Distributed Web Atlas: A Scalable MapReduce-Driven Digital Library Search Engine Architecture

Nathan Andrews
Brown University
nathan_andrews@brown.edu

Sebastian Criado
Brown University
sebastian_criado@brown.edu

Justin Phillips
Brown University
justin_phillips@brown.edu

Leishu Qiu
Brown University
leishu_qiu@brown.edu

Yuxuan Liao
Brown University
yuxuan_liao@brown.edu

Abstract

While distribution provides several benefits with regard to data segmentation and preservation, fault tolerance, and additional processing due to having additional processors, it also provides several new challenges regarding data and state distribution that must be addressed. This paper presents Distributed Web Atlas, a scalable, content-based, and distributed MapReduce-driven digital library search engine that aims to address some of the issues inherent in distributed computation. It covers the design, implementation, and evaluation of this search engine, its related crawling, indexing, and querying subsystems, and the methods and approaches used to preserve state and communicate across the independent nodes. The Web Atlas subsystem was deployed on a set of Amazon EC2 instances and was tested extensively in order to analyze the processing and memory load allocated to each node.

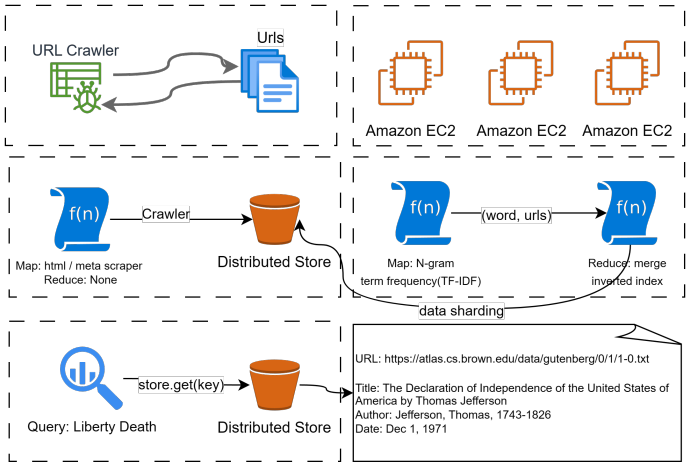

Figure 1. Distributed Search Engine

1 Introduction

Digital eBook libraries like the Gutenberg Project have proliferated across the internet, housing millions of freely accessible books and a plethora of other digital resources. While these resources are invaluable, it is equally critical to have an efficient search engine capable of querying vast datasets by specific terms and returning precise results promptly. This paper presents a distributed and scalable MapReduce-driven digital library search engine architecture, named Distributed Web Atlas (DWA). We implemented crawling, indexing, and querying workflows using the MapReduce[3] framework and processed massive textbook data on multiple AWS EC2 nodes through parallel computing, with the aim of making it efficiently searchable and accessible.

Our search target is the Project Gutenberg (Brown CS mirror). Our system features a robust capability that enables users to search not only by metadata but also by specific words within the text, facilitating deeper, more detailed, and more accurate search results. Here's the URL link to our GitHub repository: https://github.com/nathanjandrews/m6.

Table one shows the component decomposition of our DWA search engine, which consists of 4 key components: crawling, indexing, querying, and a web service, and it summarizes the component, lines of code, and the number of tests for each component. The crawling subsystem includes three sub-crawlers: URL crawler, content scraper, and metadata scraper. Each crawler takes a URL link to an e-book as input and parses the corresponding information of the book. The indexing subsystem takes the result of the crawling system, uses it to create an inverted term to URL indices through the map function, and then ranks the terms by an indexing technology TF-IDF[6]. Then the query subsystem receives a query and responds with a list of pages ordered by TF-IDF scores. Finally, the web service provides an interface for users to interact with our system.

Table 1. Component decomposition. The table below summarizes the components comprising our search engine DWAs.

