CIS 555

Search Engine

Yagil Burowski, Chris Champagne, Kieraj Mumick, Igor Pogorelskiy, Augustus Wynn
Private Github: https://github.com/igorpo/SearchEngine

Introduction: What is SearchB0iz3000?

SearchB0iz3000 is a search engine based off of the Mercator-style distributed crawler, Hadoop-based indexer, MapReduce based PageRank engine, and, of course, a query engine & a frontend with DynamoDB and S3 serving as our object stores on the backend. Yagil and Igor worked on the crawler, Gus and Chris worked on the Indexer, Kieraj worked on the UI/frontend and PageRank Engine. We all worked on the query engine and integration together.

Milestones

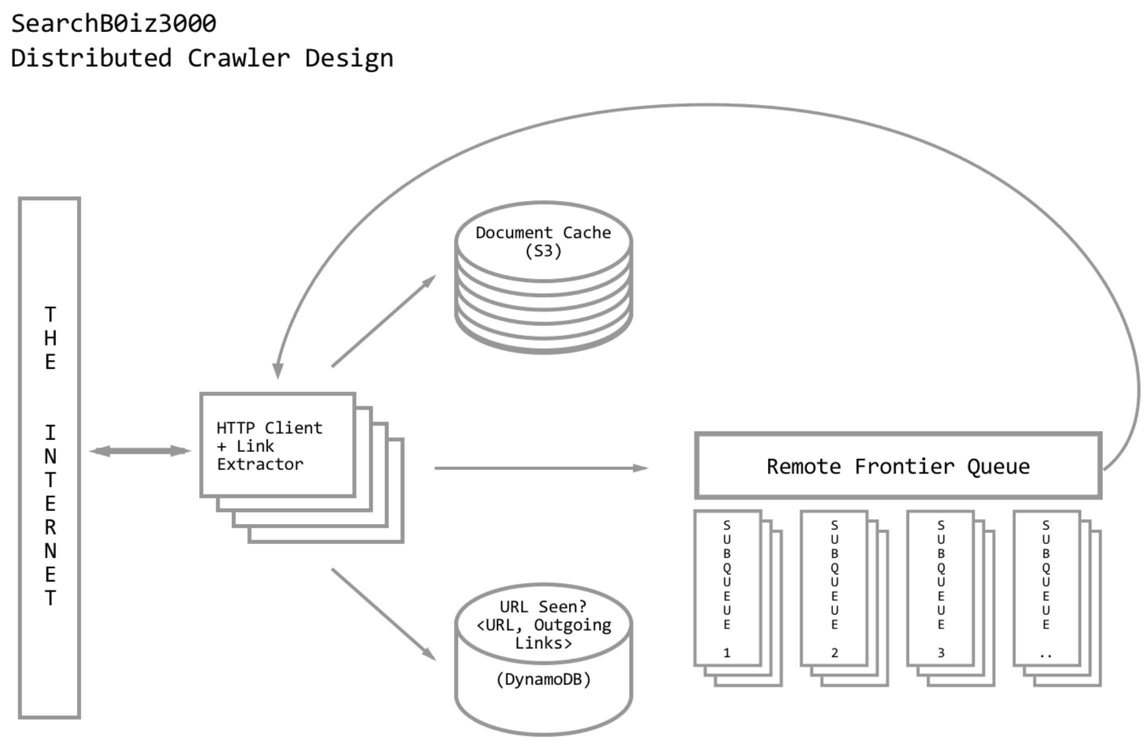
- November 27th: Formalize interfaces for Crawler, Indexer, and PageRank
- November 29th: Setup shared S3 bucket and DynamoDB table, as well as wrapper classes
- December 2nd: Crawler should be set up locally with Mercator style crawling
- December 2nd: Indexer should be integrated with map-reduce job/AWS
- December 8th: Finished Crawler, Indexer, and Pagerank
- December 10th: Create UI, any EC features, and figure out integration
- December 12th: Finished integration

Project Goals

- Understand and implement a complex, distributed system
- Understand challenges that arise with scalability and computing distribution
- Understand EMR and Hadoop

Architecture

Mercator-Style Crawler



1. **HTTP Client**
   The `HTTP Client` is used to interface with the remote systems that are in effect during our crawl which includes fetching/sending to the frontier queue system (routed to subqueues) and sending requests to fetch URLs from the web. The client consists of a transmitter and a receiver. The transmitter collects the information necessary to send the request, appends the proper "cis455crawler" headers and forwards the request (for both HTTP and HTTPS, the latter being wrapped in a Java HttpsUrlConnection) with proper request headers to the designated host server. The receiver then parses the response, fills a header map and gives access to the properties via the client.
2. **Link Extractor**
   Right now: implemented simply in series to the `HTTP Client`. After a client had downloaded a document, it stores it in the `Document Cache` (See 6.) and passes it as an argument to the `Link Extractor`, which in turn parses the HTML document and send a collection of extracted links in batches to the `Remote Frontier Queue`.
3. **Remote Frontier Queue (RPC)**
   We've chosen to implement the `Remote Frontier Queue` as a daemon process, accessible via an HTTP interface (using Java Spark). Each of the `HTTP Client` worker is an RPC-like client in respect to the `Remote Frontier Queue` which acts as the server. The workers are not aware that the queue is remote, and communicate with it through a wrapper class (the RPC interface). The queue is implemented as a map of queues, which each of the `HTTP Client` worker's thread having its own designated subqueue, with its thread ID as the key.
4. **Queue Disk Files**

