Team G22 Final Project Report 5.8.2016 clancyt, xishanni, adducci, gyixuan

Introduction

Our group, owing to a series of scheduling conflicts, recognized that time to test and integrate code together would be limited. We therefore took the approach of agreeing to use constantly-accessible S3/DynamoDB for storage and defining early on the table structures. This required planning how data would actually be processed, stored, and retrieved, which led to the realization that another previously-unconsidered component of this project, the S3 to DynamoDB data inserter, was needed. A part of this procedure was defining the search process as a series of steps: crawling data, processing data, transforming data, and using data. This was achieved by defining exactly what information was needed as input at each step, and manufacturing some test data of this format to use while the component of the prior step was being constructed. As implemented, the crawler took no input and produced a store of documents, the indexer took a store of documents and produced S3 records, the PageRanker ran in parallel and produced other S3 records, the inserter ran and wrote all output S3 records into DynamoDB tables, and the user interface drew data from these tables to construct search results.

The division of labor remained close to that outlined in the original project plan: Rachel was responsible for the distributed crawler, Tim was responsible for the PageRanker, Shanni was responsible for the user interface with pre-demo assistance from Tim, and Frank was responsible for the indexer with assistance from Tim to accelerate progress. To make this reliance on individual programming successful, everyone tried sticking to a timeline roughly similar to our initial timeline. The initial timeline was, however, too ambitious. Most portions of the project were completed in line with the original timeline shifted back by a week, to compensate for extra time needed to define the job flow for ultimately producing search results and the interfaces between steps. The indexer and crawler however took significantly longer than initially planned to complete. The group conducted bi-weekly check-ins between periods where most work could be scheduled:

By 4/13: interfaces defined, storage utilities to abstract work with Amazon defined, and overall architecture simplified.

By 4/17: PageRank implemented on HW3 framework. Crawler/S3 storage implemented.

By 4/20: Crawler retrieves sample data. Indexer reporting trouble with MapReduce, pointed to tutorials. User interface begins testing with sample data. PageRank verified working on sample data.

By 4/24: Debugging PageRank/HW3 framework on EC2.

By 4/27: Integration meeting. Realization that Indexer had not been started. PageRank re-implemented and tested on EMR to make up lost time. Crawler distributed to EC2 but reporting trouble. User interface progresses.

By 5/1: Indexer complete. PageRank complete. Crawler makes progress. User interface ready for testing. Work begins on Inserter.

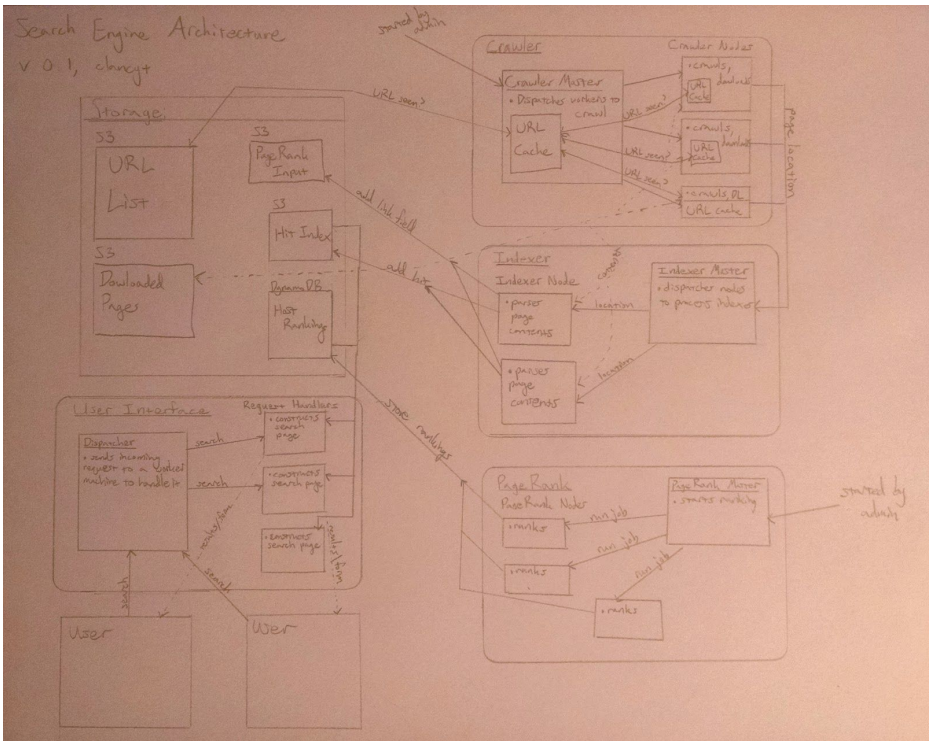
By 5/5: In crunch time, Crawler distribution was completed, Inserter tested and completed, small sample of data was processed, and User interface was completed. Several days were spent patching a wide array of integration bugs in preparing final project for demo.