Component	Lead	LoC	Test
Crawling	Yuxuan	150	9
Indexing	Nathan	100	9
Query	Sebastian	100	9
Web(frontend+backend)	Leishu & Justin	240	1

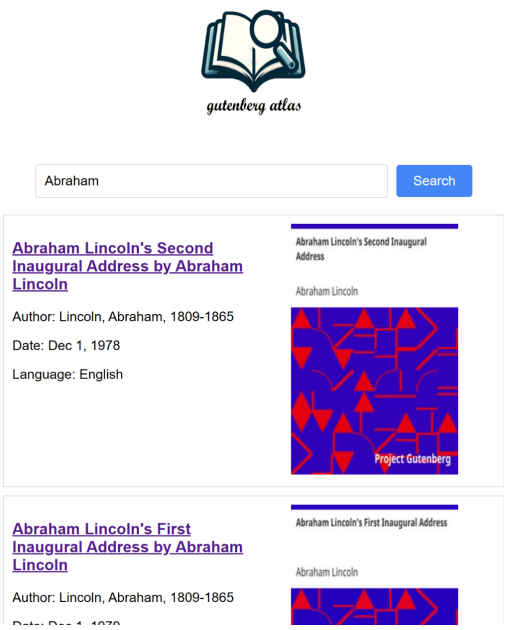

Figure 2. Search by Metadata(Author)

The paper is structured as follows. It starts by introducing an example of a typical use of our search engine DWA (§2). Sections 3–5 highlight key contributions:

- §3 outlines the crawling subsystem and contribution.
- §4 outlines the indexing subsystem and contribution.
- §5 outlines the query subsystem and contribution.

After DWA's evaluation (§7) and comparison with related work (§9), the paper concludes (§8) and (§10).

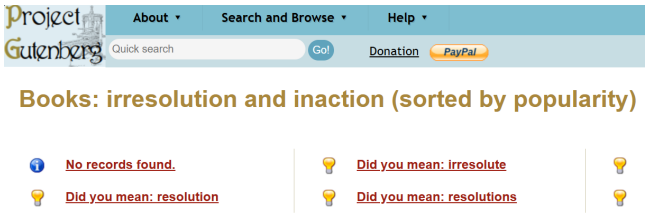

Figure 3. Gutenberg's Search Engine

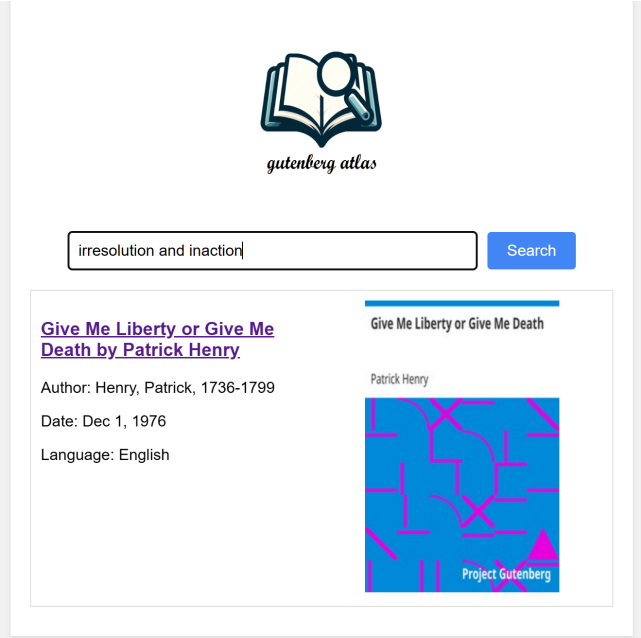

Figure 4. Search by Content

2 Example

This section demonstrates a typical use of the Distributed Web Atlas (DWA). Our user interface is simple, consisting of a logo, an input box, and a search button. End-users can enter general information about a book, such as the title and author. As shown in Figure 2, when a user types in the term 'Abraham', DWA returns a list of books whose metadata includes 'Abraham' and whose content contains the word 'Abraham'. A typical search result displays the book title, author, publication date, language, and a book cover image if available from the Gutenberg Library. Additionally, while the Gutenberg website doesn't support this, DWA allows users to input specific terms from the book content to refine their search. For instance, when a user inputs 'irresolution and inaction'—a phrase from the content, DWA returns a list of books that include this term in their content. A comparison between the search results of Gutenberg and DWA is shown in Figure 3 and Figure 4.

3 Crawling Subsystem

The crawling subsystem consists of three components, a URL crawler, a content scraper, and a metadata scraper.

3.1 URL Crawler

URL crawling involves systematically discovering and visiting web pages. Here are some techniques:

- Breadth-First Crawling: Start from a seed set of URLs and explore pages level by level, only moving onto the next level once the entire first one has been categorized. Useful for discovering a wide range of content.
- Focused Crawling: Prioritize specific topics or domains. Since the goal of our project is to implement a distributed search engine for Project Gutenberg, the URL must end in .txt, and we need to narrow down the domain name to the official Project Gutenberg site or a mirror site.

