Bing Tea Final Report

Building a Search Engine from Scratch

Bing Tea Group Members

- Danny Kim | Computer Science LSA | Junior
- Jimmy Dai | Computer Science Engineering | Junior
- Kevin Li | Computer Science Engineering | Senior
- Matthew Wang | Aerospace Engineering + Computer Science Engineering | Junior
- Rickey Dong | Computer Science Engineering | Junior
- Roee Tsimhoni | Computer Science Engineering + Math | Junior
- Wonbin Jin | Computer Science Engineering | Junior

Group Workflow

The team communicated mainly via a Discord channel. The project was stored on a GitHub organization with every component of the search engine getting its own repository. We initially tackled all of the homework assignments by splitting them up and each person working on one or two assignments. We did this in order to finish the assignments as soon as possible and to get started on the actual components of the engine. After completing the homework assignments, we assigned key components of the actual search engine in the following teams:

- HTML Parser: Matthew
- Crawler: Wonbin, Jimmy, Roee
- Index: Wonbin, Rickey, Danny, Jimmy
- Constraint Solver: Matthew, Rickey
- Frontend: Wonbin, Rickey, Danny

Each group met to distribute work and to pair-program. Once a component of the engine was done, the different teams met to integrate the components together. Key design decisions such as the architecture of the crawler, the architecture of the index servers, whether or not to store parsed HTML files, and the two stage ranking process was made during team meetings with everyone present.

LOC Contribution

Component	Danny Kim	Jimmy Dai	Kevin Li	Matthew Wang	Rickey Dong	Roee Tsimhoni	Wonbin Jin	Total
HTML Parser	0	0	0	522	0	0	0	522
Crawler	0	75	0	0	0	162	2481	2718
Index	1213	300	0	0	1623	0	2378	5514
Constraint Solver	0	0	0	678	1772	0	0	2450
Ranker	0	1183	2860	0	0	0	0	4043
Frontend	312	0	0	0	312	0	804	1428
Miscellaneous	0	0	0	0	0	274	363	637
Total	1525	1558	2860	1200	3707	436	6026	16675

Supported Functionality Checklist

HTML Parser:

- ☑ Parse HTML documents to extract words and links
- ☑ Made special distinction to words that are in headings, bold, italicized

Crawler:

- ☑ Retrieves documents by HTTP and HTTPS
- ☑ Manages a prioritized frontier of URLs to explore
- ☑ Considers distance from the seed list
- ☑ Polite – obeys robots.txt and does not DDOS anyone
- ☑ Deals with redirects
- ☑ Parallelized by multithreading
- ☑ Able to deal with loops
- ☑ Restartable and deals with crashes
- ☑ Retires failed pages periodically
- ☑ Distributed across multiple machines
- ☐ Has good junk detection (porn, spam, etc.)

Index:

- ☑ Retains the source/original form of HTML documents
- ☑ Uses chunks and keeps statistics in each chunk
- ☑ Variable length encoding along with relative deltas for compression
- ☑ Uses seek table for fast seeking of ISRs
- ☑ Provides an ISR interface for the constraint solver
- ☑ Distinguishes words found in title and body
- ☐ Captures anchor text

Frontend:

- ☑ Provides a command line interface for searching
- ☑ Provides a web interface for users to type queries in a search bar and get results
- ☑ Provide clickable URL
- ☑ Provide title of a result
- ☑ Provide text snippets for a result
- ☐ Uses pagination

Query Parser/Constraint Solver:

- ☑ AND, OR, PHRASE, NOT, parenthesization

Ranker:

- ☑ Static page ranking
- ☑ PageRank or similar (CheiRank)
- ☑ Heuristic ranking based on location of query matches (title vs. body)
- ☑ Machine learning based ranker (XGBoost)
- ☐ tf/idf or bag-of-words

Computing Distribution Diagram (GCP)

The engine was distributed across 22 VMs on GCP and one laptop serving the frontend locally. The following is a diagram of how the components of the engines were distributed.