Ultimately, while this scheduling left much to be desired, the end result was satisfactory. Lessons drawn from this scheduling are discussed below.

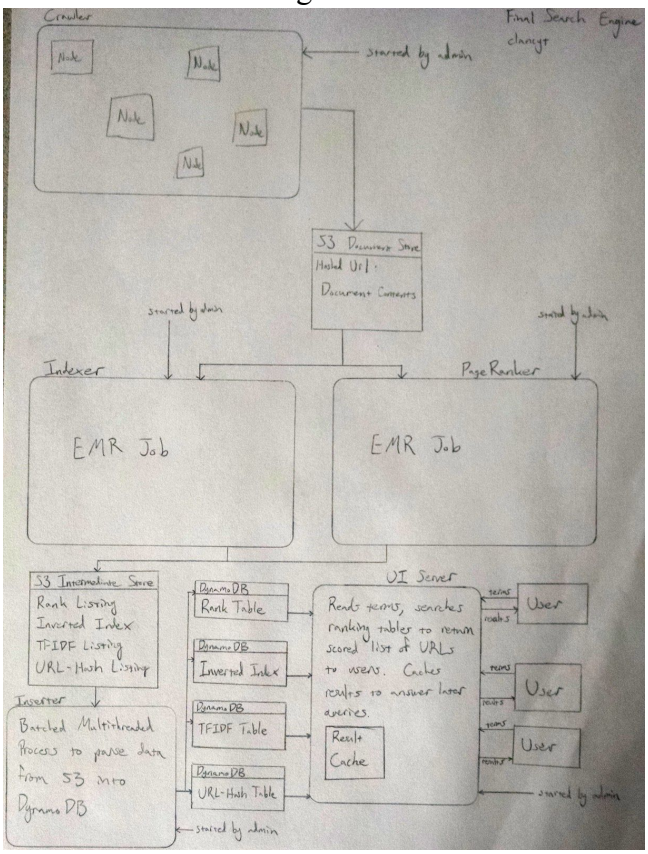
Simplifying Architecture

Crawler: The Crawler runs based on the HW3 MapReduce-esque framework of distributed nodes. The nodes share a single URL frontier and are periodically assigned new jobs from a master server. All results are downloaded into a collective S3 storage.

The project plan originally called for this architecture design.



More discussion and review of homework assignment code resulted in a simpler design.



The final design contains five components which are all manually started and have no cross-component communication. All data is read into/out of storage (either DynamoDB or S3) before use in another component.

Indexer: After the crawler downloads pages, the search engine needs the Indexer to process the content and build an index. Upon user search query, the search engine will use this index to look up relevant pages and return accurate search results in a short time. An inverted index is a mapping from a word in a document to a list of documents. In all cases of the Indexer, a word is normalized by removing all non-letter characters and

stemmed using the Porter stemmer before being worked with. For each word-document pair the Indexer must calculate a weight based on the TF-IDF model. Weight is high when the document can be highly represented by the word. Later, upon user search query, the search keywords taken as input are used in the inverted index to retrieve documents and rank them based on weight to determine relevance. For now, output is simply an S3 bucket.

The TF-IDF model suggests that the weight for each word-document pair is determined by two key factors:

1. Term Frequency (TF): The frequency of the word in the document compared to other words. This factor tells us how important the word is to a given document. The higher the frequency, the higher the importance. The TF formula is: TF(w, d) = a + (1 - a) * (freq(w) / max_freq(d)). In this formula, a is a constant usually set to 0.4 or 0.5. In the demo, good results followed when set to 0.5. The arguments of TF function w is a word and d is a document, freq(w) is the number of times w appears in d, and max_freq(d) is the highest number of times that any one word appears in the document.
2. Inverse Document Frequency (IDF): The frequency of a word's appearance across all documents. This factor tells us how unique a word is, and thus how meaningful its per-document frequencies are. The IDF formula is: IDF(w) = log (N / n). In this formula, w is a word, N is the total number of documents downloaded, and n is the number of documents that contain w. For each word there will be only one IDF, since it describes a corpus-wide aspect of a word.

