Compass - Final Report

Chaim "Jay" Fishman chaimj
Varun Jana jvarun
Jong Min Choi jongmin
Viandrudigo Djianto djianto

1 Introduction

"Compass" had a target of 4 million crawled documents based on seed urls that we deem could branch out and encompass the web. Each unique document is an input to the indexer and page rank engines, which generate the weighted TF-IDF and page rank, as well as a custom weight ranking based on token positions hits. Overall, "Compass" as our Google clone, is evaluated against multiple search queries on its speed and accuracy.

1.1 Milestones

Milestone 1: Brainstorming of architecture, create rough database structures and possibly create foundations applications for each component.
Milestone 2: Create and optimize crawler, start crawl of initial corpus. Create foundational indexer, PageRank, TF-IDF, Knowledge Graph
Milestone 3: Run page rank using Livy; crawls 1 million pages
Milestone 4: Have functional search engine locally
Milestone 5: Host search engine in AWS; finish all indexing, final component and integration testing and analysis

1.2 Division of Labor

Chaim: Crawler
Varun: Indexer, Search engine client, System Architecture
Jong Min: PageRank, TF-IDF Engines, InvertedIndex Populator
Vian: Search engine client, Databases

2 Architecture and Implementation

The overall architecture for our project can be seen in the Figure 1 on the last page.

3 Crawler

We designed a Master-Workers distributed crawling system and crawled over 6 million unique documents and around 7.17 million unique URLs. The system is highly malleable, scalable, and fault tolerant. We used the following **seed URLs**
1. https://www.nytimes.com
2. https://www.upenn.edu
3. https://news.google.com
4. https://techcrunch.com
5. https://yahoo.com
6. https://medium.com

3.1 Crawler Architecture

The crawler is built entirely in Java, and our system design is sketched out in Figure 2 (See last page) and is briefly reviewed here: The crawler can be started, paused, resumed and stopped by admin clients via HTTP POST requests to the master. When a request to start or resume a crawl is received, the master machine spawns a thread pool of *URL Preparer* threads. Each of these threads query an external RDS queue, where new URLs are stored and grouped by domain. Each connection queries for 30 domains, and for each domain pops up to 300 URLs from the external queue. For each list of URLs corresponding to the same domain, the *URL Preparer* thread first checks if a document with information about the domain's robots.txt already exist in the database; otherwise it retrieves the robots.txt file via a HTTP request, parses it, and creates a new document for it in the database. Given the parsed robots.txt information, each URL is filtered against the list of disallowed paths for its domain and then put into the *URL Ready Queue*. The *URL Ready Queue* maintains an internal priory queue ordered by each URLs crawl-delay expiration.

Each worker machine has a *Master Pinger* thread, which periodically pings the master. The master responds with its *Run Context*, which contains information about its current. When a worker sees that the master's state has changed to "running", it will spawn up all of its thread pools and begin contacting the master through its *Master Communicator* thread.

Each time the *Master Communicator* thread on a worker machine makes a request to the master, it will include within its request a list of any newly discovered URLs it parsed as well as the number of documents it has crawled since its last request, which updates the master bookkeeping values. The master responds with a list of new URLs for the worker to crawl, and the *Master Communicator* will place all the newly received URLs in the *URLs to Crawl Queue*. The worker determines how many new URLs it wishes to receive based on parameters set for the size and empty threshold of the *URLs to Crawl Queue*, and includes the number within its request. The master will remove that amount from the *URL Ready Queue* to send to the worker.

Within each worker, a thread pool of *Document Fetcher*s pop URLs from the *URLs to Crawl Queue*. For each URL, it first makes a HTTP HEAD request to fil-

ter out irrelevant documents, makes a GET request to get the document contents, and the places the URL along with its document contents and information into the *Unmarked Content Queue*. *Batch Content Seen Check* threads retrieve documents from the *Unmarked Content Queue* and marks each document based on if a document with a hash value of its content matching the current document is already stored in the database (i.e. the document content's has already been crawled). It then places the document in the *Seen Content Queue*, *Unseen Content Hashes Queue*, *Unseen Content Queue*, or *HTML Content Queue* depending on 1) if the document has already been seen before and on 2) the document's content-type.

