DEOB: A Distributed Engine for Open Books

Mandy He
Brown University
mandy_he@brown.edu

Helen Huang
Brown University
helen_huang@brown.edu

Tiger Lamlertprasertkul
Brown University
tanadol_lamlertprasertkul@brown.edu

Jialiang Zhou
Brown University
jialiang_zhou@brown.edu

Abstract

This paper presents the design, implementation, and evaluation of DEOB, a distributed and scalable search engine for open books from the Gutenberg project developed for CS1380 at Brown University. DEOB provides user-friendly single-point of interaction that enables both users to query our system and see contextualized book URL results and system administrators to monitor system performance.

1 Introduction

MapReduce is a programming paradigm that has allowed countless companies to achieve massive scalability through its restricted programming model. Specifically, this framework exposes two functions: map and reduce. Through leveraging this model, developers are able to process immense amounts of data across thousands of processors with minimal effort, and this basic but powerful framework has inspired many other more targeted systems like Spark [1], which is a "point" solution for batch processing. Spark exposes a broader range of functions that allow developers to produce solutions for data processing that are even more efficient than the traditional MapReduce.

In this paper, we present DEOB, a distributed search engine that also leverages MapReduce. This search system allows users to query for open books from Project Gutenberg, a library of over 70,000 free eBooks. Inspired by Google's search engine, we wanted to focus our system on resources used by a diverse audience. Hence, we chose books; a resource sought after by people of all ages and backgrounds. Built upon semester-long milestone implementations, our search system incorporates vital workflows and endpoints for workflow execution, performance testing, and easy querying. For a decomposition of our search engine, see Table 1.

Table 1. Component decomposition. The table below summarizes the components comprising our search engine DEOB and their characteristics. Each component was implemented as a group together and unit tested.

Component	Description	LoC
server	Interface that allows users to start/stop crawler and search	~ 190
utils	Utility functions for logging and word processing	~ 125
getBookMetadata	MR Workflow for extracting book metadata	~ 125
getURLs	MR Workflow for extracting more URLs	~ 170
index	MR Workflow for creating inverted metadata	~ 150
workflow	Helpers that start/stop crawler and retrieving statistics	~ 150
search	Returns list of relevant documents given a search query	~ 60

```
// this is an example query request
curl -X GET -d "" "18.227.0.148:8080/book/search?q=science"

// this is an example result
[["Lippincott's Magazine of Popular Literature and Science,
Vol. XII. No. 30. September, 1873",
"https://atlas.cs.brown.edu/data/gutenberg/.../14036-8.txt"],
["Lippincott's Magazine of Popular Literature and Science,
Vol. XII, No. 28. July, 1873.",
"https://atlas.cs.brown.edu/data/gutenberg/.../14691-8.txt"],
...]
```

Figure 1. Query Example This snippet shows a user makes a search query on DEOB, and what the results would look like, which include a ranked list of relevant book URLs with additional metadata about each book result.

2 Example

DEOB allows users to search for words in open books and receive a list of book URLs and additional metadata ranked in order of relevancy based on how many times terms in the search query appear in the book title. Specific details of how the book ranking is determined are discussed in *3.3 Query Subsystem*.

Interacting with a distributed system is complicated for end users, so we decided to build a web server as a single point of interaction for end-users and system administrators. Similarly, debugging distributed systems is inherently difficult, so we created our own logging system to both log statistics about our data and errors received in our system.

To search using DEOB, users can send an `http` request to the server with their search term. The server will then send back a list of book URLs ordered in relevancy. An example of a search can be seen in Figure 2.

Server API documentation

Method	Endpoint	Description
PUT	/crawler/start	Schedule recurring workflow jobs.
PUT	/crawler/stop	Stop recurring workflow execution.
PUT	/crawler/reset-workers	Create necessary groups on allall worker nodes.
GET	/crawler/stats	Get statistic info. about crawler jobs.
GET	/book/search?q=<query-string>	Search for books by words or phrases.

Figure 2. Server API Documentation A complete list of coordinator's available API endpoints

3 Background

DEOB is implemented using the MapReduce framework. This framework uses a primary-worker approach. Within a group of nodes, one node is designated as the primary/coordinator node, while the rest are worker nodes. The coordinator node is in charge of initializing the array of data object keys and splitting it amongst the worker nodes using consistent hashing. Each worker node receives its sub-array from the coordinator node and processes the sub-array before sending the results back to the coordinator. This allows subsets of the data to be processed in parallel across all the worker nodes. It is also worth noting that the MapReduce framework DEOB assumes that the data used in MapReduce is already partitioned and stored on all the worker nodes using consistent hashing.