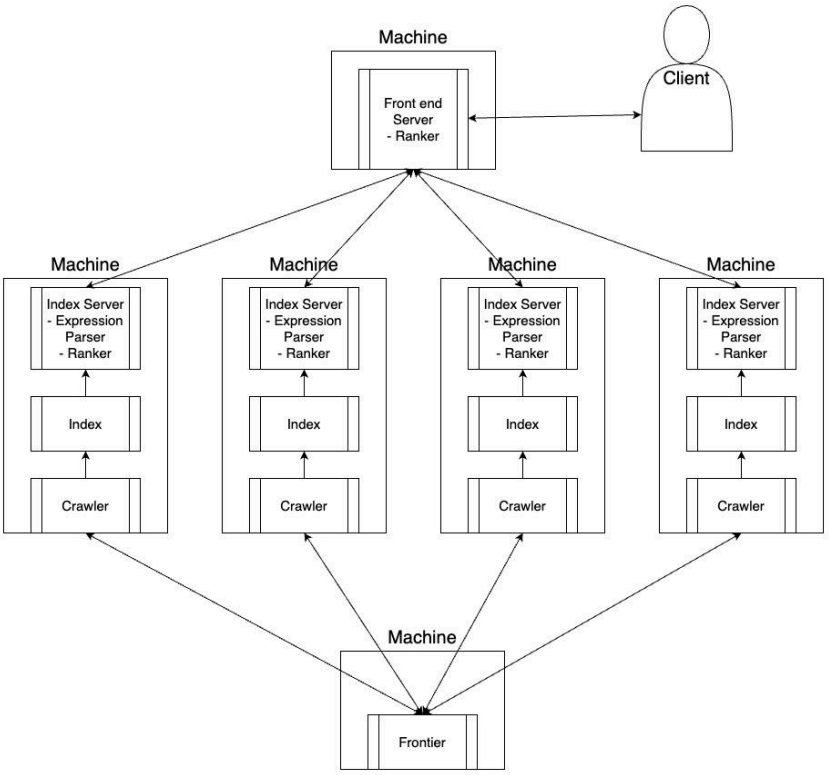


Figure 1: Distribution of computing within our search engine

Crawling: A central machine was in charge of maintaining the bloom filter and checkpointing to persistent memory. This frontier process distributed urls to 22 crawlers in batches of 50 and each crawler started one thread per url for downloading and parsing. The crawlers saved the parsed HTML files to their local disk storage for indexing at a later time.

Indexing: Once all crawling was finished, an indexing program was executed on each machine to take the parsed HTML files and index them into chunks. The indexer built one chunk at a time in memory and flushed them to disk once the chunk reached a 100 Mb threshold size.

Index Server: Once all indexing was finished, an index server was run on each of the machines. Each index server was responsible for receiving queries, parsing and evaluating the query using Index Stream Readers, and sending back heuristically ranked results back to the frontend server.

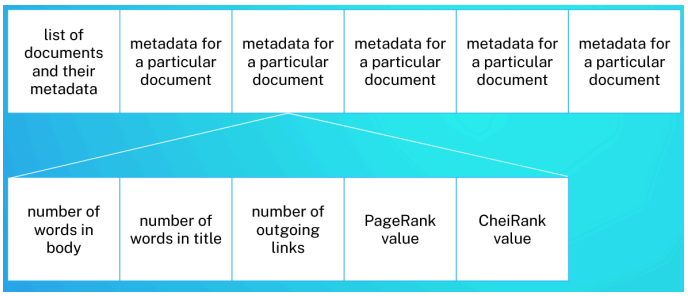
Frontend server: The frontend server was responsible for serving the landing page of the search engine and distributing the queries to all of the index servers. Once the index servers responded with results from each index, the frontend ranker ranked the results and served it to the user.

