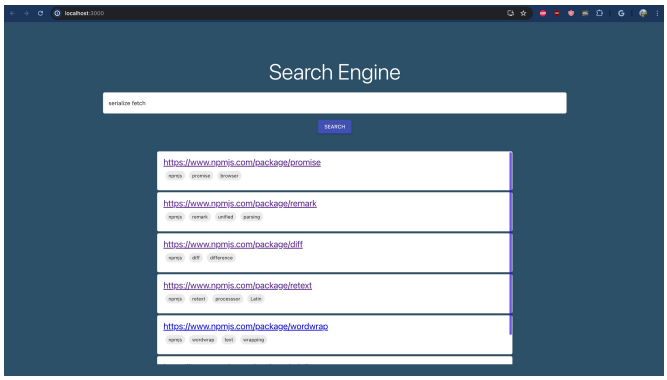

Figure 1. NPMdex Example of search for "serialization fetch."

2 Background, Example, & Overview

2.1 Example Usage

The following describes an example usage of the search engine, as depicted in figure 1. Before the user makes the query, the search engine will or will have already crawled the npm package web link graph and indexed the extracted text data. The user will enter key terms important to their query; for instance, a user searching for a serialization package might input "serialization fetch." In the background, the search engine will query the indexed data to create a list of five URLs that best represent the search criteria. In this example, the top results include the packages `promise`, `remark`, `diff`, `retext`, and `wordwrap`, all which are packages that relate to analyzing texts or to fetching or both.

2.2 Distribution Environment

Our search engine is built upon a custom distribution service that provides functionality such as distributed key-value storage, cross-node communication, sharding, and MapReduce infrastructure.

2.2.1 Distributed Key-Value Storage

The distributed key-value storage, depicted in figure 2, supports distributed in-memory(`mem`) and persistent storage(`store`) on distributed machines. Both services contain the following methods: `get`, `put`, `del`, `append`, and `multiAppend`. The `get` method takes in a key and returns an objected associated with that key. If a null key is provided to the get method, all valid keys are returned to the caller. The `put` method takes in a key and an object. The method stores the object under the associated key either in-memory or persistent. The `del`

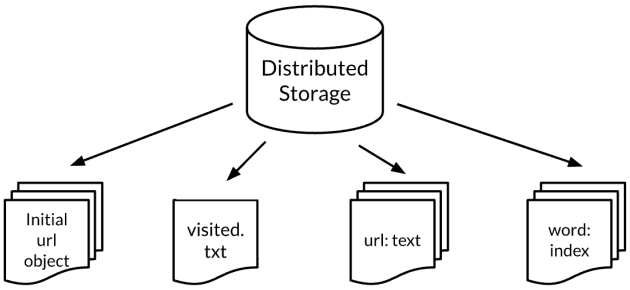

Figure 2. NPMdex Distributed storage overview.

method takes in a key and deletes the key and object associated with the key. The append method takes in an object and key, and appends the object to the key's existing object in memory. The `multiAppend` method takes in a list of key object pairs({key: value}, {key: value}, ...}) and appends each of the input objects to their corresponding key's existing memory object.

2.2.2 Communication

Our distributed search engine utilizes HTTP protocol as the foundation for communication between nodes. Each node in the system sets up a listener to listen for incoming communication requests. This listener enables nodes to send and receive messages, facilitating interactions within the distribution environment. To enable remote procedure calls (RPCs) and allow nodes to invoke functions on other nodes, our system implements a RPC mechanism. This mechanism abstracts away the complexities of network communication, allowing nodes to communicate and collaborate effectively.

2.2.3 Map Reduce Infrastructure

Our system includes a MapReduce service called `mr` designed to facilitate parallel processing of data across distributed nodes. The `mr` service exposes a method called `mr.exec` which serves as the primary interface for executing MapReduce tasks. The `mr.exec` method takes in three parameters: a mapper function, a reducer function, and a callback function. The mapper function is responsible for processing input data and emitting intermediate key-value pairs, while the reducer function aggregates and processes these intermediate results to produce the final output. Upon completion of the MapReduce task, the callback function is invoked to handle the resulting output. Optionally, the `mr.exec` method can take in another parameter called `rounds`, which is used to constrain the number of iterations the iterative MapReduce will run. More about iterative MapReduce can be fuond in §3.