3.2 Content Scraper

We can efficiently scrape content from large numbers of URLs like 10k or even 100k pages by leveraging MapReduce. To distribute the workload across multiple machines, we employ consistent hashing. This technique ensures that data is evenly distributed among nodes while minimizing data movement during scaling or failures.

Then we implement the map function for MapReduce. The input data is in the form of key/value pairs. In our case, each key represents an identifier, and the value is the URL.

For a large-scale content scraper, the map function extracts relevant information from each URL. It involves fetching web pages, parsing HTML, and extracting specific data points.

The output of the map function consists of intermediate key/value pairs. These pairs represent the processed data, where the key is a unique identifier (e.g. the hash of the URL) and the value contains the extracted content.

3.3 Extra Credit Feature: Metadata Scraper

Our search system aims to index a vast collection of books from the Gutenberg Project. We'd like to implement an extra credit feature: a metadata scraper. This scraper leverages MapReduce to extract valuable metadata from the books. There are some key differences between content scraper with metadata scraper.

- Data Source: Because we use Brown University mirror sites, where books are available as plain text and lack metadata. We need to transform the mirror site URLs to Project Gutenberg websites.
- Extraction: We utilize JSDOM to parse the web pages to retrieve metadata such as author, cover image, published data and so on. For instance, we target relevant elements using queries like `a[itemprop="creator"]` to extract the author of the book.

By combining metadata extraction with our existing content scraping techniques, we enhance the richness of our search system's index. This allows users to discover not only the content but also essential details about the books they seek.

4 Indexing Subsystem

For the indexing component, the map function plays a crucial role. It operates on the raw content obtained from the crawler component. Here's how it works:

- Tokenization: We use the "natural" library for tokenization, which is a natural language facility for NodeJS. The document is tokenized into individual terms (words).
- Stopwords: Common words (such as "the," "and," "is") are removed before the stemming.
- Stemming: terms are reduced to root form and lowercase. (e.g., "Running" becomes "run").
- N-gram: The map function will also generate a sequence of n adjacent terms for word counting and indexing.
- Counting Term Frequencies(TF): The frequency of each term in the document is calculated.
- Output: The map function emits key-value pairs, where the key is the term, and the value includes information like the document ID and the term frequency.

The reduce function aggregates the intermediate results produced by the map function. Specifically, for a given term, it performs the following tasks:

Concatenating URLs: For each term, concatenate the URLs of the documents where the term appears.

Data Sharding: After the reduce function completes, the system utilizes consistent hashing based on the term values to store and merge the indexing results on the distributed storage system. This process is employed for data sharding, allowing the query component to efficiently retrieve relevant data segments for querying.

5 Querying and Web Service

In a distributed search engine, efficient querying relies on smart indexing and retrieval strategies[2]. We can achieve this using distributed storage systems, consistent hashing, and the incorporation of TF-IDF (Term Frequency-Inverse Document Frequency) during query processing.

5.1 Query Component

In our search engine, documents and indices are distributed across multiple nodes (servers) to handle large-scale data.

During indexing (using MapReduce), we only know local information (within each shard). For a single term, we don't have a global view of its document frequency (DF) across all shards.

$$TF(t, d) = \text{ frequency of term } t \text{ in doc } d \tag{1}$$

$$IDF = \log\left(\frac{N}{n_t}\right) \tag{2}$$

Where N is the total number of documents, n_t is the number of documents where the term t appears.

In order to incorporate TF-IDF, we can combine the TF results from indexing with IDF during querying [5]: When a user submits a query, we follow these steps:

- Tokenization: Tokenize the query into terms.
- Hashing and Matching: For each term in the query, use consistent hashing to find the corresponding index file.
- Retrieve Local Term Frequencies (TF): Fetch the term frequencies (TF) for each term from the corresponding shards like equation 1.
- Calculate Global IDF: IDF is based on the total number of documents and the number of documents containing the term like equation 2.

- TF-IDF: Combine local TFs with global IDFs to obtain TF-IDF scores for each term.
- Ranking: Rank documents based on their TF-IDF scores in equation 3.

$$TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D) \tag{3}$$

5.2 Web Service

After the implementation of the crawling, indexing, and querying components, our distributed search engine is ready for deployment. We have chosen **Amazon EC2** for its robust and scalable infrastructure, which is ideal for handling the demands of a distributed system.