Filesystem Footprint

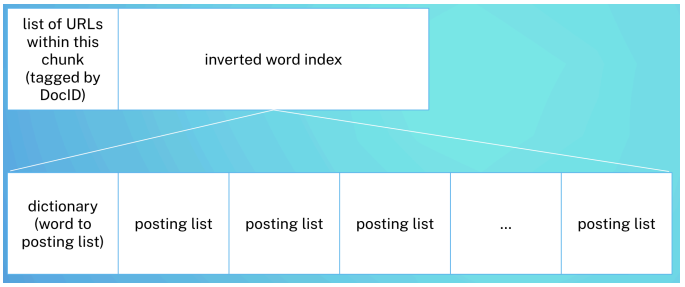
```
> tree
.
├── index
│   ├── 0
│   ├── 1
│   ├── 2
│   ├── 3
│   ├── 4
│   ├── 5
│   └── 6
├── masterchunk
└── metadata
    ├── 0
    ├── 1
    ├── 2
    ├── 3
    ├── 4
    ├── 5
    └── 6
```

Each index is split into an index directory, a metadata directory, and a masterchunk file. The index directory contains a file for each index chunk numbered from 0. The metadata directory also contains chunks numbered from 0. The index chunks and metadata chunks are built as pairs where each metadata chunk contains metadata on documents indexed in the corresponding index chunk.

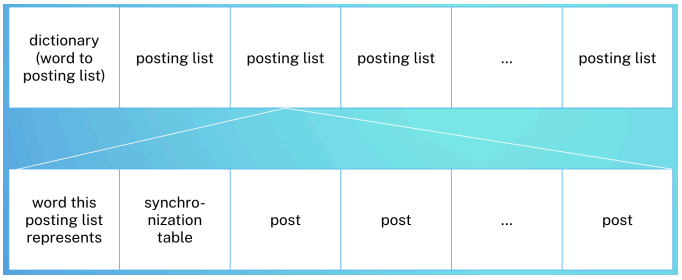
Within each metadata chunk, there is a list of documents, and for each document, its corresponding metadata.



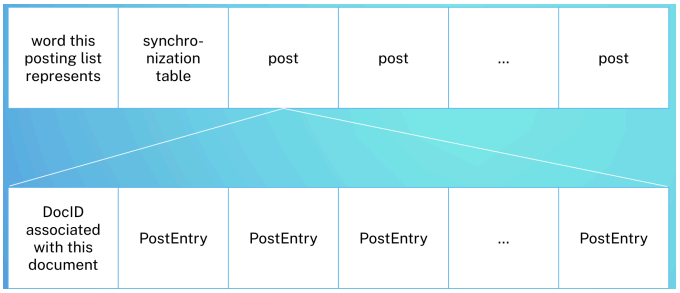
Within each index chunk, there is a map of urls to their associated document ID. This was followed by a map of words to Posting lists.



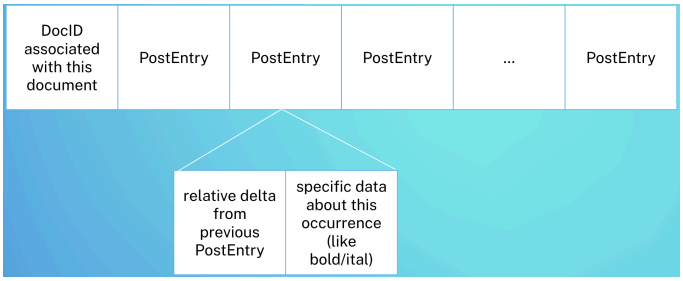
A posting list contains the word that the posting list is for, a synchronization table that is built when an index is deserialized, and a list of posts.



Each post contains a document ID that the post is for and a list of post entries.