To calculate the weight, the formula is: weight(w, d) = TF(w, d) * IDF(w). The Indexer is implemented using the MapReduce framework provided by Amazon's EMR. An admin starts the Indexer process, upon which each mapper reads a document in one key-value pair. The key is the hash of the document URL, and the value is the raw document content. The mapper parses the html-formatted document into a pure text document by taking out all the html tags and links. Then it reads through the document and counts the frequency of each word in a map. Each mapper also calculates a term frequency for each word, comitting in the format of (word, "url, tf"). The reducer receives many commits for each word. The reducer calculates the IDF for each word based on the number of commits it receives and the total number of documents downloaded. Then the weight of the word with each document that contains it is calculated and pushed to an S3 output file.

PageRanker: While the Indexer is run to process the crawled data, the PageRanker can simultaneously run in EMR on a separate cluster. The PageRanker begins by parsing through the entire corpus of documents and extracting the links from every page. With this, it produces a file consisting of two hashed URLs per line, with each line representing a link between one document and the next. This then becomes input for a series of six different jobs which form the algorithm. The jobs are initialization of data, counting documents, iterating on data, redistributing lost rank, checking differences, and finalizing output.

The initialization stage takes the list of hashed URL pairs and creates a mapping which condenses all of the pairs into one row per document, also listing rank and outgoing links. This condensed listing is then fed into the iteration job, which performs the actual PageRank algorithm in determining how much rank flows from each page across its outgoing links, and how much flows in from inbound links. Once this stage is completed, the redistribution job determines how much rank flowed out of nodes which do not have any outgoing edges. Normally the rank from these documents would be lost to the system. Instead, the addition of this job determines which portion of this rank is shared with every document, where it is added back in evenly to prevent loss of rank. The completion of this stage triggers another iteration phase, and the algorithm iterates until the ranks converge or a set number of iterations occurs. The periodic check for convergence (for the demonstration data this was set to every three iterations, which seemed like a reasonable choice for balancing good results with efficiency) simply compares all ranks between documents of the previous two iterations and finds the maximum difference. If this difference is within some threshold, iterating terminates and the job continues to the finishing stage. The finishing stage outputs a single list of pairs between hashed URLs and final ranks, which is used later in calculating result rankings.

Inserter: Upon completion of PageRank and the Indexer, the output data necessary for ranking results is stored in a series of S3 files. To facilitate faster access to these, it is necessary to put them into DynamoDB tables. This has the benefit of making it very easy to increase/decrease available throughput to the data if use rate of the search engine changes. The Inserter runs on an EC2 instance and pulls data from the S3 buckets where Indexer and PageRank output is stored. This output is parsed knowing the structure of the output files from these two processing jobs, and inserted into several DynamoDB tables. Fast insertion is supported by using multithreading, batch-writing of 25 records at a time, a machine with fast I/O connection, and very high write throughputs to the DynamoDB tables.

DynamoDB Tables: The Inverted Index, TF-IDF, PageRank and mappings between hashed URLs and actual URLs are stored as information in Amazon DynamoDB tables. These make up the interface for front end access to data. For the Inverted Index table, each item is stored with the term as the primary key and a value "hashList," which is a semicolon-delimited string of all documents containing reference to that term. Each entry in this string also tracks the position in the document where that term appears, to allow support for proximity checking. Another value "numOcur" records the total number of documents where the term appears. For the TF-IDF table, term again is the primary key. Each entry tracks a "tf" value of Indexer-calculated term frequency and an "idf" value—inverted document frequency. For the PageRank table, the hash value of the document's URL is primary key, and a "rank" value is the PageRank score of that document. Finally, the URL table uses the hash value of the document's URL as primary key, and tracks a "url" value of a document's actual URL, pulled from metadata in the S3 entry. Together, the data in these tables supports fast querying of information through the User Interface.

User Interface: This combines all data produced by previous components to provide a search engine interface for users to input queries and display results. The UI tunes the weights of TF-IDF and PageRank to score each retrieved document appropriately. For the demo, the TF-IDF and PageRank scores were simply multiplied to produce a final result. However, with a larger corpus of data, it would perhaps be wiser to weight PageRank more heavily to combat several trap pages we encountered. Some pages would simply contain enormous lists of irrelevant keywords to return as results for some of our queries. PageRank is an effective way to combat such irreputable sites.