Batch Update Content Info threads pop documents from the *Seen Content Queue* and update the list of URLs pointing to the document as well as the last crawl time for the document in the database. *Batch Add Content Seen* threads add the document hashes from the documents stored in *Unseen Content Hashes Queue* to the database. *Batch Store Content* threads pop documents from the *Unseen Content Queue*, zips them up in batches to store in S3 storage, and then stores the content information about each of the documents in the database. Document Parser threads pop HTML documents from the *HTML Content Queue* and parse them for outgoing links, which are then trimmed (all query parameters are removed) and sent back to the master in the next *Master Communicator* request.

Newly parsed URLs sent to the master's *Tasks Communication Route* are placed into the *New URLs Queue*. Each URL is then filtered by *URL Filter* thread pool and placed in the *URLs to Put in RDS Queue*. Filters for URLs include ensuring that it does not exceed a maximum length, does not end in a non-US country code, does not contain any profanity terms, and does not belong to a list of blacklisted domains (this consists of streaming, E-commerce, social media, and pornographic sites). Finally, the URLs are placed into the external central queue to be crawled.

3.2 Round Robin with Reset

We implemented a round robin queue to select URLs to crawl to avoid the crawler only crawling a limited number of domains. The queue is maintained as a table with auto-incrementing index rows in RDS and a list of tuples in master program memory. URLs are stored in RDS as entries of (index, URL, domain), and entries are never deleted. There's one tuple entry for each visited domain, and each tuple contains the domain along with the index of the last visited URL for the domain in RDS. Each time URLs are removed from the queue for a given domain, we query for entries with the provided domain and index greater than the index of the last visited URL for the domain. Results are ordered by index and limited to 300. The tuple in program memory for the corresponding domain is then updated to reflect the popped URLs. An important assumption of this approach is that most domains in queue will have many uncrawled URLs at most times; otherwise, some additional mechanism to keep track of non-empty domains would be necessary.

To find a balance between crawling the web breadth versus website depths, we implemented our round robin algorithm with a position reset of 5,000. Specifically, after reaching positions of dynamically increasing multiples of 5,000 the round robin position is reset to 0. That is, the round robin position initially increases until 5,000 and resets to 0, it then increases until 10,000 and resets, the next time around it resets at 15,00, and then at 20,000, and so on.

The intuition behind our implementation is that "good" websites will be discovered early and therefore be in relatively earlier positions in the queue. We can thus prioritize "good" websites by crawling them more frequently while also still ensuring that we crawl other more niche websites. For example, we did not include Wikipedia as one of our seed URLs but it was still discovered relatively early and placed towards the beginning of the queue. And by being in an early queue position, we managed to crawl a sizable number of Wikipedia pages relative to other sites.

3.3 Crawler Properties

Key properties of our system include the following:

Scalability: The master-workers architecture allows for workers to be added, removed, or modified as the crawler runs.

Batched Communication Design: Given the amount of documents crawled by each machine, minimizing network overhead was crucial. To that end, all database queries and communication between the master and workers are batched. This required a pipeline architecture, where at each stage one thread pool reads from one queue, performs its required tasks on the document or URL, and then moves it to a synchronized queue for the next pipeline stage, as seen in Figure 1.

Fault Tolerance: Our system robustly handles events of server failures. This includes both the case of a workers failing as well as the master server failing. The master maintains a list of all active worker servers, which is updated each time a request from a new worker comes in or no ping is received from a worker after a specified time period. The list of workers is stored within the *Run Context*, which is included in each master response to all workers. In each response, the master also notifies each worker what their current "worker index" is.

If a worker sends three consecutive failed requests to the master, the worker assumes that the master server is down. In that event, the worker having a worker index of 0 assumes the role of master. The selected worker spawns all master server thread pools on its machine and sends a request to all other workers informing them of the new

master address, and all workers will then send their requests to the new master.

4 Indexer

Our Indexer is broken up into two parts, the populator and querier. The role of each of these parts of the engine is described below.

4.1 Populator

The populator had the role of translating the various data collected by the crawler.

This was built using Apache Storm and consisted of 1 spout and 3 bolts. The description of each of theses bolts/spout are described below.

