DUSE: A DISTRIBUTED SEARCH ENGINE FOR USENIX PAPER

Yuanfeng Li
Brown University
yuanfeng_li@brown.edu

Jingyu Tang
Brown University
jingyu_tang@brown.edu

Mingchao Zhang
Brown University
mingchao_zhang@brown.edu

Xinqian Zhou
Brown University
xinqian_zhou@brown.edu

Abstract

This paper presents the design, implementation, and evaluation of DUSE, a distributed and scalable search engine developed for CS1380 at Brown University.

1 Introduction

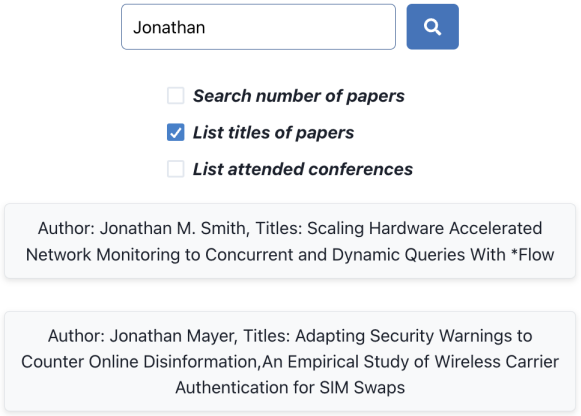
Computational tasks started out with a single machine. To speed up the task processing time, people have been focusing on vertically scaling a single computer to make it more powerful until the formal introduction of the map-reduce model. The map-reduce model enables both scalability and fault-tolerance, and thus enables the wide adoption of distributed systems. To understand the implementation details and performance benefits of a distributed system, we built a distributed search engine for USENIX. The search engine crawls papers listed on USENIX, builds an index and allows users to query aggregated information of an author such as the number or titles of papers an author has written. Both the crawling phase and the query phase perform map-reduce to significantly reduce the processing time as shown in Section 6.1. In particular, we observe that deploying three nodes achieves a speedup of 3 compared to deploying just one node. Throughout the project, we carefully evaluate different options of implementing the mapper and the reducer functions, and the optimal design decisions are explained in Section 3, Section 4, Section 5. Due to the nature of a distributed system, we were able to easily divide the system into different components and work on them separately, which are listed in Section 10.

2 Running Example: Searching for Authors

There are two ways to interact with DUSE: the command-line interface or the front-end website. To interact with the command-line interface, users need to provide two parameters for the query process: an operation code and a search keyword. The operation code is either 1, 2 or 3 which correspond to the number of papers, paper titles or conferences an author has attended. The search keyword can be any string. The following is an example when a user wants to search the number of papers of authors whose names contain the string, "Smith".

```
operation: numberOfPapers, search term: Smith
==================== Result😄 ======================
[
  { author: 'Jonathan M. Smith', numberOfPapers: 1 },
  { author: 'Matthew Smith', numberOfPapers: 1 },
  { author: 'Sean Smith', numberOfPapers: 1 },
  { author: 'Sean W. Smith', numberOfPapers: 1 }
]
```

The front-end website is more straightforward. Users can select the attribute of an author they are interested in, and type in the name of the author in the search box. The following is the rendered results when users want to see papers written by authors whose names contain "Jonathan".



Note that the back-end application does a sub-string matching during the query process, and it can also support more accurate search keywords such as "Matthew Smith".

3 DUSE Design Overview

DUSE is composed of three main components: Crawler, Indexer, Query. Figure 1 shows an overview of DUSE.

The crawler is made up of three essential components: a page crawler, an article crawler, and a content crawler. Beginning with a seed URL, the page crawler systematically traverses every page within the USENIX website, capturing and storing page data either in a distributed page storage or a file system. Subsequently, the article crawler reads from the storage about each page's information and crawls all articles on each page. The content crawler then processes the article hyperlinks provided by the article crawler, extracting details

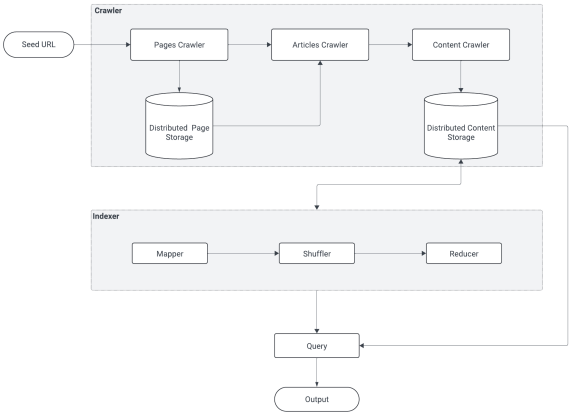


Figure 1. DUSE overview. All components in the grey boxes indicate distributed services.

such as authors, abstract and affiliations. The final result is stored in a distributed content storage or a file system for further processing.

