C++lue;

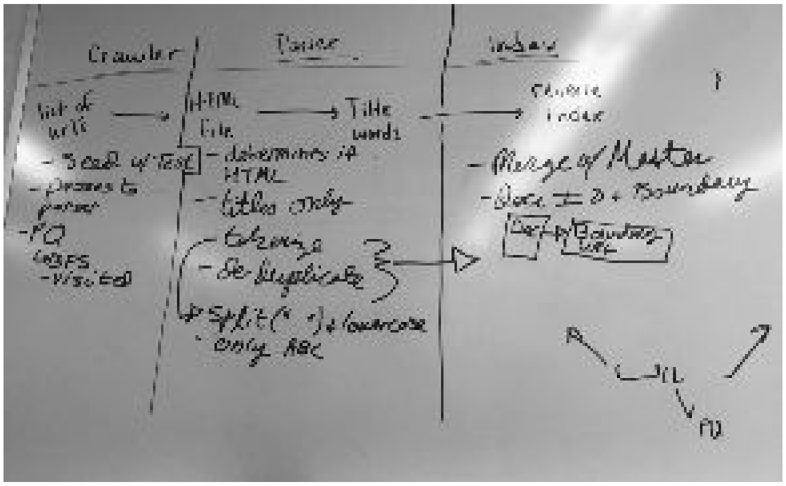
EECS 398 Search Engine Team
Final Report

Anvi Arora, Ben Bergkamp, Jake Close, Veronica Day, Zane Dunnings, Nicholas Yang

April 16nd, 2018

Overview

We began the semester with multiple, 2-3 hour weekly team meetings where we began to design and scope the various components of the project. After creating a rough initial system diagram and discussing how the API's between the classes would look, we split the group of six into three groups of two. We made an effort to figure out which pieces of the projects could be parallelized and worked on separately.

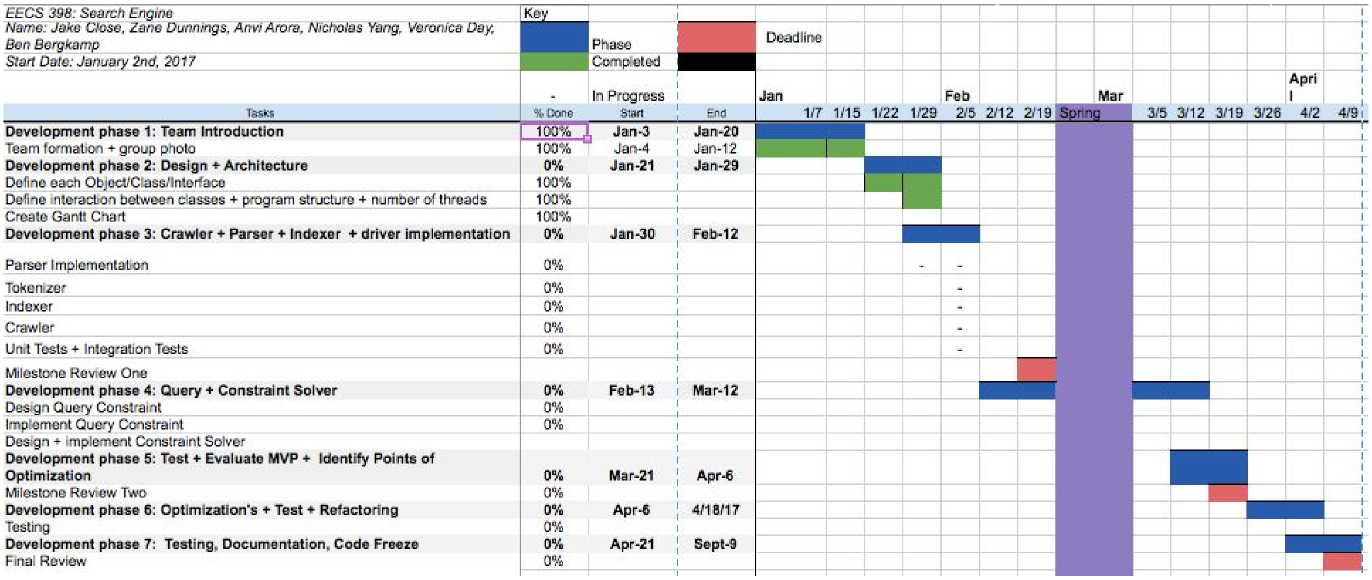


(Image of one of our first whiteboarding brainstorming sessions)

Our major design decisions were typically done in iterative processes. We would define one way that we thought the interfaces should communicate, then once we researched and discovered better approaches, we would reevaluate and refactor the changes. Some of our major design choices included using the producer consumer relationship for the crawler-url frontier, parser-url frontier, parser-indexer, and constraint solver- ranker.