The MapReduce framework can be split into 3 phases: the map phase, shuffle phase, and reduce phase. The framework's API also exposes two user-provided functions that are used within the map and reduce phase: the map function and reduce function. First, the coordinator node signals to all the worker nodes to start the map phase. During the map phase, each worker node reads the relevant data from its local store. Then, the user-provided map function gets called for each input shard, which is then returned as `(key, value)` pairs. Once all the worker nodes finish the map phase, the coordinator node notifies all the worker nodes to start the shuffle phase. During the shuffle phase, worker nodes send/receive the `(key, value)` pairs to the nodes responsible for working on them. This shuffling is done using consistent hashing on the the `key`, which ensures that all data objects with the same `key` end up on the same node. After the shuffle phase completes, the coordinator notifies all the workers to start the reduce phase. During the reduce phase, each worker node will merge all the `(key, value)` pairs with the same `key`. These values are processed through the user-provided reduce function which combines these values. The results are then sent back to the coordinator node, thus finishing the MapReduce workflow.

3.1 Searching for Books Gutenberg

Our search engine operates on the BrownCS mirror of the open books provided by Project Gutenberg. The BrownCS mirror has a file-system-like structure, which is a combination of directories and book files. Directories contain hyperlinks to other sub-directories or book files, while book files are assumed to contain no hyperlink. However, the BrownCS mirror is not completely tree-like in that directories can contain hyperlinks that can form cycles, as well as contain links that go out of scope into other web domains. Our search engine filters for out-of-scope URLs, and assumes that files that have the `txt` extension are book files while all other web pages are directories.

If the webpage is a directory, then our system will scrape the directory for more URLs to crawl. If the webpage is a bookfile, then our system will crawl the text to extract the book's metadata, which includes the book's author, title, publication date, and language. Both of these processes are explained in detail in the *Implementation* section of this paper.

4 Design & Implementation

In this section, we outline the design and implementation of various components in DEOB. Section 4.1 discusses the overview of the system. Section 4.2 covers web crawling and information extraction. Section 4.3 and 4.4 dives into the structure and construction of search index and the lookup mechanics. Section 4.5 covers how various system statistics are collected and processes. The first 3 subsystems are the backbone of the operation of DEOB.

4.1 DEOB Design Overview

We designed our system using the coordinator-worker model. The coordinator node is in charge of running a webserver that provides HTTP endpoints for both the end users and system administrators. More specifically, the coordinator node provides the following endpoints: `/crawler` and `/book`, where `/crawler` allows users to start, stop, and perform other operations on the crawler subsystem, and `/book` allows users to search for books. The worker nodes are in charge of storing data and running the subsystems.

There are three sub-systems: crawler, indexer, and query. The crawler and indexer subsystems are distributed MapReduce workflows that run concurrently to download, parse, and index books from the Gutenberg project. The query subsystem takes user input and returns URLs ordered in relevancy from the distributed store.

Each node in the system runs the `distribution` package, which contains abstractions for local and distributed actions. Namely, the package runs each node as a HTTP server, allowing for easy communication between nodes and remote procedure calls to be made to other nodes.

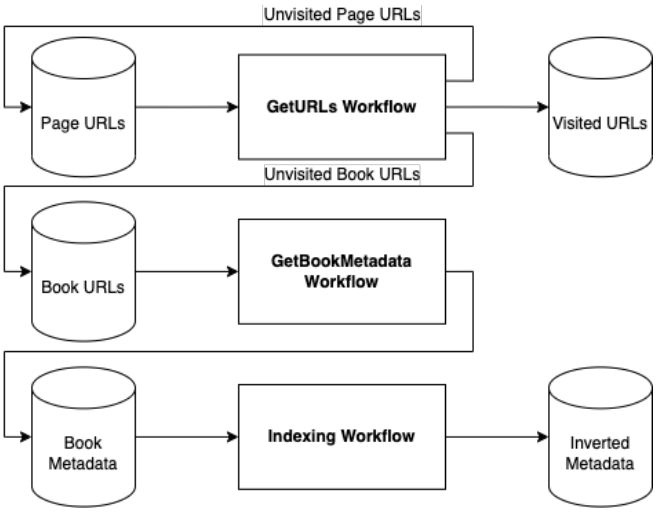

Figure 3. DEOB overview. An overview of all the workflows involved in the DEOB system.

