GAS: Gutenberg Automated Search

Isaias Reyes*
Brown University
ireyespa@cs.brown.edu

Jackson Davis*
Brown University
jdavis70@cs.brown.edu

Josue Cruz*
Brown University
jcruz14@cs.brown.edu

Nick Masi*
Brown University
nmasi@cs.brown.edu

Abstract

This paper presents the design, implementation, and evaluation of GAS, a distributed and scalable search engine for the Project Gutenberg dataset with rapid query capabilities. We leverage the unique structure of the ATLAS Group mirror of the dataset to accelerate our distributed execution engine's computation. This internal engine was built from scratch and follows the actor model for computation. Overall, GAS achieves considerable speedups over non-distributed search engines, even when deployed across just a few nodes. This project was undertaken as part of CS1380 (Distributed Computer Systems) at Brown University.

1 Introduction

Project Gutenberg is a collection of books and other texts in the US public domain (or made available by the author) which is hosted freely online [5]. Its purpose is to serve as an electronic repository and archive of these works to increase access. The ATLAS Group for systems research at Brown University hosts a lightweight mirror of the dataset; however, there is no search functionality across its text. GAS fills this void.

The Project Gutenberg dataset contains >100k entries. To succeed at this scale, we create our own distributed execution engine inspired by Google's MapReduce [3]. Our engine is run across a group of AWS EC2 nodes, and the resultant search engine is sharded across the group to enable constant time queries.

The execution of our distributed engine alternates between crawling and indexing processes as it works through the dataset. This minimizes the memory demands at any given time by not needing to save the whole corpus before indexing it. Through this optimization, we are able to deploy our process on t2.micro nodes with 1GiB of memory each. Each crawl-index step is implemented as one combined job engineered to run on our custom MapReduce engine. We iterate through the dataset with successive crawl-index steps. The final output of crawling and indexing is saved in a distributed store across the node group. Through consistent hashing, we are able to uniformly and deterministically assign search terms to the nodes. Querying then amounts to a highly-performant distributed `get` command.

*All authors contributed equally, ordering is alphabetical by given name.

The paper is structured as follows. We begin by explaining the design (§2) behind GAS and provide implementation details (§3) underpinning it. We include an evaluation of the system and its results (§4) and then discuss its limitations (§5). Finally, we lay out future and related work (§6) in the field and conclude the paper (§7).

2 Design

To understand the details of GAS, it must first be understood what is meant by "crawling" and "indexing" in the context of a search engine. A crawl subsystem looks at individual web pages; on each page, it (1) gets the text content on that page and (2) gets a list of other pages (URLs) pointed to by that page. The index subsystem operates on the text of each page. It preprocesses the text (*e.g.*, keeps only letters, lowercases, removes stopwords, stems) and then finds all of the n-grams in the remaining text. In GAS we consider uni-, bi-, and tri-grams; so, all of these for the page are found. At this point the indexer has a page, identified by its URL, and a list of n-grams it contains. This is "inverted" to create a list of n-grams and the page which they are in. This is merged into some repository of all the n-grams, so that you can have a list of all the n-grams in the corpus and the pages each is contained in. This system can then be queried with an n-gram, upon which a list of relevant pages is returned. GAS follows these basic steps. The specifics of how it has been modified to execute in a distributed environment will now be addressed.

A common approach for centralized (*i.e.*, non-distributed) search engines is to have a `URLs.txt` (see: Figure 1). The crawl and index engine is continuously reading and writing from this file as it crawls pages. When it needs to crawl a new page, it pops the top url from the queue; when a page is being crawled, the extracted outbound links are pushed to the end of the queue. However, this would pose significant concurrency challenges in the distributed approach. We avoid this altogether by interleaving crawl and index batches, as was alluded to section 1. This will now be explained in detail.

2.1 CrawlIndex

The purpose of our distributed execution engine à la MapReduce (§3.1) is to run the crawl and index steps of GAS. This

section will assume familiarity with MapReduce, and explain how the crawling and indexing of our engine are designed to work in this framework.