2.2.4 Sharding: Consistent Hashing

Our sharding technique employs consistent hashing to distribute data across multiple nodes in a scalable and fault-tolerant manner. Consistent hashing ensures that the distribution of data remains stable even as nodes are added or removed from the system, minimizing the need for data rebalancing and reducing the impact of node failures. By mapping data keys to a continuous hash ring and assigning each node responsibility for a range of hash values, consistent hashing enables efficient data lookup and retrieval. This approach not only improves system performance but also enhances fault tolerance and load balancing, making it well-suited for distributed environments where dynamic scaling and resilience are paramount.

2.3 Design Overview

Our system comprises three core subsystems: the crawler, indexer, and query subsystems, built on a custom distribution environment for storage, MapReduce, and communication. The crawler gathers npm URLs and package information by traversing the npm web link graph. The indexer organizes and indexes this data using a term frequency approach. Finally, the query subsystem delivers quick search results to developers by leveraging existing indexed data.

3 Iterative MapReduce

In our distributed search engine, we utilize an iterative MapReduce model to traverse the web link graph and discover new npm packages for indexing. After each iteration of the MapReduce process, the results of the reduce phase are fed back into the next iteration of the mapper. This iterative approach allows us to continuously explore the link graph, uncovering new connections and potential npm packages that may not have been discovered from a single sweep. By iteratively refining our index based on the evolving link graph structure, we ensure comprehensive coverage of npm packages, facilitating more accurate and up-to-date search results for our engine.

In implementing iterative MapReduce, we found it important to maintain state information regarding the keys needed for subsequent MapReduce iterations. These keys represent new URLs discovered in the web link graph, which is needed in the discovery of new npm packages. As the system traverses the link graph during each iterative MapReduce round, the system also keeps track of explored links using the distributed storage system, as a given URL is always sent to the same node. Storing this information prevented redundant visits to web pages that had already been crawled.

Additionally, in order to make use of consistent hashing within our nodes and ensure that all objects with the same key would be on one node before the next map phase, before the call to the MapReduce again, a `store.put` call is issued with all of the new keys to ensure that they are sent to the

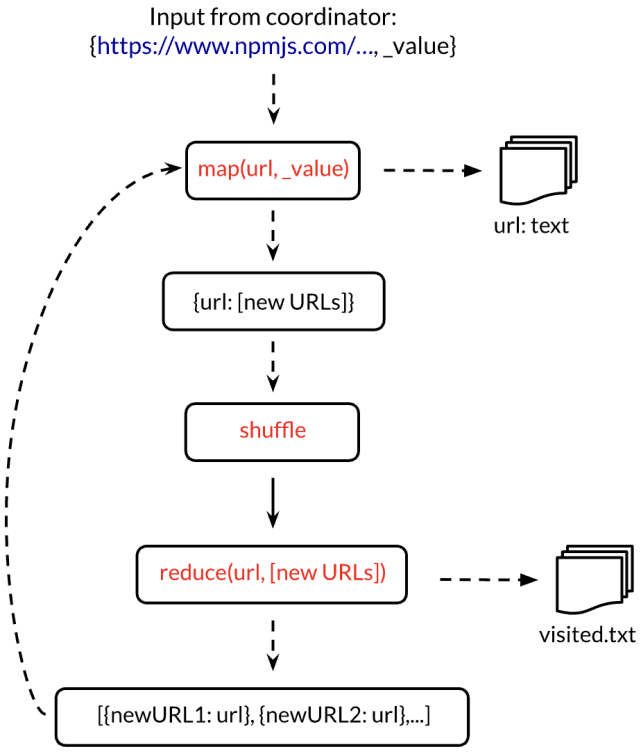

Figure 3. NPMdex Crawler subsystem overview. The coordinator node calls map on each pair of URL to an irrelevant value; the text is scraped from this URL and stored in a file; the new URLs are extracted from this page; this is sent back to shuffle and then to the proper node, where it is then reduced into an object that has key of the URL and an array of the newly extracted URLs; when sent back to the coordinator, the array is expanded to create an array of objects that are then stored, and the new keys are sent back to map to begin the MapReduce process again.

correct nodes. This ensures that during each map call, each key is on the correct node, meaning that the `url: text` object will be stored in the correct node as well. This mitigates the issue of needing to coordinate sending objects again throughout shuffle and reduce.