The indexer has three functions: a mapper, a shuffler, and a reducer. Together, they do a MapReduce on the content collected by the crawler. Each function reads the content, overwrites it, and saves the updated data back.

The query exposes a user interface which allows three types of query. Given a search term, the use can specify what to return, number of papers and the titles of papers authored by the individuals, and conferences the author attended.

4 DUSE Design Details

This section will outline the design details of DUSE. It focuses on the architecture and functionality of each component. It will describe how these components interact and support the overall system.

4.1 Crawler

4.1.1 Two Level Indexing

Since the USENIX website employs a paged data structure, in order to ensure controllability and manageability of the crawling process, we adopt a two-level indexed crawling approach rather than recursive crawling. Specifically, we first utilize the URL of each page as the primary index for crawling, and partition these primary indexes across distributed nodes via consistent hashing. Subsequently, the distributed nodes retrieve the content of each page via the primary index and persist it locally.

4.1.2 Integration into Distributed Frameworks

Based on the aforementioned design, we integrated it into our existing distributed system in two phases. In the initial phase, an orchestrator node, which also functions as a worker node, performs a centralized crawl of the pageUrl data. Since USENIX contains only 345 pages of data, this centralized crawling will not incur significant performance overhead.

Upon successful completion of the crawl, the orchestrator invokes the distributed storage service to disseminate the primary indexes to each node. The phase 1 is shown by Figure 2.

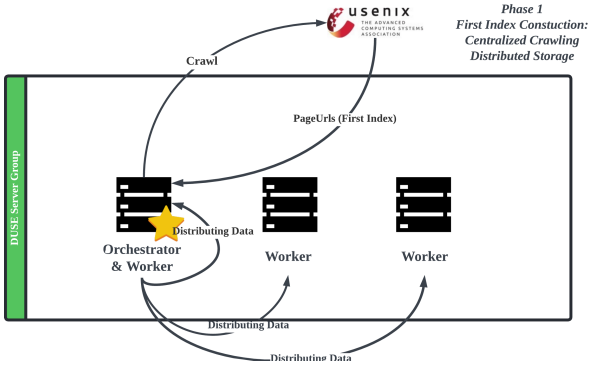


Figure 2. DUSE Crawler phase 1. Construct the first indexing with centralized crawling and distributed storage.

In the subsequent phase, all worker nodes can retrieve the pre-stored primary indexes through the distributed storage service, initiate the distributed crawling process, and subsequently, leverage the distributed storage service once more to disseminate the acquired data to the corresponding local storage of each worker node. The phase 1 is shown by Figure 3.

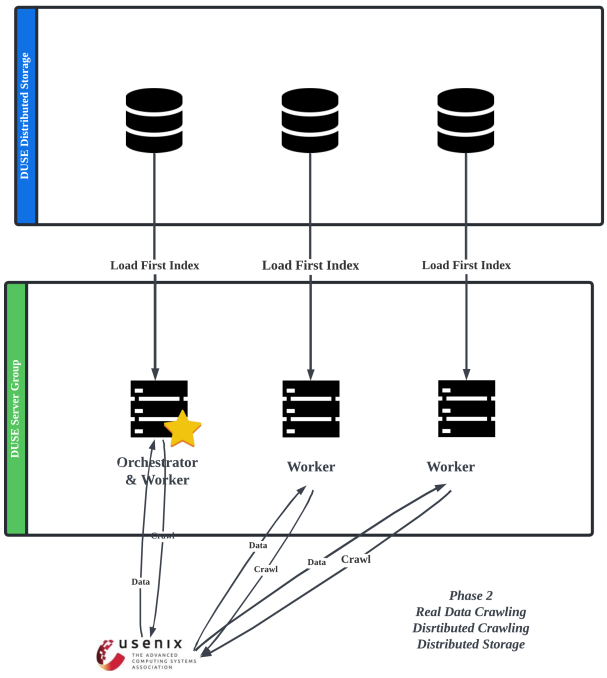


Figure 3. DUSE Crawler phase 2. Complete crawling data with distributed crawling and distributed storage.

4.2 Indexer

After the crawling phase is complete, all crawled data is stored in the distributed system. Leveraging the MapReduce architecture, our system efficiently processes vast amounts of data in parallel, harnessing the benefits provided by distributed computing.