The UI itself runs in a servlet (launched through Jetty) on EC2, and presents to the user a form where search terms can be entered. This allows it to support a much higher access speed to our DynamoDB tables. Search terms are retrieved, normalized, and stemmed. Then the corresponding list of possible documents is retrieved from the inverted index. For a multi-word query, a list of all documents relative to any term is retrieved. The retrieved documents are then ranked. For every term-document match, the TF-IDF data is gathered from the TF-IDF table. Then the query frequency is calculated, for weighting multi-word queries. After this, PageRank scores for each document are taken from the PageRank table. Finally, the TF-IDF score is multiplied with the PageRank score to calculate a final ranking score, and the list of documents is sorted to find the first page (default size is 20) results. The URL mapping table retrieves the real URLs of these pages to display as links to the user.

Ranking

Ranking is determined by equally weighing calculated TF-IDF scores and PageRank scores, although we did encounter several pages full of fake keyword lists or boosted by link farms, so further development should focus on improving rankings by combatting these underhanded SEO techniques. The components which produce these rankings describe how the calculations are performed. As a piece of extra credit, our user interface integrates Amazon search results by querying the Amazon item search API. These are displayed on the right hand side of the page, alongside the search results (note: we forgot to declare this extra credit in the submitted README, but it is implemented). No special ranking is provided to sorting the amazon search results.

Evaluation

Evaluation of the the performance from different components in the search engine, as well as the testing conducted during the demo, show that it is a viable engine quick enough to be useful. This is supported by the more empirical testing of several components.

Crawler:

The Crawler is highly-scalable, having been distributed along the lines of the HW3 MapReduce framework such that additional nodes can be added to the system to improve its speed. When testing the Crawler's performance on one node with varying level of threads we found that 50 threads resulted in the best performance. Each ten additional threads from ten to 50 increased the number of documents collected per minute. At sixty threads the performance started to decrease, and from then on continued to decrease until almost no documents were being collected. With ten threads the crawler collected around 32 documents per minute. With 20 threads the crawler collected around 98 documents per minute. With 30 threads the crawler collected around 146 documents per minute. With 40 threads the crawler collected around 182 documents per minute. With 50 threads the crawler collected around 217 documents per minute. With 60 threads the crawler collected around 152 documents per minute. This decrease in performance after 50 threads is due to thrashing and an expected problem when working with thread pools.

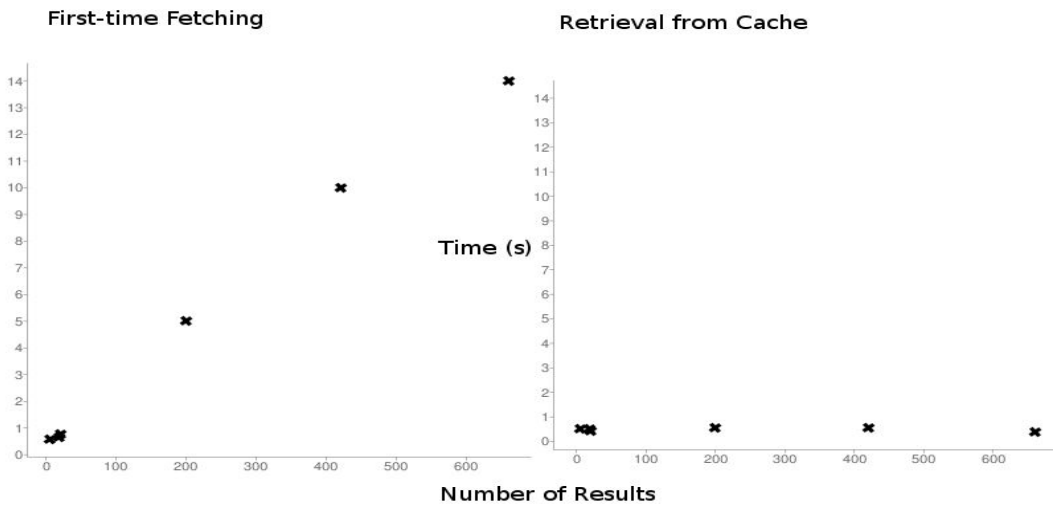
When testing the Crawler's performance on multiple nodes each running 50 threads, we got the following results. Adding more nodes increased the performance up to four nodes with each additional node from one to three increasing the performance less and less. With one node the crawler collected around 217 documents per minute. With two nodes the crawler collected around 279 documents per minute. With three nodes the crawler collected around 308 documents per minute. With four nodes the crawler collected around 290 documents per minute. We believe that choosing S3 as our database contributed to the bottlenecking issue because when so many threads on multiple nodes were issuing PUT/GET HTTP requests the database started erroring more. It could only handle so many HTTP requests at a time. We could have looked into batching requests to address this issue.