When the coordinator node receives a request, it dispatches corresponding distributed MapReduce workflows to the worker nodes, triggering the workers to perform the necessary actions. There are two benefits to this design:

1. The system is scalable because the coordinator node can easily add more worker nodes to handle heavier workloads.
2. The system is user-friendly because the distributed workflows are abstracted away from the end users and system administrators.

We evaluated our system across four characteristics: performance, scalability, correctness, and fault tolerance. See the Evaluation and Discussion sections for more details about these metrics.

4.2 Crawler Subsystem

The crawler subsystem is responsible for downloading and extracting information of interest from page content. In most traditional web crawler systems, a page downloader and an information extractor are developed and maintained as two separate components. The separation between the two components allow for reusability and easy organization of data. However, we decided to combine the two components to eliminate the need to store raw page content for better storage optimization. The decision was made based on our specific use case given that the Gutenberg database does not change regularly and small inconsistency between the search index and the database is expected and acceptable.

The crawler subsystem comprises two workflows which are executed concurrently: GetURLs and GetBookMetadata. The following subsections provide further discussion on the design and implementation of the two workflows in detail.

4.2.1 GetURLs Workflow

The GetURLs workflow is responsible for downloading webpages from the Gutenberg project and extracting URLs to more webpages or book files. It will first retrieve all the URLs from `uncrawledPageURLs` distributed store and then download the webpages. After downloading the webpages, it will store the urls of crawled webpages in the `crawledPageURLs` distributed store. Then, it will extract the new URLs and store them in the `uncrawledPageURLs` distributed store. This avoids downloading the same webpage multiple times. It can also be seen as idempotent, as the process can be repeated without worrying about the internal state of the system.

4.2.2 GetBookMetadata Workflow

The GetBookMetadata workflow manages the retrieval of book content from URLs listed in `uncrawledBookURLs` and formatting the content to solely preserve metadata. More specifically, this workflow will first retrieve all the URLs from the `uncrawledPageURLs` distributed storage and download the corresponding webpages. Then, it will perform regex matching to extract the necessary metadata and remove unnecessary new lines and spaces to streamline later processing. This cleaned metadata will then be stored in the `bookMetadata` distributed storage group. In an effort to balance runtime and memory usage, this design allows us to avoid storing the entire contents. It is important to note that our implementation can be easily adjusted to store the entire contents of the books; in other words, depending on user and developer needs (e.g. memory limitations or search trends), this workflow is designed to be easily adjusted to process and store the entire contents or just certain aspects of the open books.

4.3 Index Subsystem

The index subsystem processes the book metadata extracted from the `GetBookMetadata` workflow and converts the metadata into objects mapping ngrams to a list of URLs. The ngrams are based on the book titles, and the URLs are the book URLs whose title contains the ngram.

To do this, each worker node retrieves its local partition of book metadata from the `bookMetadata` distributed store, and passes it through MapReduce. The map function takes in the book metadata (i.e. the book title) and corresponding URL, and processes the book metadata string. This involves removing non-alphabetical characters, converting to all lowercase, and stemming and removing stop words. Then, the ngrams are generated, and the map function outputs a list of objects mapping each generated ngram to the URL. In the reduce phase, the reduce function takes in an ngram and list of URLs containing the ngram. It then updates/creates an object that maps the ngram to a list of tuples where each tuple is a URL and count of the number of times the ngram shows up in the book metadata. This data is saved in

the `invertedBookMetadata` distributed store which will be used by the query subsystem.

4.4 Query Subsystem

The query subsystem processes user search queries, retrieves the relevant book URLs, and returns a list of the book URLs order in descending relevancy.

First, the server receives a `GET http` request at the `/book/search` endpoint with a URL query containing the search query. The search query is then processed to remove non-alphabetical characters, converted to all lowercase, stemmed, and removed of all stop words. This is the same processing that happens for book metadata in the Index subsystem. Then, a list of ngrams are created from the processed query. For each ngram, server extracts the corresponding list of url-count tuples from the `invertedMetadata` store. The results for each ngram are then merged to produce a single list of url-count tuples. This list is then sorted in descending order based on count, and the top 10 URLs are returned to the user.