4.2.1 Map phase

Upon initiation by the orchestrator, the mapping phase commences, with the orchestrator dispatching mapping commands to all worker nodes. Each worker node then extracts the stored data generated by the crawling process. To serve various purposes, the data on each worker undergoes processing by custom-designed mapping functions, extracting relevant information for subsequent phases.

4.2.2 Shuffle phase

Following the completion of information extraction by each worker node, the shuffle phase begins. Utilizing consistent hashing techniques, the mapped data is redistributed across the worker nodes based on their respective keys, with identical keys being grouped together on the same worker node for reduction.

4.2.3 Reduce phase

Once shuffling is complete, each worker node retrieves the grouped data from its storage and applies custom reduce functions to process the data, yielding the final results of the MapReduce process.

4.3 Query

Querying is the primary way users interact with the system. Once all data has been processed and stored, users can input their search keywords to retrieve results. The system then dispatches the search term to all worker nodes, which each independently search their stored data for matches. Upon completion, the worker nodes return their respective matching results to the orchestrator. The orchestrator aggregates these results and compiles them into a comprehensive set, which is then returned to the user

5 Implementation

5.1 Crawler

For the Crawler component, we furnish a crawler service. It comprises two methods: **getPage**, which is employed in stage one to construct the primary index, and **getArticles**, which is utilized in stage two to consummate the entire data crawling process.

For data crawling, we employed the Crawler package provided by Node.js. It facilitates crawling HTML in the form of a task queue and furnishes callback functions for corresponding custom HTML processing. The rationale behind opting for Crawler over JSDOM lies in Crawler's asynchronous nature, which endows us with timeout, retry, delayed operation, and other mechanisms to enhance the reliability of our project. However, the introduction of Crawler also significantly augments the complexity of coding, as we have to grapple with more intricate callback processing logic.

For distributed storage, we use sha256Hash to generate keys and consistent hashing to distribute the data.

5.1.1 Phase 1: getPage

Since the construction of the primary index leverages centralized crawling, the getPage method accepts a BaseUrl, enqueues it in the Crawler queue, extracts all URLs upon completion, and disseminates the results to the individual worker nodes utilizing the distributed store service. We wrap all the pageURLs with a hash as the key during the storage process. This measure is not undertaken to ensure a fixed length for the keys in the distributed store, but rather for data security purposes - to preclude unauthorized entities from arbitrarily accessing data in our database based on the URL.

5.1.2 Phase 2: getArticles

In the getArticles method, each node receives a series of pageURLs as input. For each node, it maintains two distinct Crawler queues: the pageURL queue and the articleURL queue. The pageURL queue's input comprises pageURLs, and in the callback function, it extracts a series of articleURLs based on the crawled HTML from the pageURL. These extracted articleURLs are then enqueued into the articleURL queue. Subsequently, the articleURL queue processes the crawled HTML accordingly, including converting it to a string, removing invalid characters, and utilizing the resultant hash as the key to transform it into the required object. Eventually, when the control value of articleURL, i.e., cnt2, is reduced to zero, it signifies that all data has been processed. At this juncture, we invoke the distributed storage service to disseminate the data to each node once again. The process in shown by Figure 4.

5.2 Indexer

5.2.1 Mapper

During the mapping phase, the data previously stored in object-like string format from the crawling phase is retrieved and deserialized into JavaScript objects. Information such as author name, paper title, and conference name is then extracted from these objects. The objective of our mapper is to compile a list of authors along with their paper titles and conference names.

However, since the original data obtained from the website and crawling process may contain inconsistencies, the mapper first conducts data cleaning to remove any unexpected

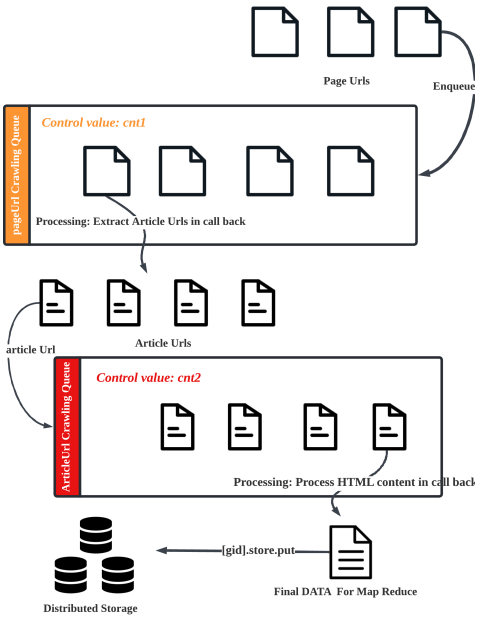


Figure 4. The illustration of getArticles in each node

words or artifacts. Subsequently, the mapper assembles the extracted information, using the author name as the key.