1. *ZipContents:*
   Fetches information about zip files stored in S3 by querying MongoDB

2. *ZipDownloader:*
   Downloads zip file by receiving S3 URI from spout and expands the zip locally

3. *ContentInfoFetcher:*
   Fetches information about the content expanded from zip

4. *DocumentParser:*
   Parses document stored locally using JSoup and updates MongoDB

Because the InvertedIndex was sharded into 3000 separate tables, the reason to which the following section would cover, the index populator internally maintained 3000 list, one corresponding to each shard. Whenever a document was parsed, the populator would append the entries to the corresponding list. To ensure batch requests would be made to MongoDB, whenever any of the 3000 lists exceeded a threshold of 2000 updates, the bolt would send a batch update request to the database.

To reduce token variety, a preprocessing step was executed as follows. All non-characters were removed and all characters were under-cased. To avoid indexing stop words, we used NLTK's list of English stop words.

The populator would compute the TF for each token in the document, as well as a weighted TF based on the number of time a token appears in certain tags in the document.

TF was computed using the following metric

$$0.5 + 0.5 * \frac{\text{Token Frequency}}{\text{Max Token Frequency}}$$

Weighted-TF was computed by accumulating a weight dependent on the tags to which the different tokens appeared in. Each time a token appeared in a certain tag, a predetermined amount shown in the table would be incremented for the particular token.

Weighted TF increment values for HTML Tags

Tag	Weight
title	0.30
h1	0.10
h2	0.08
h3	0.066
h4	0.04
h5	0.033
others	0.01

Once the document was parsed, the weighted-TF would be computed by multiplying each token's accumulated sum by the total number of words that appeared in the document. Intuitively, the weighted-TF would store how much meaning each token meant for a certain document and multiplying it by the total number of words would allow an even comparison between documents of differing lengths.

4.2 Querier

The querier is a Spark Java server that uses the sharded inverted index tables in mongoDB to return results to incoming search queries from the search engine client. The (unsharded) inverted index itself is a map from token to the document information objects - document hashes that contain the word and the positions of the word for that document. In the final implementation, we created 3000 (three thousand) shards of the inverted index, sharded by a) the token's hash and then b) the content's hash. More specifically, the sharding function used was: $shardnum$ = (|token.hashcode()| mod 30) $\times$ 100 + (contentHash mod 100). After thoughtful experimentation, we realized that greater than 95 percent of the query time is taken up by querying MongoDB, our bottleneck, so we aimed to have the shards in local storage to reduce network delays. We experimented with BerkeleyDB but found the most optimal form to be JSON files, which would be loaded in as necessary. The optimal strategy would have been to store this in memory, so as to decrease time spent in lazily reading in, but the inverted index was too large to fit in memory. Hence, we first maintained all the shards in the master node's disk (in case any worker crashed) and then used worker nodes that kept this in memory for faster access, and would send them over to the master as needed.

Document Weight Calculation - Parallelizability: Each token in a search query is divided into 100 (possibly overlapping with another token) shards by content hash and there is a one-to-one correspondence between the token shards by contentHash. For example, if in the query 'macbook pro', macbook and pro corresponds to shards [0-99] and pro corresponds to shards [300-399] (based on the first two digits in the shard number), we know that doc-

ument hash occurrences in shards 0 and 300, 1 and 301, ... respectively will be the same.

This allows us to create 'DocumentVectors' corresponding to each document in consideration of the search query. DocumentVectors are a list of position sets for where the tokens occur in the document, which can then be weighted by the search weight calculating algorithm.

Given the above system, we create 100 threads per query to create 'DocumentVectors'. These threads load in the necessary shards and then put 'DocumentVectors' into a queue, while a controller like collector keeps track of the number of such vectors emitted. A thread pool of weight calulators take in these vectors and then emit a tuple of (contentHash, weight), which the collector puts in a max heap sorted by weights. Once all emitted document weights are received by the collector, the server returns the results to the search client.