Each post entry contains the relative delta from the previous post entry and the location of where the word was found (title, body, or emphasized)



Basic Statistics

Crawling statistics

Total Documents Crawled	Total Crawl Time	Average Documents per second	Peak Documents per second	Aveage Frontier Response Time
51,618,383	170 hours	84 documents / second	215 documents / second	2.2 ms

Index statistics

Number of tokens	Number of chunks	Number of documents	Size (GB)
31 billion	2881	50 billion	670

Filesystem statistics

Source material (GB)	Index Chunks (GB)	Metadata Chunks (GB)	Total Size (GB)
388.4	278	3.6	670

Architecture Overview

Building the Index

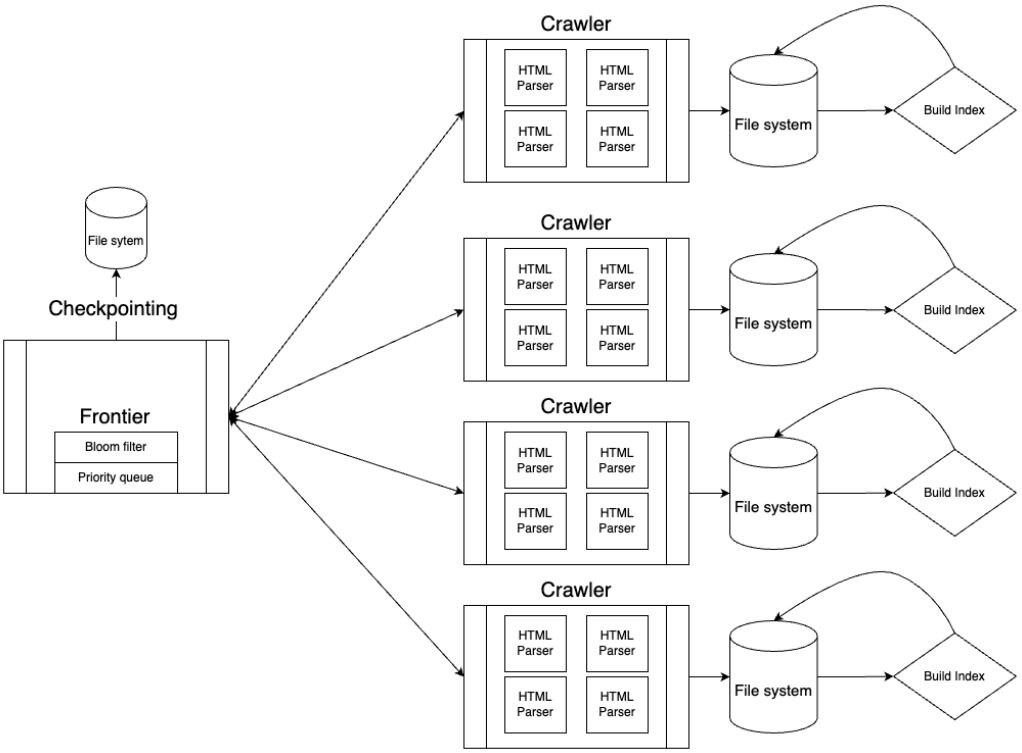


Figure 2: Logical flow of web crawling/index building

The frontier process listens to crawler processes by polling `select` and sequentially processing and responding to requests by adding new urls to the bloom filter and the priority queue and taking out urls for the crawler to crawl next. The frontier sends urls in batches of 50 that the crawlers fetch and parse using 50 threads. The seed list was generated from a collection of handpicked popular news sources, wikipedia, top 100 visited sites on google, and a mini crawler that was built in Python that selectively crawled and saved urls of specific domains like .edu and .gov. The crawler handled robots.txt by maintaining a separate bloom filter of urls not to crawl on the frontier process side. Whenever the frontier received a new url, it would check that it was not in the bloom filter for no-crawl urls before adding it to the priority queue.

When a given URL is crawled, it is passed along to the HTML Parser. The parser in turn parses the HTML, but it doesn't add it to the index right away. Instead, it creates an intermediary parsed file that still retains the source of the original HTML document. We initially made this design decision out of convenience and to ensure developer progress without needing to couple the two together, but in the end, these intermediate files turned out to be useful for snippets.