Accessibility and User-Friendliness: To ensure that our search engine is accessible and user-friendly, we have developed a simple web interface. This interface allows users to interact with our search engine without needing to understand the underlying complexities. We follow the deployment steps:

- HTTP Backend Server: Since our nodes are using the HTTP PUT method to communicate, we can easily implement a simple back-end server to call the node of the indexing group to search the query keys from the distributed storage system.
- Web Interface: We implemented a UI, which can be run on web browsers to start searching through the indexed books.

By following these steps, our distributed search engine will be up and running, providing a seamless search experience to users as shown in Figure 2 and Figure 4.

6 Deployment

For the deployment of our distributed search engine on AWS EC2, we utilize Amazon Machine Images (AMIs) to streamline the setup process as shown in Figure 5. Here's a breakdown of the deployment steps:

- **Amazon Machine Images**: We begin by creating a custom Amazon Machine Images(AMI) that includes our base code and a start-up shell script. This AMI serves as a template for launching multiple EC2 instances with identical environments, ensuring consistency across our distributed system.
- **EC2**: Using the custom AMI, we launch the required number of EC2 instances. Each instance is configured with the necessary resources (CPU, memory, storage) to handle the demands of web crawling and indexing tasks.
- **Security Groups**: Since our nodes and web services run on different ports, we need to create security groups for the EC2 instances and define inbound/outbound rules for the ports.

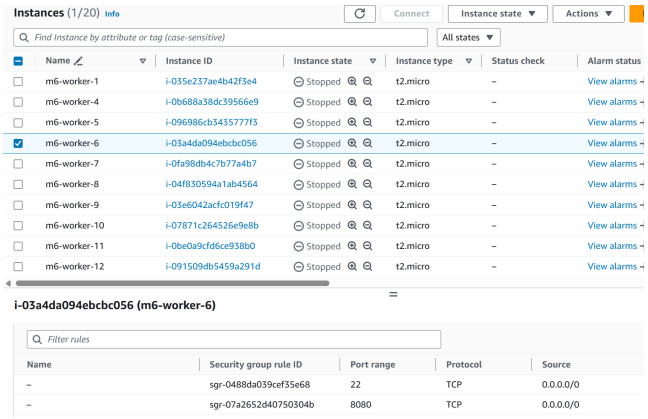

Figure 5. AWS EC2 Deployment

To deploy our distributed system, we manually instantiated EC2 instances that we then configured to be able to communicate with one another to run our benchmarks. Between the five people in our group, we were able to test on a maximum of 100 't2.micro' EC2 nodes.

7 Evaluation

The evaluation of a search engine's performance on single-node versus multi-node configurations is a critical aspect of distributed computing research. This subsection will explore the rationale behind conducting such an evaluation, which is twofold:

Firstly, it provides insights into the scalability of the system. By comparing the performance on a single node against that on multiple nodes, we can understand how well the system utilizes additional resources and whether it can handle larger datasets efficiently. This is particularly important for search engines, which must process vast amounts of data rapidly.

Secondly, the evaluation sheds light on the fault tolerance and reliability of the system. Multi-node environments are more complex and can introduce new challenges, such as network latency and synchronization issues. Understanding how the system behaves under these conditions is essential for ensuring robustness and continuous service availability.

In this subsection, we will delve into the outcomes of our evaluations, discuss the implications of our findings, and consider how they can inform future enhancements to the search engine's architecture and design.

7.1 Scraper Evaluation

Figure 6 shows the execution time of the crawler workflow executed with different numbers of nodes (EC2 instances).

One thing to note about Figure 6 is that with less than 20 nodes, crawling 100,000 web pages actually failed with our implementation. This is why execution times for crawling