5.2.2 Shuffler

Following mapping, the shuffle phase redistributes the data across nodes using consistent hashing. Data with the same key is appended to corresponding files on respective nodes.

5.2.3 Reducer

After shuffling, the reducer then retrieves this grouped data. For each file, representing each author, the reducer counts the number of papers contributed by the author and lists out all paper titles and conference names. This information is then stored back on the node and awaits the querying phase.

5.3 Query

After the MapReduce process completes, all data is processed and ready for querying. Users input their search terms and select one of three operations: 'search number of papers', 'list titles of papers', or 'list attended conferences'. In the query phase, the search term and operation number are sent to all worker nodes. Each worker node searches its storage for filenames containing subsequences of the search term and returns the relevant data based on the operation number. The orchestrator gathers all data returned by nodes, aggregates it, and returns it to the front end for user access.

5.4 Deployment and Front end

Our project is deployed and running on an AWS EC2 Linux instance. Our setup consists of 1 coordinator node and 3 worker nodes for map-reduce related tasks. The coordinator node acts as a proxy so that in the back end, we can adjust the number of worker nodes without having users to modify the IP address in their requests. The coordinator node listens on port 8080 which has been configured to take HTTP requests asynchronously. Therefore, the front-end needs to get text input from the UI and create an http request with the format: http://{server IP address}:{proxy node listening port}/query/queryNumberOfPapers?ops={1, 2 or 3}&keyword={'author name'}. Similar to the request to other distribution modules, the url asks the queryNumberOfPapers method of the query service to handle the request. After the map-reduced query, the coordinator returns a list of author attribute pairs which are rendered on the front end.

6 Evaluation

In this section, we evaluate the scalability and correctness of DUSE. We examine how well the system performs under various conditions by simulating different workloads. All results are averaged over 10 runs.

Deployment specifications: The evaluation runs locally on an Apple MacBook Pro with M1 chip and 16 GB memory.

6.1 Scalability

For scalability, we conducted evaluations across three key components: the crawler, indexer, and query functionality. During crawler evaluation, the system traverses every page and article within the USENIX domain, distributing and storing the outcomes across multiple nodes. Indexer evaluation, on the other hand, runs after the articles are persisted on disk, ensuring independent testing of the crawler and indexer functionalities. The outcomes of these evaluations are detailed in Figure 5. Overall, the runtime of both components, separately and combined, decreases as the number of nodes grows.

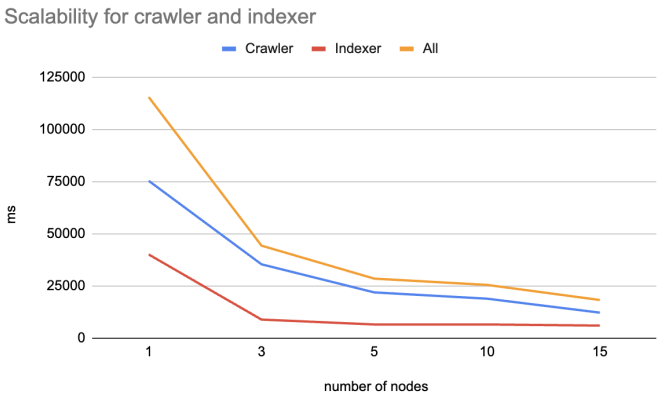


Figure 5. Scalability for the crawler and indexer over different number of nodes.

Regarding the query component, scalability testing was conducted using the search term "Liu", a common surname,

ensuring widespread presence across most nodes. The experiments are conducted with the corresponding search specifications (number of papers, published papers and conferences) following the completion of the indexer phase. The corresponding results are depicted in Figure 6. Overall, the runtime of query component does not decrease as the number of nodes grows.

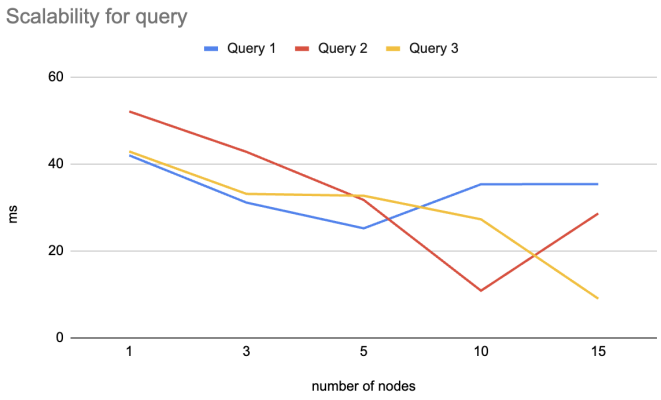


Figure 6. Scalability for the query over different number of nodes.