4 Crawling Subsystem

The crawler subsystem, as shown in figure 2, serves as the discovery mechanism for traversing the npm web link graph and extracting information on npm packages. The system is composed of two parts: a map and a reduce function. For a broad overview, the crawler subsystem is responsible for checking whether or not a given URL has been visited, fetching URLs, extracting text from the HTML of web pages, searching HTML for links, and storing information on the web link graph. The input to the crawler system is composed of a set of URLs to explore, while the output is the set of URLs discovered from the input pages. As such, the subsystem is run with iterative MapReduce to iteratively traverse the web link graph, so only one call is necessary for the subsystem to run.

Although a key and value are input to the map function, only the key is used, as the key is the URL to crawl, while the value is the source URL for the provided key. Using the URL to crawl as the key is necessary for our MapReduce infrastructure to guarantee that the same URL is always sent to the same node. As a result, nodes only need a local set of visited URLs so that a given URL is not visited multiple times. Within the map function, there are first two checks: a check to ensure the URL to crawl is within the npmjs.com domain and a check to ensure the URL has not already been visited. These are followed by the actual fetching of the URL, which uses the package sync-fetch. With the HTML of the page, the text is extracted and stored within the distributed system under a filename that ends with "text." In addition, the HTML is parsed for links to crawl in the next round of MapReduce. Therefore, the map function returns a dictionary, where the key is the URL that was crawled, and the value is an array of the URLs that were discovered on the page.

With the array of new URLs, the reduce function first stores the links for future references to the web link graph in the distributed system. Similar to before, the object stored is a dictionary, where the key is the source URL, and the values are an array of sink URLs. This information is stored for future work to update the indexing subsystem to use PageRank. Afterward, the links are filtered for duplicates before returning an array. This array is composed of dictionaries, where each dictionary has a sink URL as the key, and the source URL as the value. The output must be an array of separate dictionaries since the crawler subsystem utilizes the iterative MapReduce infrastructure.

5 Indexing Subsystem

The indexing subsystem is responsible for converting the text extracted from web pages into a form more suitable for queries in order to facilitate efficient searches for users. As with the crawling subsystem, the indexing subsystem is comprised of both a map and reduce function. It uses the files from the distributed key-value store that contain text data to create new files that contain information for each unique word in the text data. A term frequency-based indexing approach is used to capture the relevance of npm packages within the web link graph. Once the indexing is complete, a file should exist for each unique word in the text data. These files can be retrieved using the words in any search to determine the most relevant URL.

The input to the map function of the indexing subsystem is a URL and the text that was extracted from the HTML of that web page. In order to create metrics of the extracted text data, the text is first cleaned by removing words that contain numbers or special characters, followed by the deletion of any excess spaces in the text. The text is then tokenized by

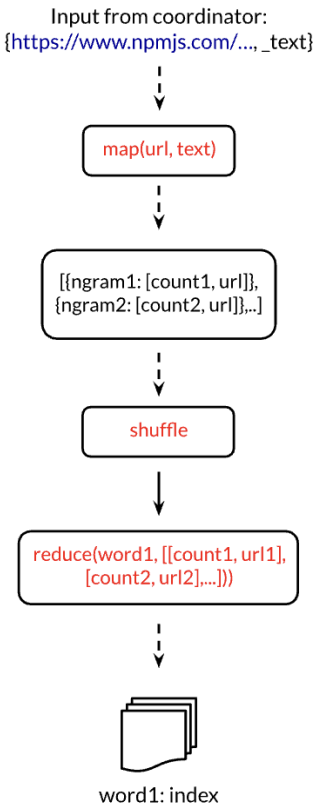

Figure 4. NPMDex Indexer subsystem overview. The coordinator node passes in

splitting along spaces, and all stop-words are removed from the array of words. Once this is complete, words are stemmed using the Porter Stemmer algorithm from the natural package. The final step of the map function is converting the words into dictionaries that contain the count of the word, along with the URL the text originated from. As a result, the output of the map function is an array of dictionaries, where the key of each dictionary is a word, and the value is a dictionary that contains the count of the word and the URL the text originated from. While the focus of the system is on the counts of individual words, the code also includes bi-grams and tri-grams.

The reduce function of the indexing subsystem is much less complex, in that it solely converts the output from the map function into a form that makes retrieval during querying more efficient. The input to the reduce function is a key, which is a word, and values, which is an array of dictionaries that contain the respective counts of that word in each URL. This is converted into a dictionary, where the key is the word, and the value is a dictionary that maps URLs to their count of the word. This output structure makes aggregating the counts among URLs simpler in the query subsystem.