Multi-word Search Hits: Apart from TF/IDF and PageRank, we also consider a weight for documents with regard to incoming search multi-word queries. The idea behind this was to ensure that for multi word search queries, the weight of the document is also proportional to the length of the matched query string in the right sequence. While weighting by a simple Pearson correlation score is possible, we observed that it often weighs documents with more occurrences of substring higher than a document that matched the whole string, but a fewer number of times. Hence, in this algorithm, for each occurrence of token i in a document, say at position p we check if the $(i+x)^{th}$ token exists in the $p+x^{th}$ position, and so on. Since positions are stored in sets, this is done in $O(1)$ time, while also deleting the matched elements from the set, making sure that we process an an occurrence for any token only once. For a match length l the 'hit-weight' is calculated as e^{l^l}. Finally, the full weight of the document is calculated as:

$$weight(d) = 0.45 * \text{hit weight} + 0.20 * log(\text{PageRank}(d)) + \\ + 0.35 * \text{weighted(TF)}$$

5 PageRank

PageRank was implemented in two phases, a file generation phase and a computation phase. A file representing the structure of the graph was created by assigning a unique ID to each URL and representing the existence of an edges between two URLs by having lines of space-separated numbers.

Since we kept track of URLs based on the content hash, each content would have outgoing links as well as potentially multiple links that pointed to that content. As a result, when generating the file, we would have to take a cross product of both these sets to accurately capture all directed edges.

The computation phase was implemented using Apache Spark referencing the iterative algorithm laid out in the Google paper, with the number of iterations set to 15. We used AWS EMR platform and used Livy to submit Spark Jobs. Sinks were removed by adding back-edges, and the PageRank was initially set to 1.0 for all URLs.

5.1 Search Engine - User Interface

The search interface is a React.js client application with the following features:

- Wiki-Box for searches, implemented by matching the search term from popular Wikipedia pages, and then using a XPath parser to extract information from the page DOM.
- Voice search, implemented using the Mozilla SpeechRecognition API under the hoods
- Live multi-source news for searches, implemented using the News API
- Location-based current and forecast weather information, implemented using the OpenWeather API, with location on user permission

As described in section 4.2 and 6.7, it requests the querier, while fetching the first 50 search results with information to pgaintate them, and the next documentHashes as the user moves on to a new page.

6 Evaluation

6.1 Crawler

Initial testing with a master and single worker on the same machine showed that batched communication increased overall crawling time by about 10 times.

Measuring the crawler speed once completed varied dramatically over time. We continually changed both the number of workers and the ec2 instance types of our machines. Given our distributed and fault tolerance design, we were also able to change the actual code of our system and gradually phase our workers with redesigns. For example, at a point the URL Filtering step in the master became our system bottleneck, so we moved that within the workers without stopping our system in the processes.

We actively monitored our database usage and periodically changed our DynamoDB throughput limits. We also increased our MongoDB clusters multiple times throughout the run. We also had to periodically pause the crawler or slow it down (by killing workers) to avoid overwhelming mongoDB.

Additionally, we designed the master to automatically pause removing new URLs from the central queue if any of the internal queues exceeded specified limits and resume when levels receded. We similarly designed the workers to pause requesting new URLs from the master

when their internal queues were overfilled. These stop limits resulted in a robust, self-monitoring system, which caused the speed to fluctuate considerably.

Overall, we crawled over 6 million unique documents in about two and half days of uninterrupted time. At the peak, with 10 m4.xlarge workers and one m4.2xlarge master, we crawled 2 million pages in less than 6 hours.

6.2 Indexer - Indexing Speed vs Executors

The bottle neck of this procedure was effectively the DocumentParser bolt due to the high amount of computation required when parsing each document relative to the other bolts/spout. To evaluate the indexing speed, we stressed the number of executors for this bolt.

The number of documents parsed on one EC2 Instance of c5.24xlarge linearly increased from 1 to 2000 executors. Tested in increments of 200 executors, one executor parsed 100 documents per 30 seconds, while 2000 executors were able to parsing 100 documents per 8 seconds. After 2000 executors, however, increasing it up to 5000 executors seemed to have no increase in speed. This was likely due to the bottle neck moving away from the DocumentParser bolt and instead being the insertion/update queries times that could be made to MongoDB. Since at the peak, one machine could only index around 500,000 documents per 12-hours, we attempted to deploy the indexer on multiple EC2 instances. This seemed to increase the speed of indexing by the factor of number of instance we had running, which suggests that it wasn't MongoDB itself but rather Mongo's Java SDK that limited the insertion/update times for each instance.