After the whole crawling and parsing process is done, we can initiate a second phase of actual index building. We have a wrapper class, "DocStream", that gathers these intermediary parsed files in order to use them. The actual driver program uses this DocStream class and feeds its output into a MasterChunk class. This MasterChunk class in the index then handles the

construction of the rest of the chunks, specifically the IndexChunk and each corresponding MetadataChunk. Each IndexChunk is sized to be about 100 MB.

Serving Queries

Constraint Solver

Our constraint solver is able to take an input query and convert it into an ISR tree that is used to iterate through posts/documents that satisfy the original query. When a query is received, it is parsed (lowercased, remove punctuation and junk words) and converted into a linked list with special tokens for operators. We then use Top Down Recursive Descent to parse the linked list into a recursive expression tree, which supports Nested, Phrase, And, Or, and Not expressions. The expression tree is then evaluated into an ISR tree, with each type of expression constructing a different type of ISR.

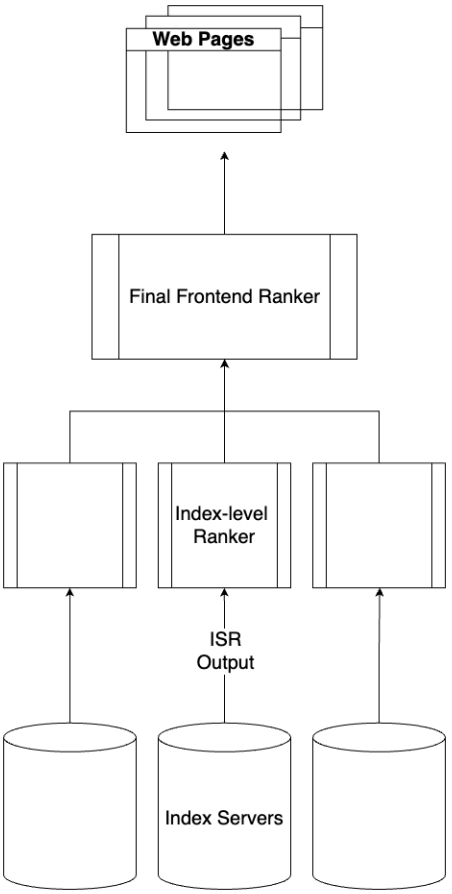
To build the recursive ISR tree structure, all five of our ISR classes (Phrase, And, Or, Container, Word) inherit from a common base ISR class via polymorphism, allowing non-word ISRs to take other ISRs as child arguments. Internal ISRs combine and manipulate their children to search for documents matching the query constraints. At the leaves, ISRWord nodes directly access posting lists from the index. At the root ISR, we can use the NextDocument() and GetDocumentID() methods to iterate through matching documents.

Ranking

For ranking, our engine leverages a two-stage ranking system for improved efficiency and better search results. The first stage ranker is located at the index server level, which is ranked solely on page heuristics. These documents are then sent to the front end from each index server to be ranked by the second stage ranker. The second stage ranker uses machine learning in the form of eXtreme Gradient Boosting (XGBoost) as a learn-to-rank model to predict the rank of pages based on their given static and dynamic characteristics. A diagram outlining this architecture can be seen on the right.

Regarding page heuristics, our ranker considers both the static qualities of a page as well as a dynamic ranking score based on query matches within a page.

- **Static data:** document length, title length, outward links, PageRank score, CheiRank Score, Louvain community index and size
- **Dynamic Data:** query title matches, query body matches



To compute network-based heuristics like Louvain community detection, PageRank, and CheiRank, we made a sparse network class to calculate static network-based heuristics. Data from all index chunks were aggregated and fed into a minimum perfect hash. This maps N unique URLs to a unique ID from 0 to (N-1), which were used to build the network. Heuristic scores were then calculated using this network and saved as metadata within the index. Dynamic network-based were not considered, since it would require loading or building a network at query time. Networks are extremely interconnected, so they are complicated and perform poorly when distributed. When on a single machine, they are still very memory expensive and slow.