6 Query Subsystem

The query subsystem extracts information from the constructed index to return a response to the user based on their provided search parameters. The URLs that best match the query are determined by the counts of the words in the query in the text of the URLs. As is performed in the indexing subsystem, the search is cleaned and stemmed to match the format of the data created by the indexer. The query subsystem then iterates through each word in the query, retrieving the data for that given word from the distributed key-value store. While iterating through each word, the data from each word is aggregated, i.e., the counts are combined for each URL. Once the data has been fully retrieved and combined, the subsystem selects the five URLs that have the highest total counts, and these URLs are returned to the user. If no URLs contain any of the provided words in the user's query, then no URLs are returned, as there are no relevant URLs.

The interface of the query system is a simple web search bar that allows the user to input search terms. The interface provides a button to submit the search term and returns a visual list of the most relevant search terms. The interface is implemented using ReactJS, NodeJS, and ExpressJS.

7 Evaluation

Our system's performance will be evaluated using two key metrics: exploration time and query response time. Exploration time will measure the duration to explore a set number of npm links, assessing the efficiency of the crawler and indexer. Query response time will gauge the system's responsiveness by measuring the time taken for a user to receive search results. Through analysis of these metrics across varying conditions, we aim to assess the system's performance, scalability, and fault-tolerance objectively.

This version maintains the essential information while reducing unnecessary adjectives for a more concise and objective evaluation approach.

7.1 Correctness

While our query subsystem returns sensible results for many search queries, we have observed instances where some of the results are not closely related to the query. We attribute this inconsistency to the simplicity of our current indexing strategy, which relies solely on term frequency. This strategy may not capture the nuanced relationships between npm packages and their relevance to specific queries.

For example, in the example shown in figure 1, searching for "serialization fetch" returns results such as `promise`, `remark`, `diff`, `retext`, and `wordwrap`, which are reasonable search results, but perhaps not as promising as results like `node-fetch` would be.

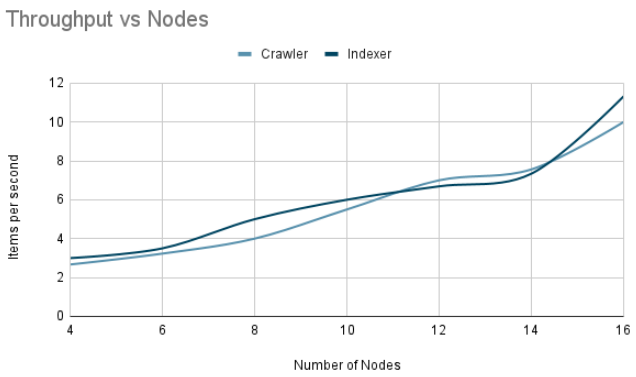

Figure 5. NPMDex Crawler and indexer throughput vs. number of nodes.

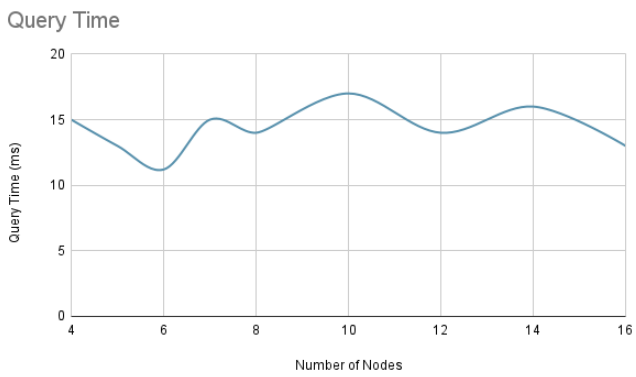

Figure 6. NPMDex Query time with respect to number of nodes.

7.2 Performance: Crawler and Indexer

To assess the efficiency of the crawler and indexer, we continuously monitored the rate at which they processed items. This was achieved through periodic logging of the crawling and indexing processes, allowing us to calculate a metric representing the number of items processed per second. Our experimentation aimed to gauge the scalability of both subsystems with an increasing number of nodes. As illustrated in Figure 5, we observed an increase in throughput by the crawler and indexer as the number of nodes increased within the range of 4 to 8 nodes.

Our final system was able to crawl and index around 3500 URLs, with the items per second being crawled and indexed increasing steadily with the number of nodes, observing that if averaged across the nodes, our system processed a little less than one item per second. After around 3500 URLs, our system ran into HTTP request errors with fetching, as detailed in the discussions section.