The dataset we look at is structured such that there are 10 top-level directories, within each are 10 more directories and 10 leaf directories, the latter of which contain one book each. There are no links back up this tree, and the books are raw text pages with no outbound links. This means the dataset is not appropriate to for using a link-based relevance ranking algorithm such as PageRank [2]. However, we are able to leverage this structure in the design of our search engine. This is because it is guaranteed that each link will only be visited once if we start from the top of the tree and work down. To do so, we seed our engine with the top-level directory URLs placed in `URLs.txt`. Then, we are able to crawl and index each level one at a time, get the links to the next level down in the tree, and overwrite these into `URLs.txt` before running another iteration of crawling and indexing. We implement crawling in the map phase and indexing in the reduce phase of our distributed execution engine, and thereby dub it CrawlIndex.

Each CrawlIndex step occurs at one layer in the dataset tree. We seed the engine with a set of top-level URLs for the tree placed in `URLs.txt`. Rather than continuous reading and writing to this file, we will crawl and index all of the urls in this file and then overwrite it with the URLs for the next layer down in the tree. At the start of a CrawlIndex job, the coordinator node takes in the `URLs.txt` file, assigns each url a unique pageID, and then shards `{pageID: URL}` entries across the group.

The crawl (*i.e.*, map) phase starts on each node with a subset of these entries. It then fetches the URL to get the text on that page. Outbound links on the page are extracted using `jsdom` and written with our distributed `append` method to a `tmp.txt` file. The text on the page is then preprocessed: tokenized, stemmed, and stopwords removed. The number of occurrences of each remaining n-gram is counted, and as a memory-conscious optimization we keep only the most 100 popular n-grams in a given document. N-grams are then grouped by their first two letters. The intermediate data that gets returned from map and shuffled take the shape of

```
[{first2letters: [{word: word, count: count, url: pageURL}]}]
```

The intermediate keys are aggregated and shuffled out again to the worker nodes by the coordinator node. At this point, the index (*i.e.*, reduce) phase initiates. For this phase the worker nodes work through the key-value pairs they received, which take the shape of those output by the crawl phase. The reducers aggregate records by the first two letters. The end result is a list of objects which are keyed by the first two letters of an n-gram, and the value is another JSON object. In this object, you have each of the n-grams (starting with those first two letters) as the keys and the values are objects with the count for that n-gram in the page and the url of the page.

When the coordinator node gets back from reduce, `tmp.txt` overwrites `URLs.txt` and `tmp.txt` is cleared. This is our key optimization that takes advantage of the Project Gutenberg dataset we run on. We keep our memory demands low at any given time by clearing saved memory in between CrawlIndex runs and additionally there are no concurrency issues with the `URLs.txt` file. At this point, the coordinator also takes all of the key-value pairs it was sent back and then merges and sorts them into the existing distributed index; the top 10 most relevant links are kept for each n-gram.

2.2 Query Design

At the end of all of the CrawlIndex iterations we have the indexed files stored across our group of nodes. Each of the n-grams are organized into a file based on the first two letters of the n-gram. The reasons for doing so are discussed in subsection 3.3. At query time, a user inputs the n-gram they would like to search for. Our query routine tokenizes and stems it, then takes the first two letters. From the hashing employed throughout our node groups, the query routine knows exactly which file will be responsible for the file corresponding to those two letters. It then retrieves the file from the node. All of this is abstracted through our previous work in CS1380 and gets executed by calling the `get` method in the `store` service. This returns a JSON object with all the n-grams starting with those two letters. Then object is keyed into by the queried n-gram, and the retrieved value is a list of URLs. The URLs have been sorted by the counts of the query n-gram within them, and this relevance-ordered list of results is served to the user.

3 Implementation

3.1 Distributed Execution Engine

The building block of our distributed system is the nodes it runs on. As part of CS1380, we built these nodes in accordance with the actor model [1, 6]. Each node maintains internal state and methods, and can message or spawn other nodes. The nodes' methods are organized into services. We have written our own implementation of MapReduce as one of these services. For local execution, we start with a coordinator node. The coordinator node then spawns a group of new nodes, stores data across the group with our sharding service, and broadcasts a message to the group to execute their MapReduce service over the data they have been sent.