For these reasons, dynamic network-based heuristics are rarely used in modern search engines.

For the index-level (first stage) ranker, each index server uses purely heuristic-based ranking. These rankers use weighted sums to compute a static and dynamic score for each document, and then computes a final heuristic score by computing another weighted sum of the static and dynamic scores. Each index server then picks the top *M* documents to send to the frontend based on the final heuristic scores, where *M* is determined proportionally to the size of the index chunk. This allows for more parallelism regarding ranking, as each index server can individually prune pages to reduce the load of the frontend ranker.

For the final frontend (second stage) ranker, we chose XGBoost, a distributed gradient-boosted decision tree machine learning model. It's known for its speed and data efficiency, while still performing on par with neural networks. For each query, data can be fed into an ensemble of small trees in parallel, instead of propagating through a deep neural network. To train our model, we generated a list of sample queries and acquired the search results from Google for each of the queries. These results (URLs) were assigned relevance scores based on their position in Google's response.

Frontend
After the second stage / frontend ranker finishes ranking all of the documents, the frontend server returns an HTML page with search results to the user that orders them from highest score to lowest score. Each search result that is returned has a clickable title link, the URL, and a text snippet to go along with it.

Evaluating Performance

Performance on Specific Queries:

When searching for something using our engine, forcing the engine to use a Phrase ISR instead of an AND ISR provides much better results. The Phrase ISR was best because by explicitly looking for documents that have a certain phrase where all the words occurred in consecutive order, that inherently provides better results for what the user is looking for. However, if a user types words in the search bar without surrounding them in quotations, an AND ISR would be used. But because an AND ISR looks for documents where all words have occurred on the document, it can provide a lot of irrelevant results that just so happen to coincidentally contain those search terms. An example of this is shown below with the search "brawl stars"