Initially Jake and Ben worked on the Crawler, Anvi and Veronica on the Parser, and Nick and Zane on the Indexer. We would meet twice weekly for about an hour after class to discuss issues, problems and changes to the interface. Once those components were done, Jake and Ben worked on the Constraint Solver, Anvi, Zane and Veronica worked on the Ranker, Nick continued improving the indexer and worked on ISRWord, and Zane worked on the Query Language.

Our development process and communication were key parts to our success. We each used Clion as a our debugger, linter, and build system which helped save valuable time. We also set up a vagrant machine to run all our code in a consistent environment, although we ended up leaning towards using Mac OSX for the majority of the project. We used the GitLab hosted on the UMich EECS servers for our version control. We made sure to follow best practices by testing new components thoroughly, branching for each new features, and merging only stable builds to master. Outside of our twice-weekly meetings, we were constantly messaging in our GroupMe messaging app, figuring out times to meet up and solve issues.



EECS 398: Search Engine
Name: Jake Close, Zane Dunnings, Anvi Arora, Nicholas Yang, Veronica Day, Ben Bergkamp
Start Date: January 2nd, 2017

Key
Phase
Completed
In Progress
Deadline

Tasks	% Done	Start	End
Development phase 1: Team Introduction	100%	**Jan-3**	**Jan-20**
Team formation + group photo	100%	Jan-4	Jan-12
Development phase 2: Design + Architecture	**0%**	**Jan-21**	**Jan-29**
Define each Object/Class/Interface	100%		
Define interaction between classes + program structure + number of threads	100%		
Create Gantt Chart	100%		
Development phase 3: Crawler + Parser + Indexer + driver implementation	**0%**	**Jan-30**	**Feb-12**
Parser Implementation	0%		
Tokenizer	0%		
Indexer	0%		
Crawler	0%		
Unit Tests + Integration Tests	0%		
Milestone Review One			
Development phase 4: Query + Constraint Solver	**0%**	**Feb-13**	**Mar-12**
Design Query Constraint	0%		
Implement Query Constraint	0%		
Design + implement Constraint Solver			
Development phase 5: Test + Evaluate MVP + Identify Points of Optimization	**0%**	**Mar-21**	**Apr-6**
Milestone Review Two	0%		
Development phase 6: Optimization's + Test + Refactoring	**0%**	**Apr-6**	**4/18/17**
Testing	0%		
Development phase 7: Testing, Documentation, Code Freeze	**0%**	**Apr-21**	**Sept-9**
Final Review	0%		

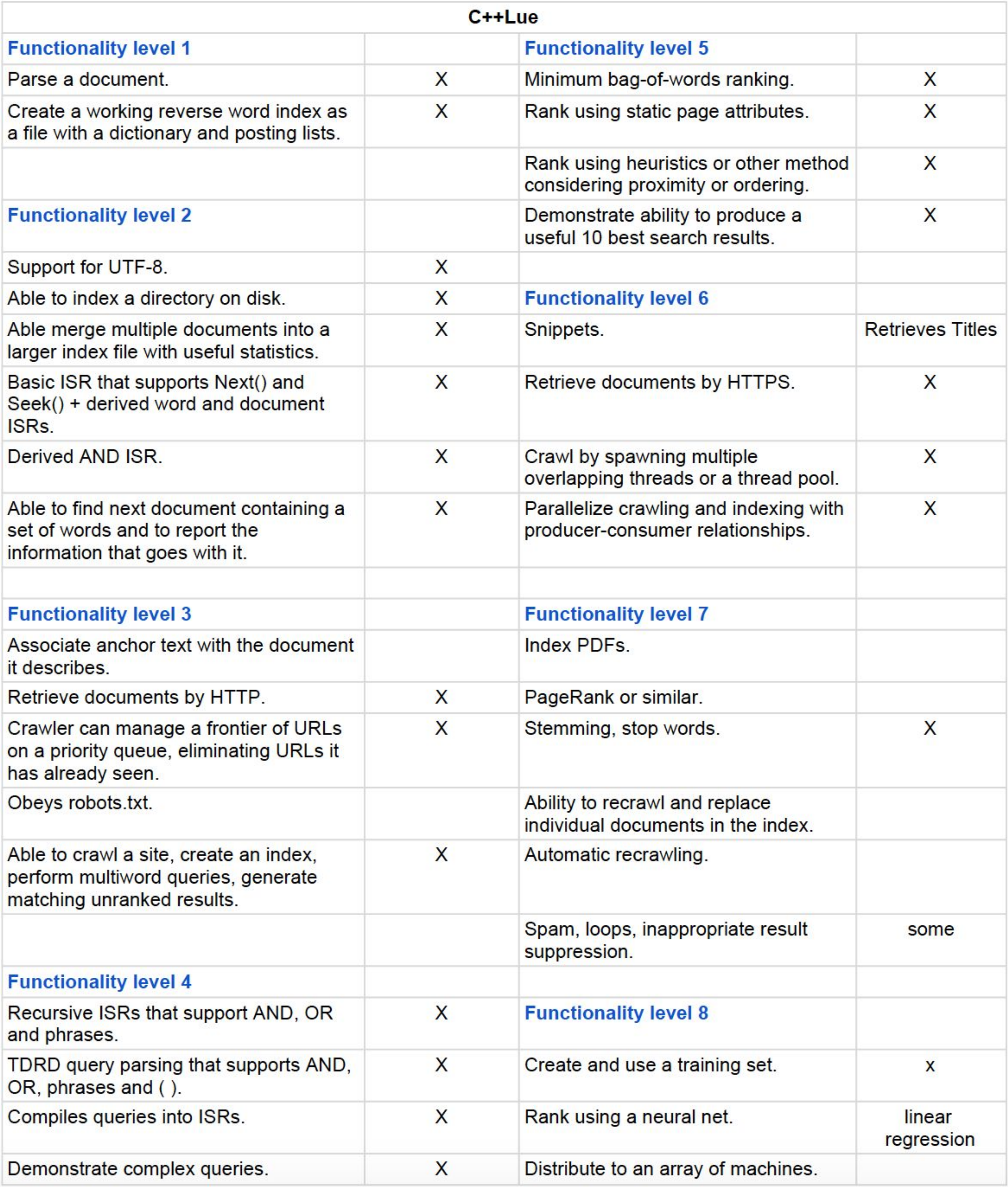
(Screenshot of our original Gantt chart of progress deadlines)