7.3 Query Latency

To assess query latency, as depicted in 6, benchmarks were created to test the amount of time it took between receiving the query and the results being output. This was around 15 milliseconds, regardless of the number of nodes, as this was not a MapReduce call and required no shuffling, reducing, and other lengthy calls, so it did not scale with the number of nodes.

7.4 Component Runtime

To assess component runtimes, logs were created to mark the beginning and ends of each process (essentially each call to the MapReduce service from crawler and indexer). With four nodes, it was found that for the 3500 URLs, crawling them took around 2220 seconds, while indexing them took 1115 seconds. This included time gaps between fetches for URLs that had to be implemented in order to fetch the thousands of URLs. This information is displayed in table 1.

Table 1. Component Runtime. The table below summarizes the components comprising our search engine NPMDex, and their runtimes, when there were 4 nodes and 3500 URLs.

Component	Runtime (seconds)
Crawler	1534
Indexer	1117

8 Discussion

8.1 Limitations

Our system's scalability was limited in that it could crawl around 1000 URLs but was unable to go further than this; this was the result of HTTP requests and socket hangups that are further discussed in the Challenges section below.

Furthermore, our system's indexer, as previously mentioned, is limited in that it only sums up word frequencies, leading to potentially skewed results, as the resulting URLs that are passed back from the system may not most accurately represent the search terms. To address this limitation and improve the relevance of our query results, in the future we would like to improve our indexer strategy to incorporate additional indexing techniques, such as Inverse Document Frequency (IDF), Term Frequency-Inverse Document Frequency (TF-IDF), or a distributed version of PageRank.

Another limitation of our system is the number of pages it is able to crawl; we were limited to about 3500 URLs, proceeding with more URLs resulted in errors when fetching webpages too closely at a time. As a result, we implemented waiting periods between each fetch call in order to not overload the server, which allowed us to reach 3500 URLs but failed afterwards. As such, the scope of our project is somewhat limited in terms of the number of pages it was able to fully crawl and index, so it does not encompass as many NPM packages as we would like.

Furthermore, although we had originally planned to use Brown University's snapshot of the NPM packages, which were easier to navigate and did not include extraneous pages,

the webpage that hosted this snapshot broke earlier into our project, and as such we had to use the actual NPM JS webpage, which includes many extraneous URLs that are not solely packages.

8.2 Challenges

8.2.1 Batch Processing

We had numerous socket hangup issues and connection reset issues that were found to occur because the number of HTTP messages sent throughout the MapReduce service. Each time any of the nodes uses `store.put`, or any services, the `comm.send` service used to send these messages employs an HTTP message; as we were debugging this issue, we found that our system was sending over 75,000 HTTP messages, which would overload the network and cause nodes to fail.

This meant that there was a 1-to-1 relationship between the number of objects (i.e. the number of extracted URLs) and the number of messages sent, which results in hundreds of thousands of messages on a large scale when we want to crawl 100K pages. We implemented a compact function to reduce the number of HTTP requests being sent before shuffle and reduce, and this ended up reducing the number of messages by a factor of 3.0-3.2. However, we continued to have socket hangup issues, as there were still 25k HTTP messages.

We addressed this issue by changing some of the functionality in our shuffle phase to use batch processing; previously, every individual object that was returned from map, would be shuffled and one message would be sent to the node responsible for handling the reduce for it. In an effort to reduce the number of messages being sent, we implemented a `multiAppend` method and a `batchOperations` in our `store` that would go through all the objects and figure out which objects would be sent to the same node, and then make larger batch messages that would send one message to each node with all the objects that needed to be put on that specific node. After implementing and integrating `multiAppend` and `batchOperation`, the number of messages was constant in terms of the number of nodes; specifically, for n worker nodes, the number of messages sent in the shuffle phase became n^2. This greatly improved performance, as the number of HTTP messages sent decreased from > 75k to a constant number. However, this did increase the runtime of our service as we needed to account for the time it took to batch objects together.

8.2.2 Distributed Storage on Correct Nodes

A design implementation challenge that we faced in the process of implementing this system was figuring out how to combine all of the map functionalities in one function and correctly put items in the distributed storage such that all nodes could access the files. For example, the `visited.txt` file should be in the global storage and all nodes at any given moment should be able to read from it and write to it. Furthermore, we also wanted to ensure that when we stored the new files that would be created after reading the text from the URLs, they would be on the correct node (meaning that when `map` is called, all the keys that are hashed to that node are already on it, and *only* those keys are on it). This meant that before the next call to the iterative map reduce, all of the new objects that were created (i.e. `{newURL: oldURL}`) needed to be put in the distributed storage, as the `store.put` call would appropriately hash and route them before the next iteration began. As such, we implemented this call to occur with the outputs of `reduce`.