6.2 Correctness

To ensure the correctness of our system, we developed several JavaScript unit tests for the crawler and the indexer. Additionally, we conducted tests on our query system using a small, controlled set of articles. Overall, the results confirm that the system performs according to expectations.

Crawler: the corresponding unit tests can be found here. Before each run, the local storage should be cleaned up.

Page crawler: given the main page of USENIX, the page crawler function retrieves 345 pages as expected.

Article crawler: given page 345, the articles crawler function correctly stores 17 articles.

Content crawler: given an article url *https://www.usenix.org/conference/usenixsecurity24/presentation/cheng-xiaoyu*, the content crawler successfully stores the authors, title, and abstract in the storage.

Indexer and Query: Indexer and Query shared the same unit test which can be found here. The test case involves crawling a single page of the website, which contains 17 articles. Identical search terms and operation numbers are used across the test cases to ensure consistency. Upon execution, all resulting outcomes match the expected results, validating the accuracy and reliability of index and query functionality.

7 Discussion

7.1 Limitations

As indicated in Section 6.1, while the runtime of both the crawler and indexer decreases with an increase in the number of nodes, the difference is not substantial between 5 nodes and 15 nodes. This observation may stem from the relatively moderate total number of articles, approximately 8600. Consequently, the system achieves optimal performance at around 5 nodes.

Furthermore, despite the query's scalability inconsistency, there is an overall reduction in runtime. This discrepancy may arise from several factors, including significant variations between individual runs, uneven distribution of articles across nodes, and concurrent background tasks running on the computer.

Another limitation of DUSE is its lack of support for search terms beyond those explicitly specified. Additionally, when a node reaches the rate limit, it will be blocked from accessing the website, leading to system faults which deviates from the intended design.

7.2 Reflections

- Roughly, how many hours did the paper take you to complete?
  Hours: 12h
- How many LoC did the distributed version of the project end up taking?
  DLoC: 3000
- How does this number compare with your non-distributed version?
  LoC: 2000
- How different are these numbers for different members in the team and why?
  For all members, distributed systems are more complex than we expected, and implementing them via JavasScript adds even more complexity to the code, resulting in significantly more code than we would have otherwise expected.

8 Related Work

Our map-reduce implementation is greatly inspired by the foundational work presented in [1].

9 Conclusion

In summary, our project aimed to explore the capabilities of distributed systems through the development of DUSE, a distributed search engine for USENIX. By leveraging the map-reduce model, we achieved significant reductions in processing time, with the speedup scaling linearly with the number of nodes deployed.

DUSE consists of three main components: the Crawler, Indexer, and Query. The Crawler systematically traverses the USENIX website, capturing and storing page and article data. The Indexer employs map-reduce to process this data, creating an index for efficient querying. The Query component provides users with a user-friendly interface to search for authors, papers, and conferences attended

Overall, our experience building DUSE has provided valuable insights into the complexities and benefits of distributed systems. By leveraging map-reduce and carefully designing system components, we have demonstrated the potential of distributed systems to handle large-scale data processing tasks efficiently and effectively.

DUSE's implementation, as well as all the example code and benchmarks presented in this paper, are all open source and available for download in this GitHub repository.

10 Contributions

Yuanfeng Li Designed and implemented the engine. Refined the crawler. Implemented query service. Integrated the components. Worked on the frontend.

Jingyu Tang Developed the crawler component, the crawler's unit tests, and conducted evaluations.

Mingchao Zhang Worked on the mapper function improvement, EC2 instance integration and deployment and the front end.

Xinqian Zhou Developed mapper and reduce function and assisted some of the query implementation.

Acknowledgments

We would like to thank Professor Nikos Vasilakis and all CS1380 staff for the support and help.

References

[1] Ghemawat S. Dean, J. 2004. MapReduce: Simplified Data Processing on Large Clusters. (2004).

A Reflections

Hours for paper: 3 hours per member

B Using our System

The search engine can be deployed on any Linux machine. We deployed it on an AWS EC2 Linux instance and configured port 8080 to accept http requests from any IP address. To start the program, run the following command in the main directory, csci1380-m6:

```
npm install
node engine/App.js
```

The engine will start the crawling process followed by the map-reduce process. After these two phases, the engine will start to read input from stdin and start to listen for http requests.

```
Please enter the operation codes and search term (enter q to exit):
 operation code :
 1 for numbers of paper
 2 for paper title
 3 for conference attandence
```

Users can access the front-end UI through this link. Examples on how to interact with the front-end UI is shown in Section 2.