Spreadsheet of major components and who did what in LOC

We used the CLOC terminal script (Count Lines of Code) to computed lines of code count for each file, removing comments and whitespace.

	Anvi	**Ben**	**Jake**	**Nick**	**Veronica**	**Zane**	TOTAL PER PART
Parser	371	0	0	0	1568	0	1939
Crawler	0	430	430	0		0	860
Indexer	0	0	0	488		0	488
Query Language	0	0	50	0	86	564	700
Constraint Solver	0	187	187	405		0	779
Ranker	150	0	15	0	522	600	1287
Server/Web	0	0	300	0	30	0	330
Util	80	300	100	358		540	1378
TOTAL PER PERSON	601	917	1082	1251	2206	1704	7761

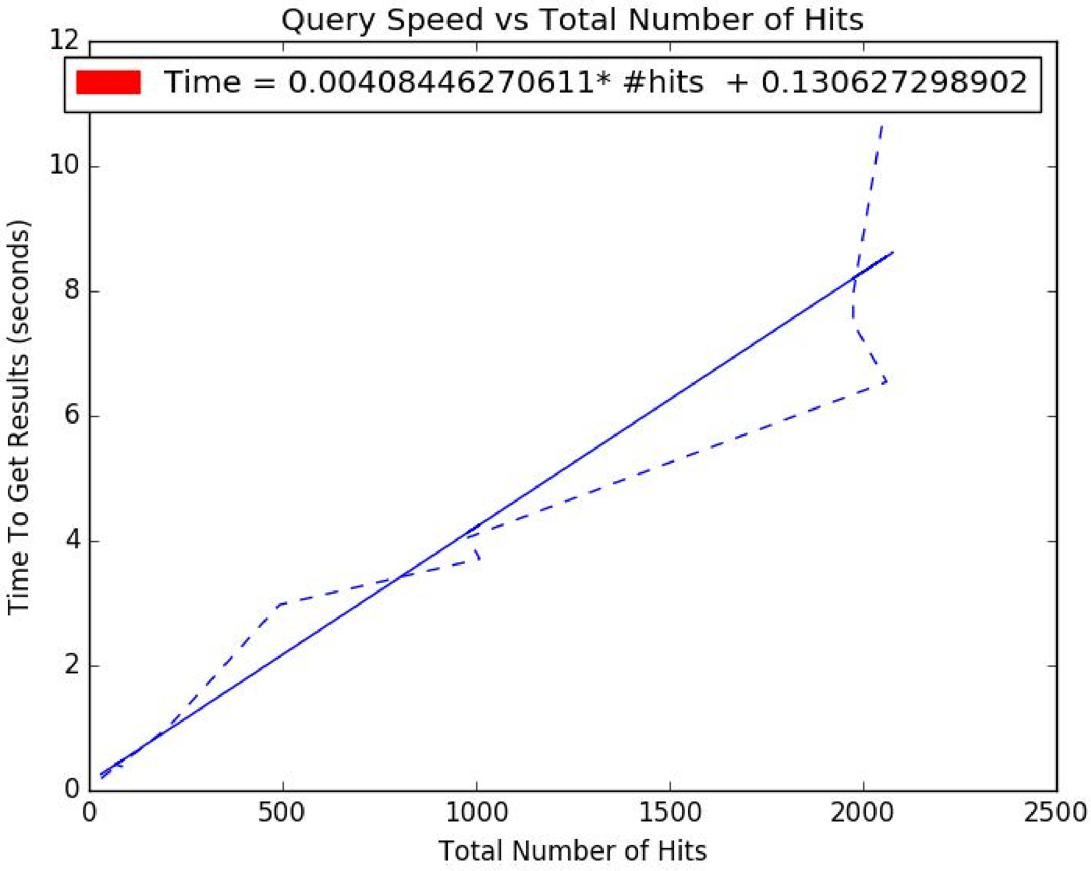
Functionality Level: ~7



C++Lue			
Functionality level 1		**Functionality level 5**	
Parse a document.	X	Minimum bag-of-words ranking.	X
Create a working reverse word index as a file with a dictionary and posting lists.	X	Rank using static page attributes.	X
		Rank using heuristics or other method considering proximity or ordering.	X
Functionality level 2		Demonstrate ability to produce a useful 10 best search results.	X
Support for UTF-8.	X		
Able to index a directory on disk.	X	**Functionality level 6**	
Able merge multiple documents into a larger index file with useful statistics.	X	Snippets.	Retrieves Titles
Basic ISR that supports Next() and Seek() + derived word and document ISRs.	X	Retrieve documents by HTTPS.	X
Derived AND ISR.	X	Crawl by spawning multiple overlapping threads or a thread pool.	X