The MapReduce service contains just the `exec` method, which runs a MapReduce job. Groups are an abstraction that get treated as a single node according the actor model, but internally each group service is written to communicate to the nodes in that group in order to distribute functionality. The `exec` method takes in the engineered `map` and `reduce` functions to run. It can also take in other configurations such

as whether to run in memory (using the `mem` service) or to save intermediate keys and final results to files (using the `store` service).

3.2 AWS Deployment

To deploy GAS in a truly distributed sense, we create a group of AWS EC2 instances. We then are able to initialize these instances as our actor nodes and run our execution engine across them. We begin with the coordinator node instance, which is spun up manually. We start an express server on the coordinator node using the `pm2` command.

Then, we use the AWS interface to set up all of the worker nodes. When creating an EC2 instance, you can specify that more than one node is created; we utilize this functionality to start as many worker nodes as we desire. In this setup, we also modify the metadata permissions of the nodes so that we can get their private IP addresses. Each AWS account has all of its EC2 instances within the same virtual private network [9, 10], so the coordinator addresses the worker nodes by these private IP addresses (within the 172.16.0.0/12 IPv4 block). Finally, we provide "user data" which is really a bash script that is run on each instance when it starts up. The script we use is:

```
#!/bin/bash

# Update YUM
sudo yum update -y

# Install Node.js
curl -sL https://rpm.nodesource.com/setup_14.x | sudo bash -
sudo yum install -y nodejs

# Install Git
sudo yum install -y git

# Change to the ec2-user's home directory
cd /home/ec2-user

# Clone the project repository
git clone https://github.com/ireyesp107/m6.git
cd m6

# Install npm dependencies
npm install

# Give execute permissions to the script
sudo chmod +x distribution.js

# Obtain the private IP of the instance
IP=$(curl http://169.254.169.254/latest/meta-data/local-ipv4)

# Run the Node.js application in the background
sudo ./distribution.js --ip "$IP" &

# Wait for the Node.js application to start
sleep 10

# Send a POST request to the coordinator node
# to add the worker node to the node groups
curl -X POST -H "Content-Type: application/json" \
-d "{\"ip\": \"$IP\", \"port\": 8080}" \
http://<coordinator_IP>:3000/nodes
```

The script prepares each node with the needed dependencies and a clone of our repository. This `./distribution` command runs the `start` method of the node service on each instance which creates an HTTP server on it that enables inter-node communication, pursuant to the actor model. The final `curl` command posts the worker node's own information to the coordinator node's express server. The coordinator then writes the information to a text file, so that when GAS is initiated the coordinator can read the worker nodes it has available, create a group out of them, and run the CrawlIndex engine over them.

3.3 N-gram Storage and Querying

Our original approach was to have a different file, saved in our distributed store across the node group, for each unique n-gram in the corpus. This ended up bogging down the performance of our system because of the amount of file writes that it incurred and the overhead of opening and closing files. While developing GAS, we started with local tests. For this naive approach, it took `36s` to index the text on three pages (*i.e.*, for three books). To solve this challenge, we ended up grouping all of the n-grams by their first two letters. Each file is then a serialized JSON object wherein the keys are n-grams (all of which, for a given file, have the same first two letters) and the values are the list of page URLs to return for that n-gram and the count of occurrences of that n-gram on the page. This decreases the file operation overhead. After the shuffling of the index step, we can then batch together all of the n-grams starting with the same two letters and write them all to the same file. To perform the file writes, we implemented the `append` extra credit method from M5. The method is added to both our `mem` and `store` services. This approach took our local three page runtime down to `8s`, an over `4x` speedup.

At query time, a user provides either a uni-, bi-, or trigram. The distributed storage services we wrote, and include in GAS, utilize consistent hashing. This means that we can take the n-gram from the query and automatically know which node is responsible for the file corresponding to the first two letters of that n-gram. We then request that file and retrieve it from the node. All of this logic is handled in our `mem` and `store` services that are provided to the nodes as part of our GAS repository which we install on nodes as they are spawned (§3.1). When the file is received by the coordinator node handling the query, it deserializes the file and keys into the resultant JSON object by the query n-gram. This returns the list of URLs already sorted by relevance (as determined by the number of occurrences of that n-gram in the text) to the user.