Indexer and PageRanker:

Similar to the Crawler, both Indexer and PageRanker run on a MapReduce structure—this time, EMR. Performance-wise, the PageRanker is about twice as fast for a set number of pages than the Indexer. The bottleneck here is perhaps the need to use JSoup to parse all text in the Indexer, while the PageRanker need only use JSoup to parse links. PageRanker can also use any existing URL mappings in DynamoDB to avoid parsing or interaction with S3 altogether, which would result in even greater performance gains.

User Interface:

The following graph shows UI performance through retrieval speed versus number of results.



Both tests were run with the same set of queries on the same DynamoDB tables provisioned for 1,000 reads per second. When a set of terms is searched for, all possible results are returned and ranked. Retrieval time is linear in the number of results. This is acceptable for smaller sets of results, but obviously terrible for words with many potential results. One solution to this might be the implementation of multiple threads/nodes retrieving results for one query. Once all results are compiled into memory processing is incredibly fast, it is just a matter of retrieving from DynamoDB. This solution could also be solved through effective implementation of pagination, so that not all results are fetched at once. An important part of the UI is the server-side cache of previously-retrieved terms. The second table shows that retrieval from cache is constant in the number of results. If another user has previously searched for requested terms, then returning the result pages is much faster. To ensure that the cache does not become overfilled, the least-recently-used entry is evicted if need be. The caching notion is interesting in that it allows us to "prime" the search engine with interesting results. One idea was to, if the server was idle, scrape Twitter for trending hashtags and prime the engine with the results.

The server itself, simply being a Java servlet talking to a DynamoDB table, could be very easily distributed simply by launching multiple instances and using a separate master servlet to fairly redirect any incoming requests to these instances. Periodic heartbeat checks would allow for the list of request-handling nodes to be updated should any fail. This would allow for much greater support for simultaneous searching.

Inserter: As described previously in the implementation of the Inserter, performance for this component is excellent once multithreaded, batched, and running on a high performance EC2 instance. Only one instance was ever needed to insert data for the demonstration corpus in this project, although it is easily parallelized by having multiple Inserters each working on one table, or by simply partitioning the data in the S3 buckets for multiple Inserters to use. This could be easily implemented as something automatically done by the Crawler/Indexer.

Lessons Learned

Overall, we consider the project a success. While reliant on Amazon for many components, the integration work done by the group is still something we are proud of. The end product was able to successfully complete queries and display ranks despite having only worked with a very small corpus. While every component was able to work at some basic level, a few worked much better than others. The Indexer, for instance, is still quite slow when compared to the speed at which we are able to retrieve new documents for processing. Additionally, the group recognizes some problems in storing every crawled document in a giant, directory-less S3 bucket. It makes searching for specific content much slower, which proved to be a bottleneck hindering the feasibility of display textual samples from each ranked result. A good fix for this would have been to define a well-defined folder structure which the PageRanker and Indexer could then parse around when needed. Searches would be faster for text from the raw documents. Samples could also be stored in another DynamoDB table for even faster results. Another concern is that of our data structures operating at scale: memory structures in all mappers and reducers require checks on their bounds to ensure they never grow so large that they begin to run out of memory and crash. This was not a problem at the scales tested for this project, although solving this vulnerability properly would require periodic emission and clearing of these structures, as is implemented in the server-side cache. If the group had to implement this project again, we would recognize much sooner the performance benefits that could be found by running projects directly on EC2. To avoid final crunch time in trying to integrate, it would have also benefited this group to establish much stricter deadlines with one another. This way we could have had more time to implement some extra credit and features that we simply had planned originally, such as automatically triggering Indexing or PageRanking jobs without constant administrator intervention.

Conclusion

This project was enjoyable and certainly served as an excellent way of bringing together everything learned in this class. In the end, the resulting search engine was functional, interesting, and a truly distributed system.