Given the high volume of URLs, the in-memory queues of the `Remote Frontier Queue` are likely to fill up fast. To handle this, we've chosen to implement the following scheme using a lightweight and simple system of .txt files:

a. We maintain a map <thread ID, is queue full> to denote whether we should enqueue URLs to the in-memory queue, or rather to disk.
b. When a subqueue approaches some capacity threshold C1, we start storing (in batches) in URLs to disk.
c. We continue to dequeue URLs (also in batches) from the in memory queue until some threshold C2.
d. At this point the system decides that it is safe to start enqueue to the in-memory queues once more.
e. When storing to disk, we keep track of the file size of each disk queue. When the file size approaches some threshold C3, we create additional queue files while keeping track of which one to read from and write to.

The reason we chose to implement this system using .txt files and not for example BerkeleyDB is because we defined our need as a simple requirement for read and write to disk, which we were able to implement easily.

5. **Content Seen, URL Visited, Robots Parser**

The crawler has several components that are necessary for politely and efficiently crawling the web. First of all, in order to keep the frontiers from overflowing with already seen links, we used the DynamoDB table for storing {keys → outgoing links} as a keyed cache to quickly check if a link had been visited or not. This of course cannot be done locally because in a distributed setting each node can visit different URLs and because of memory overflow issues. Furthermore, for the sake of politeness, we implemented a robots.txt parser that takes a robots.txt file and parses out the proper `Allow, Disallow, Crawl-Delay` per host. We keep a stored map of {url → host's robots} so we do not have to hit the same host over and over again for the same file if we have seen it before. We also keep a map of {host → last time requested} in order to efficiently determine whether we need to wait or not. Finally, we also have a module for content seen (determining if we are in a spider trap). In creating this module, we experimented with string kernels and cosine similarity, ultimately going for the cosine similarity approach. String kernels are interesting but extremely complex to implement and have many causes of inaccuracies/bugs. Cosine similarity with a threshold value of ~90% (of parsed web page html) gives a very good accuracy of determining whether or not 2 webpages were the same content or not.

6. **Document Cache**

We've implemented a wrapper class over Amazon S3 `S3Wrapper` which handled saving raw HTML documents in our S3 bucket using Amazon's Java SDK. Saving to S3 is done in batches. The `Document Cache` is shared with the Indexer and PageRank components of our overall system.

7. **Workload Distribution**

Our crawler is distributed as follows:

a. A single node is comprised of parts 1 - 4. Parts 5 and 6 are universal to the entire system.
b. Each node has a pool of *n* `HTTP Client+Link Extractor` threads. To crawl our corpus of 1M documents we set 60 worker threads per node.
c. Each node is seeded with an initial list of *n* URLs, one for each thread. For our implementation we created those lists manually, aiming to create a diverse collection of initial links (news, shopping, travel, government, universities, etc.)
d. We distribute one node per one EC2 instance.

The scale of our system is a function of the number of threads per node, and number of nodes.

8. **Miscellaneous**

Additionally, there are several scripts provided for convenience. JARing files and uploading to instance nodes, proper seeding of instances, and running the instances are all done via scripts.

9. **Evaluation of System**

Metric: # of documents processed / minute
Processed Document: A processed document constitutes a document which has been successfully parsed, stored (including metadata), all links extracted and sent to frontier, and it is ready for indexing/pageranking.
EC2 instance: Instance Type → t2.medium, vCPUs → 2, Memory → 15GB
S3 Bucket: US Standard
DynamoDB: 25 read/write capacity units

# of EC2 instances: 1			
1 worker thread	30 worker threads	60 worker threads	180 worker threads
135 docs/min processed	1312 docs/min processed	1455 docs/min processed	2389 docs/min processed

# of EC2 instances: 2			
1 worker thread	30 worker threads	60 worker threads	180 worker threads
257 docs/min processed	2465 docs/min processed	2577 docs/min processed	4025 docs/min processed

# of EC2 instances: 3			
1 worker thread	30 worker threads	60 worker threads	180 threads
401 docs/min processed	3752 docs/min processed	4117 docs/min processed	6665 docs/min processed