4 Evaluation

It is first necessary to justify the approach of using a distributed system in the first place. To do so, we show that the COST [8] of the distributed execution in GAS is less than the amount of nodes (n) that we use. That is, we prove that execution time actually decreases when distributing execution across a node group as compared to running it on a single node. This ensures that the overhead of distribution does not negate its benefits.

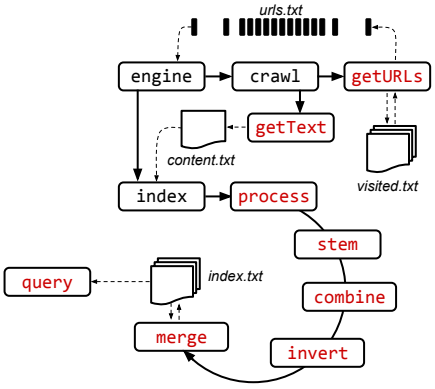

Figure 1. An overview of the crawl, index, and query steps performed by a centralized (n=1) search engine.

To do a COST analysis we measure the amount the time it takes for an engine to completely crawl and index a set of pages. We characterize the set of pages by the approximate amount of text data stored across all of them together. We compare the results of GAS to a centralized (n=1) search engine that we built previously in CS1380. The architecture of this single-threaded engine is shown in Figure 1. As part of that earlier work, we measured the performance of our engine on some contained web sandboxes.[1] We compare that performance to the results of running GAS on Project Gutenberg. The results (Figure 2) indicate that GAS has a COST <7, which justifies our implementation of GAS across 7 nodes. This confirms there are speedups in using the distributed approach of GAS to crawl and index web pages.

[1]These sandboxes can be found at 1, 2, and 3.

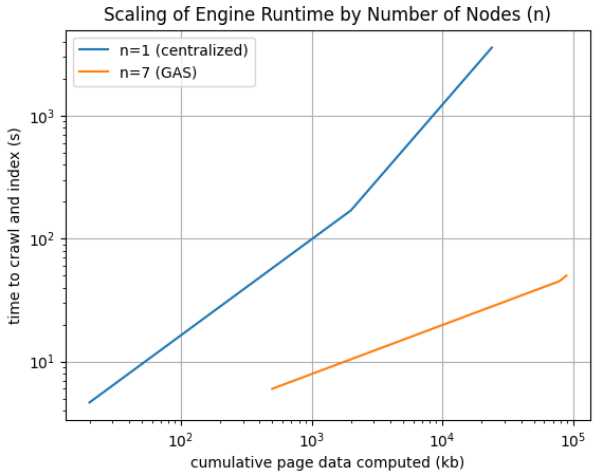

Figure 2. A COST analysis on a centralized search engine versus GAS prove the time benefits and justify the distributed nature of GAS.

4.1 Query

The runtime for a query is in the order of magnitude of milliseconds. An example of this is shown in Figure 3.

```
console.log
  project

    at findInBatch (test/crawl.test.js:2098:17)

console.log
  [
    {
      count: 103,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/2/11122/11122-8.txt'
    },
    {
      count: 103,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/2/11122/11122.txt'
    },
    {
      count: 95,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/7/11179/11179-8.txt'
    },
    {
      count: 95,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/7/11179/11179.txt'
    },
    {
      count: 94,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/5/11159/11159-8.txt'
    },
    {
      count: 94,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/5/11159/11159.txt'
    },
    {
      count: 93,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/1/11117/11117.txt'
    },
    {
      count: 93,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/5/11154/11154-8.txt'
    },
    {
      count: 93,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/1/11117/11117-8.txt'
    },
    {
      count: 93,
      url: 'https://atlas.cs.brown.edu/data/gutenberg/1/1/1/5/11154/11154.txt'
    }
  ]
```

Figure 3. An example of using our command line-based query interface to search for the term “project." This query took in `38.1ms` to run.