4.5 Performance Logger

The high complexity of DEOB calls for organized and granular logging system. After an execution of a workflow, information about that particular execution is collected and stored locally as a separate entry on the executing worker node or on the coordinator. The information collected includes the time of execution, the total execution time, and the number of key-object pairs processed. The execution time of a MapReduce execution is calculated by subtracting the system time before the execution from the system time right after the execution. We recognize that this number might not be the most accurate representation of the CPU time if multiple processes are running concurrently on the same machine. However, this calculation approach is consistent with how workflows are scheduled to run in a recurring fashion every 1 unit of system time.

4.6 Extra Features

We implemented three extra credit features:

- **Additional metadata:** In addition to returning the list of relevant URLs, we return the book title associated with the URL or the sentence containing the instance of the queried word/phrase. Returning the title allows the user to quickly see what books were returned without having to manually click on each returned book URL. See Figure 1 for an example. Moreover, by returning the contextual sentence, the user is able to see a small excerpt of the original document that holds the query.
- **Debugging infrastructure:** We implemented a `/crawler/stats` endpoint that checks the status of all the worker nodes. For details, see section 3.4 Performance Logger.
- **Fault tolerance:** Our system implements crash-restart on the worker nodes. On the ec2 machines, we schedule a cron job that runs every 1 minute to check if the worker process is still running, if not, it will launch a new worker process on the same port. The coordinator node will pause all work if a worker failure is detected. Once the worker node comes back up, the coordinator will be able to resume previous workflows.

5 Deployment

We deploy our system on a cluster of 5 worker nodes and 1 coordinator node on AWS. All of the nodes are running on t2.micro instances. Note that the number of worker nodes can be easily scaled up or down depending on the workload.

To allow for convenient and scalable deployment, we built a custom AMI (Amazon Machine Image) for the worker nodes and the coordinator node. The AMI for the worker nodes and the coordinator node are mostly identical; they both contain the following:

1. Packages for the EC2 machine to clone and run the system, including `git`, `nodejs`, and `npm`.
2. The github repository containing the system code, which contains the `distribution` package, the `coordinator` package, `ec2` scripts, and other artifacts. Note that the `coordinator` package is present on both the worker nodes and the coordinator node, but only the coordinator node runs the `coordinator` package. The worker nodes do not need packages other than the `distribution` package.
3. A shell script that runs the system on boot. It runs `git pull` on the repository, installs the necessary packages, and starts the system. This is the only difference between the AMI for the worker nodes and the coordinator node. The coordinator node's shell script starts the `/ec2/coordinator.sh` script, which starts the coordinator node's webserver. The worker node's shell script starts the `/ec2/worker.sh` script, which starts the worker node's webserver. We show an example of the coordinator node's shell script in Figure 4.

Additionally, we configure the AWS login and Github ssh keys on the AMI to allow for easy access to the system.

We did not implement any security measures in our deployment. In other words, in order to allow for easy communication between the nodes, we relaxed the security settings on the AWS instances, allowing for all traffic to be sent and received. This is not recommended for production systems. Given more time, we would improve the security of the system by implementing a Virtual Private Cloud (VPC) and setting up security groups to restrict traffic to only the necessary ports.

```
#!/bin/bash

# get current ip of the instance
current_ip=$(curl -s
http://169.254.169.254/latest/meta-data/public-ipv4)

# get all other ips
ips=$(aws ec2 describe-instances --query
'Reservations[*].Instances[*].PublicIpAddress' --output text
| tr '\t' '\n' | awk -v ip="$current_ip" '$1 != ip'
| paste -sd,)

# start the coordinator node
tmux new-session -d -s c
"node /home/ubuntu/m6/coordinator/server.js
--workers $ips --port 8080 --ip 0.0.0.0 --workerPorts 8080"
```

Figure 4. Server example /ec2/coordinator.sh script that starts the coordinator node's webserver.

6 Evaluation

We evaluated our system on a workload of 107386 pages on 5 worker nodes and 1 coordinator node and across four characteristics: performance, scalability, correctness, and fault tolerance. All nodes are deployed on t2.micro instances with a custom AMI built on top of Ubuntu Server 24.04 LTS. They are all using the `distribution` package, and the coordinator node is also running the webserver that users can make HTTP requests to.

For the workloads, we decided to pick 107386 pages to be on par with the recommended load specified for this project (over 100k pages), a nontrivial number of pages.

6.1 Performance

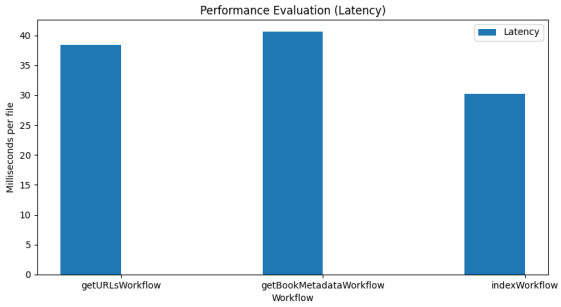

Figure 5. Latency of each MapReduce workflow in milliseconds per file.