Indexer+tfidf

1. **Input from S3**
   The Indexing and tfidf job is implemented as a MapReduce Job running on a Hadoop Cluster. The input to the map job is the files as values, along with the base32 encoded matching normalized-url. To do this on EMR, we can simply use the bucket filled with the results of the crawler as the input, though we have to be careful to make sure that Hadopp is configured to "split" the directory so it keeps the files complete so the parsing in the map step makes sense.
2. **Map Job**
   The map job does three things: parse the document, produce tf values for each word in the document, and output each word as a key and the tf and url as the value. Parsing is done with JSoup, as an embedded jar in our EMR job. Jsoup conveniently provides a way of extracting text from a document. We then split the text by word boundary, stem, and keep track of the number of times the word occurs. We use the max occurring word to calculate tf, the move on to collect the outputs.
3. **Reduce Job**

The reduce job counts the number of urls associated with each word (i.e. the key) and uses it to calculate idf along with the number of docs. Then reduce creates a list that is the list of (url, tfidf) pairs that are associated with each word.

4. **Output to DynamoDB**
   The reduce step simply puts this list with the word as the primary key into our index.
5. **Evaluation of System**
   Metric: # of words indexed / minute
   Indexed Word: An indexed word constitutes a word which has all of its instances found each document in the corpus, the list of all urls and associated tf-idfs paired together, and each word with the list of url, tf-idf pairs stored and ready for querying..
   EC2 instance: Instance Type 2 → m4.xlarge, vCPUs → 4, Memory → 16GB
   S3 Bucket: US Standard
   DynamoDB: Various write capacity units

   Extrapolated, to the best of our ability, data from one big one run with lots of nodes
   Assuming 500 indexable words per page

# of EC2 instances			
1 Core	5 Cores	10 Cores	15 Cores
.24 docs/min processed	1.2 docs/min processed	2.3 docs/min processed	3.5 docs/min processed
100 words/min indexed	600 words/min indexed	1150 words/min indexed	1750 words/min indexed

PageRank Engine

1. **First PageRank Map**
   The first PageRank Map reads in the documents in the same way as the indexing job, but ignores the document content. It reads in a list of urls that this document links to, precalculated in the crawler. It outputs (url, link) pairs for each link found.
2. **First PageRank Reduce**
   This reduce outputs a list-as-a-string for each url, where the list contains the url and the value 1.0 tab separated from a comma separated list of urls this url links to.
3. **Second PageRank Map**
   This job takes in the output of the first page rank job, parses the page, the pagerank value and its outgoing urls. The map outputs (outgoingLink, [url, urlRank, numberOfLinksOut] for each link leaving the from the url. It also outputs (url, links) and (url, isCrawled) which contain the links from the url and whether the url was crawled.
4. **Second PageRank Reduce**
   The reduce collects the data from the map so it can compute the pagerank of each url that was crawled. If a url was not crawled it is discarded from the output. Otherwise, it outputs the url, the pagerank, and the comma separated list of urls this url links to. The output of the reduce gets input back to the second map to converge the page ranks of each url.
5. **Third PageRank Map**
   The third map job takes the output of the second reduce and filters out unnecessary information. It outputs (PageRank, page) pairs to the reduce.
6. **Third PageRank Reduce**
   The third reduce job outputs (page, PageRank) to S3.
7. **Transfer of PageRank Data**

After running the third MapReduce job, the urls and corresponding page ranks are transferred from S3 (where the data is stored) and transferred to DynamoDB by using Hive.

8. **Evaluation of System**

Metric: # of pages ranked per minute
EC2 instance: Instance Type 1 → m3.xlarge, vCPUs → 4, Memory → 15GB
Instance Type 2 → m4.xlarge, vCPUs → 4, Memory → 16GB
S3 Bucket: US Standard

Extrapolated, to the best of our ability, data from one run on 5 cores

# of EC2 instances			
1 Core	5 Cores	10 Cores	15 Cores
11.0k urls/min processed	55.1k urls/min processed	110k urls/min processed	160k urls/min processed

Database Structure & Conclusion

DynamoDB & S3

Content	**Documents**	**Parsed Links**	**Index+tfidf**	**PageRank**
Key	**Url (base32) (String) S3**	**Url (String) DDB**	**Word (stemmed String) DDB**	**Url (String) DDB**
val	Document contents+metadata	List<url)	List<List<String>> (the internal list is a pair:(url, tfidf))	PageRank (int)
`I`	`S3Wrapper`	`DynamoWrapper`	`DynamoWrapper`	`DynamoWrapper`

Conclusion

After implementing each component detailed above and integrating it with an intuitive front-end, we successfully built SearchB0iz3000, a thorough and lightening fast search engine. We tested the engine and its components on a variety of types and numbers of EC2 instances including: m3.xlarge, m4.xlarge, m3.2xlarge, and m4.2xlarge. We also experimented with several tools and techniques to use for implementing and running the components including: DynamoDB, S3, EMR, Hive, Hadoop, and more. By trying so many tools, we all learned important lessons and trends about web systems and scaling and have been able to identify the best tools to use for the best speed and cost efficiency for varying data sets.