5 Limitations

We ran into two main errors when scaling up crawl that limited how much data our system could process. These were (1) running out of file descriptors and (2) and socket hang up errors, both of which were occurring on the worker nodes. We first attempted to resolve (1) by increasing the number of file descriptors available; next, we tried to group extracted links together so that they would be written into the same file; then, we tried to split up the extracted links into different rounds and iterate over them. This last solution fixed issue (1), but resulted in the worker nodes hanging up

To address this, and have a some results to submit, we chose to run GAS over a smaller subset of the dataset. This led to us indexing 157 number of book text files. Additionally, because of how many directory links there are, the total number of URLs crawled is about `10`x this number of books.

6 Future & Related Work

The most beneficial future work would be making our search interface more robust. To start, it would be ideal to allow for search longer than three terms. This could be implemented by taking a sliding window over the query and getting multiple trigrams to search our distributed store for. We would also like to add some search operators. Specifically, the minus ("–") operator would be a good first place to start since it would require minimal code modifications. We could retrieve the list of pages for the n-gram after the hyphen and make sure to exclude these from the results returned to the users. To make this more accurate, we would have to extend the list of pages we store for each n-gram beyond 10. Additionally, we would like to make searches more flexible than exact string matches. This could be achieved by using a trie of n-grams in the corpus and doing fuzzy searching to retrieve results both for the query and other nearby n-grams. Such an approach would also enable spellchecking as the user types their query [7]. Another approach to more flexible queries would be implementing semantic search [4].

As part of making the search interface more accessible, we could create a website to use it rather than having to work through the coordinator's terminal CLI. This would have the additional benefit of not needing to route all requests through the same node. We could provide the client side website code with the same hashing algorithm used by the node group as well as the nodes' IP addresses. Then, the client page could request the n-gram file from the responsible node directly. This would unlock even further benefits of our distributed system as the number of users scaled up.

The implementation of fault tolerant distributed systems was beyond the scope of CS1380, but incorporating this into GAS would significantly bridge the gap between being an academic project and a real-world application. Lastly, highlighting the location of queries within the retrieved pages would also be a significant improvement to GAS. This would also represent a contribution to the Project Gutenberg endeavor overall, as the official search functionality of the library does not contain this feature. One approach to doing so would be storing metadata on the location of n-grams (prior to preprocessing) within the page as it is crawled prior so that this information can be retrieved at query time.

7 Conclusion

The creation of GAS was the most significant technical challenge that we have undertaken. It pushed our abilities to design, code, and debug. Furthermore, it exposed us to a new paradigm in distributed systems programming. Importantly, we got to bridge the theory we had learned in CS1380 (*e.g.*, actor model) with practical development tools (*e.g.*, AWS EC2). By using the MapReduce distributed execution engine we built from scratch, we were able to build a system that scales to massive amounts of data. Ultimately, we achieved a nearly `2000`x speedup over our non-distributed search engine implementation (Figure 2) when normalized by the amount of text data indexed.

References

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press. 8–9.

[2] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Computer Networks and ISDN Systems 30.* 107–117. https://doi.org/10.1016/S0169-7552(98)00110-X

[3] Jeffery Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI '04: 6th Symposium on Operating Systems Design and Implementation.* USENIX. https://www.usenix.org/legacy/events/osdi04/tech/full_papers/dean/dean.pdf

[4] R. Guha, Rob McCool, and Eric Miller. 2003. Semantic Search. In *WWW '03: Proceedings of the 12th international conference on World Wide Web.* ACM, 700–709. https://doi.org/10.1145/775152.775250

[5] Project Gutenberg. 2024. Terms of Use. https://www.gutenberg.org/policy/terms_of_use.html

[6] Carl Hewitt. 2015. Actor Model of Computation: Scalable Robust Information Systems. https://arxiv.org/abs/1008.1459

[7] R. C. Howell. 2020. Fuzzy Prefix Searching with a Trie. https://github.com/RCHowell/rchowell.github.io/issues/4

[8] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *HotOS XV: 15th Workshop on Hot Topics in Operating Systems.* USENIX. https://www.usenix.org/system/files/conference/hotos15/hotos15-paper-mcsherry.pdf

[9] Amazon Web Services. 2024. Amazon EC2 instance IP addressing. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-instance-addressing.html#concepts-private-addresses

[10] Amazon Web Services. 2024. Virtual private clouds (VPC). https://docs.aws.amazon.com/vpc/latest/userguide/configure-your-vpc.html