To evaluate the performance of our system, we examined the throughput and latency across the different workflows. For latency, we examined how long our system takes to store and index a page by tracking the average execution time and average number of keys processed per call of each workflow. We then calculated the latency by dividing the average execution time by the average number of keys processed. See Figure 5 for our performance results. For throughput, we examined how many pages/files our system can store and index per second. We then calculated the throughput by dividing the average number of keys processed by the average execution time (ms) and made conversions so our results are per second. See Figure 6 for our throughput results.

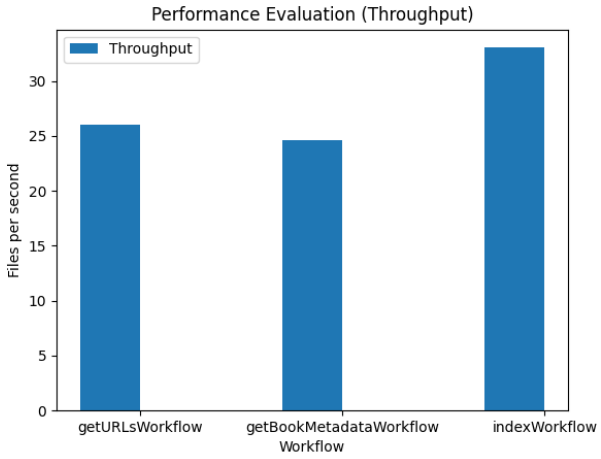

Figure 6. Throughput of each MapReduce workflow in files per second.

6.2 Scalability

We evaluated the scalability of our system by examining the performance of our system on 1 worker node versus 5 worker nodes. See Figure 7 and Figure 8 for our scalability results. The latency and throughput for these figures are calculated in a similar way as mentioned in the Performance section.

We also evaluated the scalability of our system by examining the distribution of storage across the workers. See Figure 9 for these results.

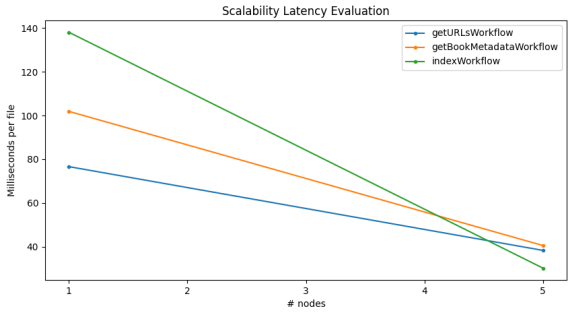

Figure 7. Latency of MapReduce workflows as the number of nodes increases.

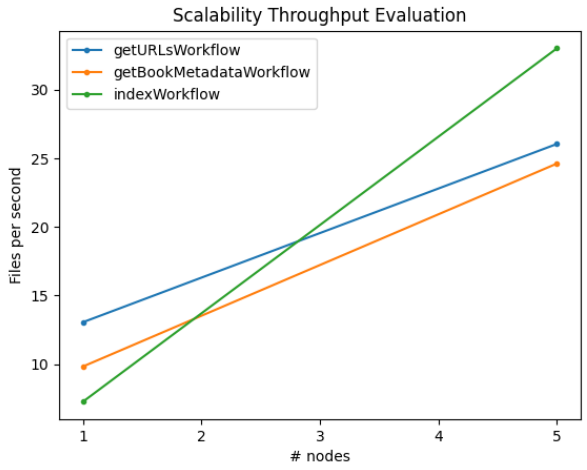

Figure 8. Throughput of MapReduce workflows as the number of nodes increases. Note how the throughput increases with number of nodes.