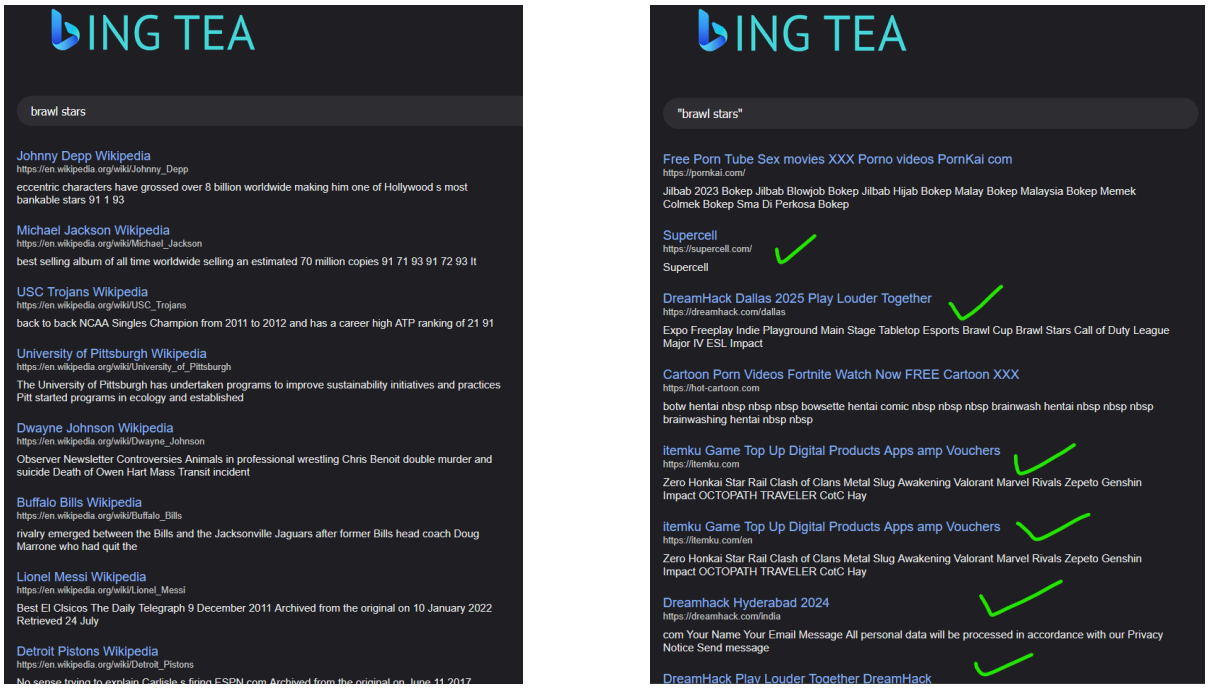


Figure 3: Unforced phrase search (left) vs. forced phrase search (right)

Existing Bugs:

- A search result on the frontend returns “amp” instead of “&”, as seen above, or things like “nbsp”
- A returned text snippet may not contain the actual search query
- There sometimes may be duplicate search results returned
- The index servers receive malformed requests from the frontend time to time
- The frontier process deadlocks on very rare cases where more than two crawler processes crash at the same time

Reflection and Future Steps

If we were to turn our work into an actual product, even if only for our own personal use, there are a couple of next steps / improvements we could do. For one, the crawler/frontier could have been made more robust. Currently, its mechanism of determining whether or not a site is good enough to crawl is kind of lackluster. It doesn’t filter out pornographic content or word spam very well. It crawls these sites, as well as sites that have long titles, long URLs, sketchy/suspicious domains. Another thing we could do is capture anchor text and use that in our ranking process.

One part of the project that turned out to be harder than expected was the Index Stream Readers. Because our search engine didn’t use DocEnd postings lists, our implementation is probably different from other groups. We couldn’t follow the pseudocode on the slides exactly. To elaborate, instead of a Posting List class having many entries where each Entry was a singular occurrence of a term, we had a Posting List class have a Post class, and within this Post class were the usual entries. This Post class was our alternative to the DocEnd postings lists. It stored the actual document identifier associated with the PostEntries. However, to

make matters worse, our very first implementation used the actual URL string itself as an identifier. This was bad because of two reasons: 1) wastes a lot of space – each URL could be 40 characters long which would be 40 bytes used just for the identifier. 2) made implementation of certain ISRs like Container and AND harder because there was no way of telling which document was before or after another document just based off of the URL string alone. Once we found out this mistake, we fixed it so that the document identifier in the Post class was a unique ID. This saved a lot of space – our sample toy index went from 3.1 GB to 810 MB. Additionally, now with a way to tell the ordering of documents, the problematic ISRs could be implemented.

An additional improvement we should make is to implement both ranking steps with XGBoost. Instead of a weight sum, the index ranker can be implemented as an XGBoost rank classifier, labelling results are relevant or not, while still being very fast. Most of the query time is spent searching the index, and the actual ranking is negligible in comparison.

If we had to do it all over again, we would use DocEnd postings lists instead of the Post class structure. And with DocEnds, the overall size of the index could be reduced by a lot of MBs (or even GBs). If we had more time, we would've liked to improve the crawler/frontier and its detection of good sites.

To improve the course for future semesters, we think it'd be beneficial to have another scheduled check-in meeting with Nicole Hamilton. We think having three mandatory, scheduled meetings with her would be good to help keep us on track as well as check in with our designs and progress. It could also be a good opportunity for her to raise concerns about our designs/choices without getting too deep in the weeds of the implementation. The first meeting could be right after the Project Plan is due, the second meeting could be 2 or 3 weeks after we return from Spring Break, and of course, the third and final meeting is at the end of the semester.