8.2.3 Crawler

We faced a blocker with the crawler subsystem where it appeared that the iterative crawling seemed to only work for a set number of URLs per origin URL. For example, when we were testing this out, crawling a small URL such as https://www.example.com would allow for three rounds of MapReduce iterations, crawling a total of 36 URLs before breaking, while for https://www.gutenberg.org, the MapReduce would break halfway through the second round of iterating and would crawl 47 URLs; however, they would always crawl the same number of URLs for each starting URL. After debugging, this was found to be the result of two errors in our code. The first was that we did not originally handle the case when `map` returned null (this would happen if `map` received a URL that had previously been crawled, and would return null in order to not crawl duplicate URLs), and instead this issued and error and halted the process.

Once this was resolved, a secondary issue was that our crawler was returning an error extremely early in the process because nodes would be unable to find certain keys that were passed in. However, because of the nature of a distributed system, where the keys are hashed to be balanced among different nodes, this is natural, as no one node will have all the keys stored. Therefore, returning an error prematurely was incorrect behavior, and this was modified to only print out the error but not return it and quit the process if a node returned a `"Key not found!"` error; in this manner, we were able to still go through the logs and see that nodes were unable to find certain keys if this was an erring behavior, but would otherwise ignore this during expected behavior.

8.3 Link Cycle Detection

One problem we wanted to solve is to detect cycles in the web-link graph. This would prevent the crawler from scraping web pages that have already been visited. This is implemented by having each node in the distributed system keep track of all the URLs that hash to said node. Therefore, when the crawler comes across an already visited URL, the URL will hash to the corresponding node which can detect the web-link cycle using it's local visitation record.

9 Related Work

The MapReduce structure of the search engine largely follows the original framework described by Jeffrey Dean and Sanjay Ghemawat in their seminal paper "MapReduce: Simplified Data Processing on Large Clusters" [2]. As described in section 8.1, the implementation for NPMdex opts for batch processing instead of a combiner function (section 4.3 of the original paper). All other components of the MapReduce infrastructure mirror that of the original paper.

Our gossip or rumor-spreading protocol was largely inspired by The Promise, and Limitations, of Gossip Protocols by Ken Birman [1]. Notable characteristics that our gossip protocol share with the one discussed in the paper include random peer selection, communication reliability is not assumed, and involves periodic, pairwise, and inter-process interactions.

The distributed search engine described in the paper "Design and Implementation of Scalable, Fully Distributed Web Crawler for a Web Search Engine" is quite similar to NPMdex [3]. Both search engines follow a cascade model, whereby crawling, indexing, and querying are three distinct and sequential steps. In addition, both systems use consistent hashing for their respective distributed storage infrastructure. Key differences lie in the fact that NPMdex does not consider the quality of pages, only uses a simple frequency-based index, and does not implement some of the fault tolerance features found in the paper.

10 Conclusion

NPMdex's implementation, as well as all the example code and benchmarks presented in this paper, are all open source and available for download: https://github.com/Calvineng72/m6.

11 References

References

[1] Ken Birman. 2007. The Promise, and Limitations, of Gossip Protocols. *ACM SIGOPS* 41 (2007), 8–13. Issue 5.

[2] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51 (2008), 107–113. Issue 1.

[3] M. Sunil Kumar and P. Neelima. 2011. Design and Implementation of Scalable, Fully Distributed Web Crawler for a Web Search Engine. *International Journal of Computer Applications* (2011).

Acknowledgments

Thank you to Professor Nikos Vasilakis and the wonderful teaching assistant team of the course "CSCI 1380: Distributed Systems" for their support and guidance throughout the semester. Their expertise and commitment have been essential in the completion of this final project.

A Reflections

Given the complexity of the overall distributed search engine, the paper took longer than expected. Though we started quite early, it was difficult to finish before the deadline since we were somewhat uncertain how much we would accomplish in terms of code. On the other hand, we found the poster to be a simpler task, especially as it does not contain as much content. Adding together the contributions from each team member, the paper took approximately 18 hours to complete, which includes the time spent making diagrams. The distributed version of the project ended up with 3.5k lines of code, much greater than the 150 lines of code required for the non-distributed version. Most team members approximated the lines of code to be 5,000, which only slightly overestimates the actual value.