6.3 PageRank - Nodes vs Speed

One iteration of PageRank was computed using two seperate methods, locally on an EC2 instance and another on AWS' EMR platform to compare the speed. Since we crawled over 6 million documents, the text file containing a line and two number for each edge was around 20 Gb.

PageRank was locally run on an c5.24xlarge EC2 instance and took over 9 hours to computer one iteration on. On an EMR cluster with 1 master (c4.8xlarge), 2 cores (c4.4xlarge), and 2 tasks (c4.2xlarge), the time it took to compute one iteration would decrease to be under 2 hours. However, the EMR cluster would continuously throw errors due to the lack of heap-space once the PageRank values were attempted to be returned after aggregating.

Due to the errors on EMR, we ultimately decided to run it locally and shard the URLs into three subsets by hashing and modding the URL string. We would then create three text files, only considering the edges that were between two URLs belonging to the same subset. This would dramatically reduce the size of the text files, dropping to slightly above 1 Gb for each, and allowed us to run PageRank locally for 15 iterations in under 6 hours for each shard.

6.4 Search Querier - Sharding

The query time dropped by about 8 times for a two-word query embedded in 250,000 documents from an average of 1 minute and 20 seconds when we had a single InvertedIndex table down to a bit over 10 seconds with 3000 tables for the same count of documents. For a single-word query, it dropped by 6 times, from a minute to about 10 seconds. This queried for about 75,000 documents.

6.5 Search Querier - Query Speed by Storage Methods

See Section 4.2 for storage methods. To mitigate the network delay from mongoDB, using JSON files in local storage and having to parse them proved to be quicker, with time equivalent to a 0.1 to 0.4 of the network delay, depending on the number of document from as small as 1000 documents to 340,000 documents respectively.

6.6 Search Querier - Query Speed in Concurrency

Having multiple concurrent queries does not affect the speed because we have a threadpool in which we have a thread for every single query. However, we realize that having more than 5 simultaneous queries can cause a Java out-of-heap-space exception.

6.7 Search Querier, Client - Server + Client-Side Pagination

We noticed that sending all search results to the client caused the web page to load very slowly on the client side, with browser FPS (using Google Chrome on a 6 core Intel Core i7) dropping to < 7, for 83k search results, while loading in 149 seconds. We changed our implementation to have only the first 50 results sent to the client, with the next content hashes and paginated, to load the first page of results in under 20 seconds.

7 Findings and Conclusions

Scale was our greatest challenge and lesson. Having to process and query over 6 million crawled documents overwhelmed many of our systems with regards to processing rates and storage, which required us to redesign many parts at the last moment.

8 Extra Credit Implementation

We implemented the following extra credit features we deem notable, which were discussed throughout this report.

Crawler: See the *Round Robin with Reset* and *Crawler Properties* sections (3.2 and 3.3) above.

Search Engine Client: See the *Search Engine - User Interface* section (5.1) above.