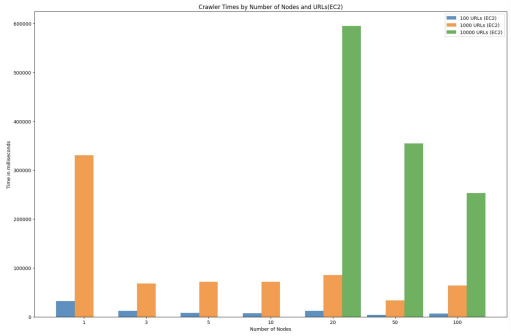

Figure 6. Crawler and Scraper Benchmarks

```
➜ m6 git:(main) X node ./urls/loadUrls.js --urls 1000
[loader]
count of nodes: 100
count of urls: 1000
procedure time: 1026.3160 milliseconds
➜ m6 git:(main) X ./checkHash.sh
Number of nodes: 25
Number of files under ./store/cccfa/groups/crawler: 44
Number of files under ./store/3a1ca/groups/crawler: 46
Number of files under ./store/c2a08/groups/crawler: 40
Number of files under ./store/47e33/groups/crawler: 44
Number of files under ./store/6fcd3/groups/crawler: 35
Number of files under ./store/f9b78/groups/crawler: 43
```

Figure 7. Uneven Data Distribution (100 nodes)

100,000 URLs are only present with distributed systems with 20 or more nodes.

Another anomaly we found with our graph is that there is not a significant change in execution time between crawling 1000 pages between 3, 5, and 10 nodes. This is due to an uneven amount of work assigned to each node during the map phase, the cause of this unbalance is discussed further in the Discussion section.

In our implementation, we employ SHA-256 for hashing keys of objects and even the id of nodes, which are then converted to hexadecimal format. This process is integral to our naive, consistent hashing or rendezvous hashing mechanism, designed to distribute URLs across a network of nodes. Ideally, with 100 nodes, the expectation is to have an even distribution, where each node is responsible for approximately 10 URLs.

However, upon inspection, we observed an anomaly: only 25 nodes were allocated URLs, each handling around 40 URLs. This uneven distribution contradicts the principle of consistent hashing, which aims for a balanced load across all nodes.

The discrepancy disappears when the system scales to 101 nodes with the same number of URLs (1000), resulting in an even distribution of approximately 10 URLs per node. This observation led us to hypothesize that the issue lies within the `parseInt(sha256, 16)` function used to convert the hash into an integer. The suspected cause is a precision loss during this conversion, which could lead to hash collisions.

```
➜ m6 git:(main) X node ./urls/loadUrls.js --urls 1000
[loader]
count of nodes: 101
count of urls: 1000
procedure time: 920.7292 milliseconds
➜ m6 git:(main) X ./checkHash.sh
Number of nodes: 101
Number of files under ./store/cf224/groups/crawler: 10
Number of files under ./store/97fc3/groups/crawler: 9
Number of files under ./store/09a96/groups/crawler: 10
Number of files under ./store/ec716/groups/crawler: 9
Number of files under ./store/082dd/groups/crawler: 12
```

Figure 8. Even Data Distribution (101 nodes)

Consequently, multiple key IDs may end up being routed to the same node, causing uneven distribution.

To address this, we propose investigating alternative parsing methods that preserve the full precision of the SHA-256 hash. Additionally, implementing a collision resolution strategy could mitigate the impact of any potential hash collisions. By refining these aspects of our hashing process, we aim to restore the intended uniformity of our consistent hashing distribution.

7.2 Indexer Evaluation

The indexing subsystem generates a reverse index that maps n-grams (1-grams, 2-grams, and 3-grams) to a list of (URL, frequency) pairs. The URL is the link to a webpage that the n-gram appears in. The frequency is the number of times that the n-gram appears on that page.

Figure 9 shows the execution times for the indexing subsystem to generate these reverse indexes for the page content of 1 and 10 URLs. Surprisingly, we saw that the indexing subsystem took about 1 minute and 48 seconds to index a single page, and took 4 minutes to index 10 pages. Attempting to index larger page amounts results in execution times that we believe to last over an hour with our current implementation (we stopped trying after indexing took over 30 minutes).

We believe that the long execution times come from inefficiencies in the indexing process as well as potential unnecessary overhead with our distributed service methods. Another potential cause of the long execution times is our uneven distribution of map-reduce tasks described in detail in the crawler evaluation section.