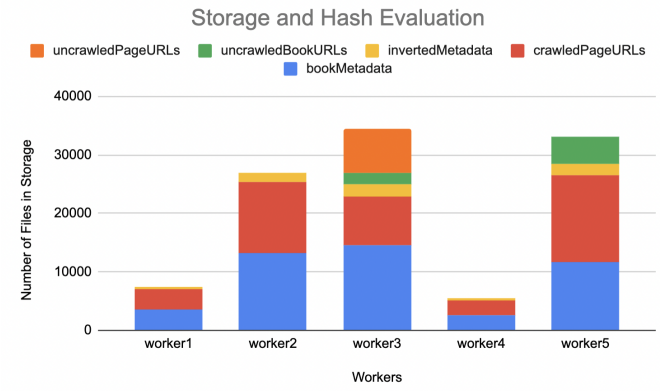

Figure 9. Evaluation of distribution of load/storage across workers

6.3 Correctness

We manually inspected for correctness by calculating the book URL ranking on a few books by hand and comparing it with our system's results. We also validated that our system results contains the same book URL's as the original Gutenberg's results.

6.4 Fault Tolerance

We manually inspected the worker nodes and observed that they were able to come back up after crashing. For our system to detect a failure and restart the node, it takes roughly ~ 15 seconds. Since we pause all workflows if a worker failure is detected, our throughput decreases during this time period, then returns to normal once all the failed workers are back up.

7 Discussion and Reflections

Our results in the Evaluation section mostly align with our expectations. In terms of performance, each workflow has similar latency and throughput. In terms of scalability, latency decreases with an increase in the number of workers and throughput increases with an increase in number of workers. One thing we did find interesting was the distribution of files across workers. As seen by our scalability evaluation results in Figure 9, the distribution of storage is very uneven across our workers. This can lead to bottlenecks in the future in terms of throughput for scalability. Moreover, this indicates that we may not be fully utilizing the resources on certain worker nodes. This could be adjusted by changing our hash technique. Currently, our implementation uses consistent hashing. Future work could include adjusting our hashing to distribute the files across workers more evenly. With further research into common word use and user query trends for books, we could also potentially use different hash techniques for different storage groups so that our system aligns with these trends.

While we accomplished much with this project, another limitation is the security of our system. As mentioned in the deployment section, our system is currently not secure. All worker nodes and the coordinator node allow traffic from all ip addresses. This is not secure because an attacker familiar with the `distribution` package could call `comm.send` to call the `routes` endpoint to place a malicious service on any node. For future work, we could improve the security of our system by placing all nodes in the same VPC, restrict external traffic, and implement different access control on the API endpoints for the end users and the administrators.

Reflecting back to our design and implementation process, our team invested significant effort in brainstorming and meticulously planning our system's design, and this effort paid dividends. Our implementation focused on having a single point of interaction so the end users and administrators would never need to manually inspect individual workers. See Figure 2 for our API. Not only does this design strategy improve usability, but it also makes the development and debugging process easier.

Summarize the process of writing the paper and preparing the poster, including any surprises you encountered. Both the paper and poster were completed as a group. For the poster, we split up each section between us so that we were each in charge of completing and talking about that section of the poster. For the paper, we came up with a list of discussion points for each section of the paper as a team. Then, we split up the sections between us and completed writing each section asynchronously.

Roughly, how many hours did the paper take you to complete? The paper alone took 20 hours to complete; the poster

took about 4 hours.

How many LoC did the distributed version of the project end up taking? In total, the distributed version of this project ended up taking in total ~ 16560, where ~ 1000 LoC came from our m6 implementation. To see the decomposition of our m6 code, see "Table 1. Component decomposition".

How does this number compare with your non-distributed version? The original prediction from M0 ranged between 5000–10000 LoC.

How different are these numbers for different members in the team and why? We mostly coded/debugged this project together, so there is little difference in number of lines of code. Any asynchronous work was used for further debugging or deploying on AWS.

8 Related Work

Our MapReduce framework is based on the paper MapReduce: Simplified Data Processing on Large Clusters. Like the original paper, our framework has a map phase, shuffle phase, and reduce phase. However, it does not have some optional functionalities like a combiner function that is run during the map phase. The fault tolerance is added to the machine running the process instead of the library itself.

However, like the Spark paper, Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, our workflows store intermediate data in-memory, rather than on disk. This improves the efficiency of our workflows significantly, as it eliminates the overhead caused by disk I/O and serialization.

9 Conclusion

DEOB'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/tlamlert/deob.

Acknowledgments

To complete our project, we used code from our previous milestones and minimal suggestions from Co-Pilot (e.g. syntax suggestions). StackOverflow and w3schools were also useful resources, especially for understanding error messages and advice on how to structure regex strings (see code for URL citations). We also received suggestions from Professor Michael Greenberg and Professor Nikos during our poster presentation. We used these suggestions to guide the focus of our paper.

References

[1] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (oct 2016), 56–65. https://doi.org/10.1145/2934664