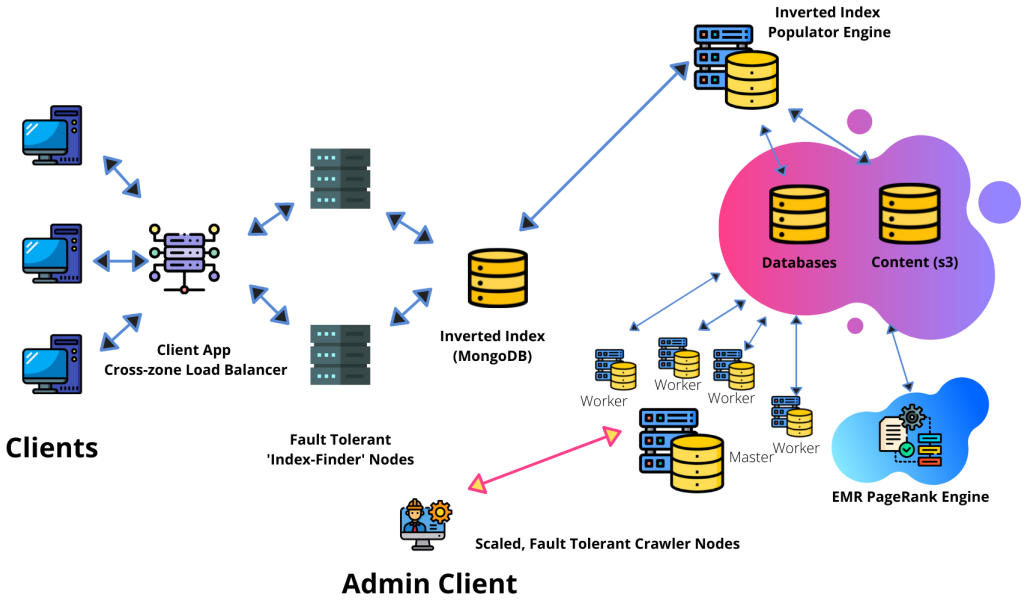

Figure 1: Project architecture, Credits: Varun

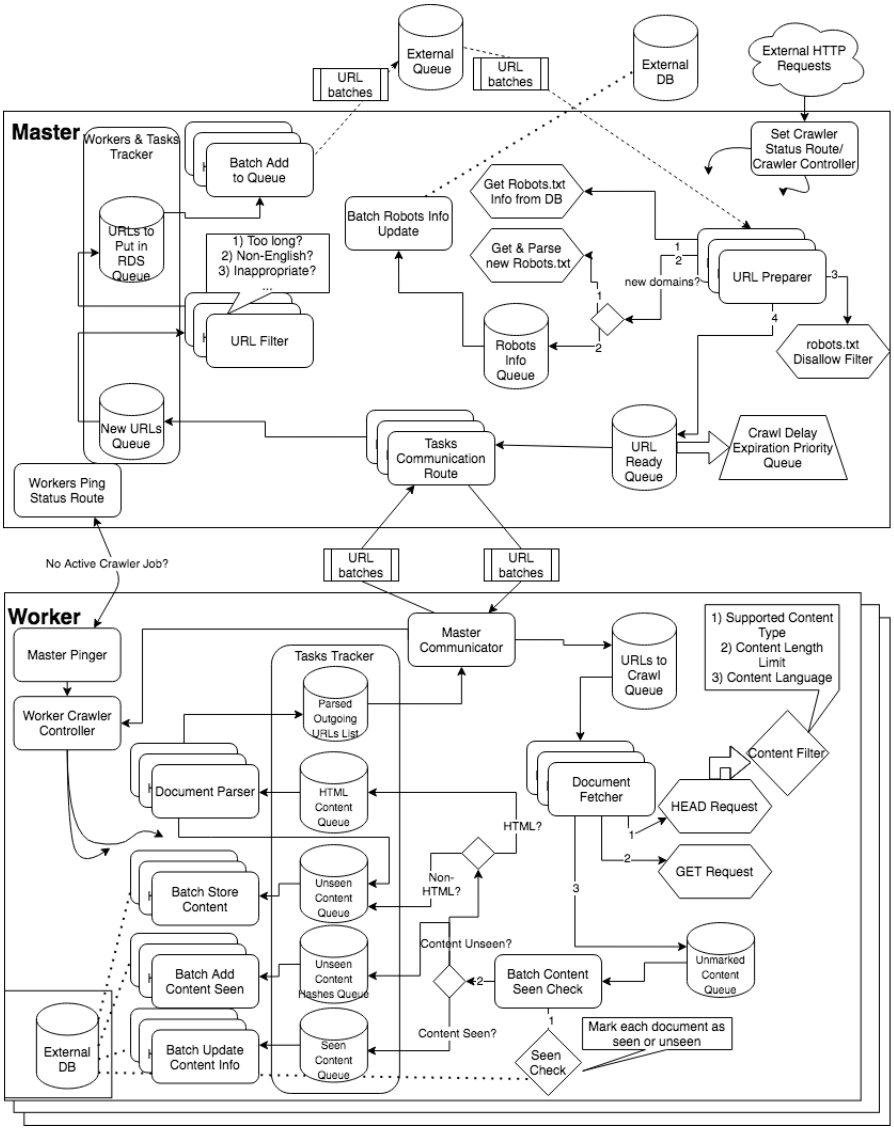

Figure 2: Crawler architecture, Credits: Chaim