7.3 Query Evaluation

As the number of nodes within a distributed system increases, a corresponding increase in query latency is often observed as shown in Figure 10. This phenomenon can be attributed to the complexities introduced by data sharding and the intercommunication required among the nodes. The inter-node communication required to process queries in a distributed environment can compound latency. Each node must often wait for responses from other nodes before proceeding,

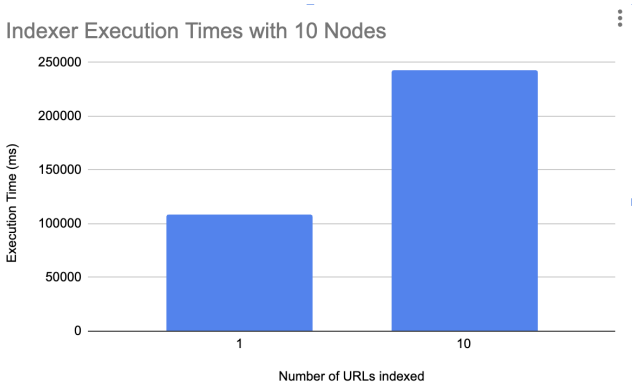

Figure 9. Indexer Benchmarks

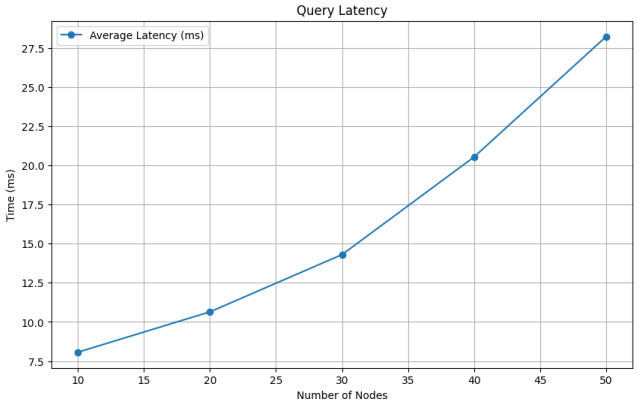

Figure 10. Query Latency

which can introduce delays, especially as the network of nodes grows larger.

However, this increase in latency is a trade-off for enhanced scalability and fault tolerance. With a larger number of nodes, such as 50, the system is designed to handle a greater load, thereby increasing its capacity to scale. This scalability ensures that the system can accommodate growth without a proportional increase in latency or degradation in performance.

Fault tolerance is another critical advantage of a higher node count. The system's ability to continue functioning even when one or more nodes fail is crucial for maintaining availability and ensuring that queries can still be processed. With 50 nodes, the loss of a single node has a relatively minor impact on the system's overall capability. In contrast, a system with only 10 nodes would suffer a more significant loss of capacity in the event of a node failure.

In essence, while a larger number of nodes may lead to higher query latency due to data sharding and inter-node communication, the benefits of improved scalability and fault tolerance are substantial. These benefits are particularly valuable in scenarios where the ability to handle large volumes of queries and withstand node failures is paramount.

8 Discussion

Reflections: Building a distributed search engine was a complex and enlightening process that involved several key phases. Initially, we focused on designing a scalable architecture, choosing between distributed databases and file systems based on our data management needs. Throughout the implementation, we tackled challenges related to data partitioning and load balancing to ensure efficient data distribution and quick query responses. Testing played a crucial role, as we rigorously evaluated each component to identify and rectify issues related to concurrency, data integrity, and system failures.

The paper alone took 8 hours to complete; the poster took about 5 hours.

In total, the distributed version of the project ended up taking 3000 lines of code (LoC).

The code for the distributed system was significantly more complex than that of a non-distributed system. The distributed architecture required additional mechanisms for managing communication between distributed components, handling failures, and ensuring equal data distribution across different nodes. This complexity was reflected in the increased LoC and the need for sophisticated concurrency control mechanisms to manage simultaneous operations across multiple nodes. Furthermore, robust error handling and recovery procedures were essential to deal with potential network failures, node failures, and data replication issues.

The original prediction from initial estimates (M0) ranged between 2000–2500 LoC. The actual LoC for the distributed system ended up being higher, primarily due to the added complexity needed for ensuring robust fault tolerance and scalability.

Differences in these numbers across team members can be attributed to various factors. Some team members focused on developing core functionalities like data partitioning and load balancing, which required writing more complex and extensive code. Others concentrated on integration and testing, which involved less coding but more debugging and optimization. This division of labor was strategic, allowing members to specialize and focus on different aspects of the project, ultimately enhancing efficiency and effectiveness in tackling the complex challenges posed by a distributed system.

One of the reasons for the initial underestimation of the code complexity was a lack of accounting for the intricate details involved in ensuring data consistency and fault tolerance in a distributed environment. The prediction did not fully anticipate the challenges associated with network communications and the need for robust error-handling strategies, which significantly increased the coding requirements.

Limitations: Despite the significant advancements made in developing our distributed search engine, one area where our solution fell short is in the robust handling of fault tolerance. This limitation has had noticeable impacts on system reliability.

- **Single Points of Failure:** While efforts were made to distribute data and processing across multiple nodes, certain critical components of our infrastructure remained centralized, such as the MapReduce coordinator. This centralization resulted in single points of failure that could lead to partial or total system outages during node failures.
- **Inadequate Replication Strategies:** We did not have a replication strategy thus data availability during simultaneous failures of multiple nodes could not be guaranteed.
- **Reliance on Callbacks:** The callback mechanism is susceptible to failure when there are network issues. Over-reliance on callbacks in MapReduce diminishes the ability of our system to be fault tolerant.

9 Related Work

In our Related Work section, we analyze the deployment of distributed databases versus distributed file systems in search engine architectures. Distributed databases are often favored for search engines requiring dynamic and complex querying capabilities, particularly useful for systems that need to manage structured data with real-time responsiveness and transactional consistency. This makes them ideal for search engines that frequently update their indices and respond to user queries instantly. Conversely, distributed file systems are preferred for their scalability and efficiency in handling vast amounts of unstructured data, such as logs, documents, and text files. They are particularly beneficial for search engines that rely on batch processing techniques to periodically update indices, offering a cost-effective solution for managing large-scale data. This comparison underscores the need to align technological choices with the specific operational demands and data processing requirements of search engine projects.

10 Conclusion

In conclusion, the development of a search engine based on MapReduce represents a significant achievement in the field of distributed computing and information retrieval. The implementation of our crawler, indexer, and query component, along with the integration of web services, demonstrates a comprehensive understanding of search engine architecture. Utilizing TF-IDF for relevance scoring and conducting extensive evaluations across various node configurations (1, 3, 5, 10, 20, 50, 100 nodes) with the dataset open books ranging from 100 to 100,000 URLs on AWS EC2 instances has provided valuable insights into the scalability and performance of the system [1, 4, 7].

Through this venture, we have gleaned critical insights into the essence of judicious resource allocation within distributed frameworks and the consequential effects of node quantity on processing expediency and precision. The assessments conducted have underscored the inherent trade-offs between computational expenditure and search efficacy, thereby charting a course for subsequent refinement endeavors. Moving forward, there are several avenues to explore:

Moreover, the journey has unveiled certain imperfections within our code—specifically, the uneven distribution of data sharding attributable to precision loss. This discovery underscores the complexities and challenges inherent in constructing large-scale distributed systems. It serves as a reminder that while we strive for scalability and fault tolerance, attention to detail in data management and algorithmic accuracy remains paramount.

As we forge ahead, the path is rife with opportunities for enhancement and innovation:

Algorithmic Enhancements: Delve into the exploration of sophisticated ranking algorithms that transcend TF-IDF, such as PageRank, to augment search pertinence.

Real-time Indexing: Devise and implement methodologies for instantaneous data indexing, thereby ensuring the search engine mirrors the latest information available.

Code Optimization: Address the identified bugs by refining data sharding techniques to mitigate precision loss and achieve a more equitable distribution of data across nodes.

Scalability vs. Performance: Continue to investigate the delicate balance between scalability and performance, aiming to optimize both without compromising the system's robustness or user experience.

These initiatives are poised to not only elevate the operational efficacy of our search engine but also to contribute to the broader discourse on the trade-offs and optimization strategies pivotal to the evolution of large-scale distributed systems[7].

References

[1] Emily Ann Belli, Jeff Candy, Igor Sfiligoi, and Frank Würthwein. 2022. Comparing single-node and multi-node performance of an important fusion HPC code benchmark. In *Practice and Experience in Advanced Research Computing (PEARC '22)*. Association for Computing Machinery, New York, NY, USA, Article 10, 4 pages. https://doi.org/10.1145/3491418.3535130

[2] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks* 30 (1998), 107–117. http://www-db.stanford.edu/~backrub/google.html

[3] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[4] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European*

Conference on Computer Systems 2007. 59–72.

[5] Bing Liu. 2011. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data* (2nd ed.). Springer Publishing Company, Incorporated.

[6] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2008. *Introduction to Information Retrieval*. Cambridge University Press, USA.

[7] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, USA, 10.