Able to find next document containing a set of words and to report the information that goes with it.	X	Parallelize crawling and indexing with producer-consumer relationships.	X
Functionality level 3		**Functionality level 7**	
Associate anchor text with the document it describes.		Index PDFs.	
Retrieve documents by HTTP.	X	PageRank or similar.	
Crawler can manage a frontier of URLs on a priority queue, eliminating URLs it has already seen.	X	Stemming, stop words.	X
Obeys robots.txt.		Ability to recrawl and replace individual documents in the index.	
Able to crawl a site, create an index, perform multiword queries, generate matching unranked results.	X	Automatic recrawling.	
		Spam, loops, inappropriate result suppression.	some
Functionality level 4			
Recursive ISRs that support AND, OR and phrases.	X	**Functionality level 8**	
TDRD query parsing that supports AND, OR, phrases and ( ).	X	Create and use a training set.	x
Compiles queries into ISRs.	X	Rank using a neural net.	linear regression
Demonstrate complex queries.	X	Distribute to an array of machines.	

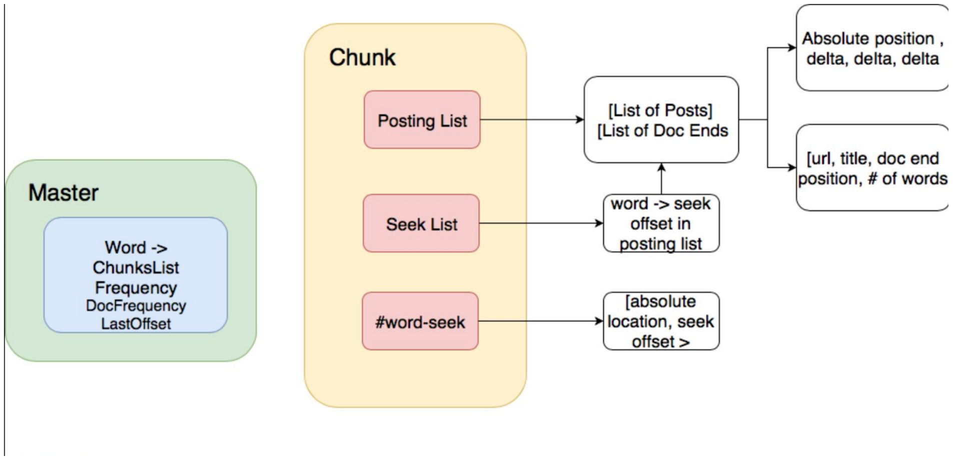
Basic statistics

a. Total Docs = 50,000 docs
b. Total Size = 157 MB
c. 10 crawler worker threads
d. Crawl Speed = ~100 docs every 10 seconds
e. 6,632,329 total tokens indexed
f. 228,465 unique tokens (may be inaccurate to our use of stemming)
g. Graph below depicts query speed:



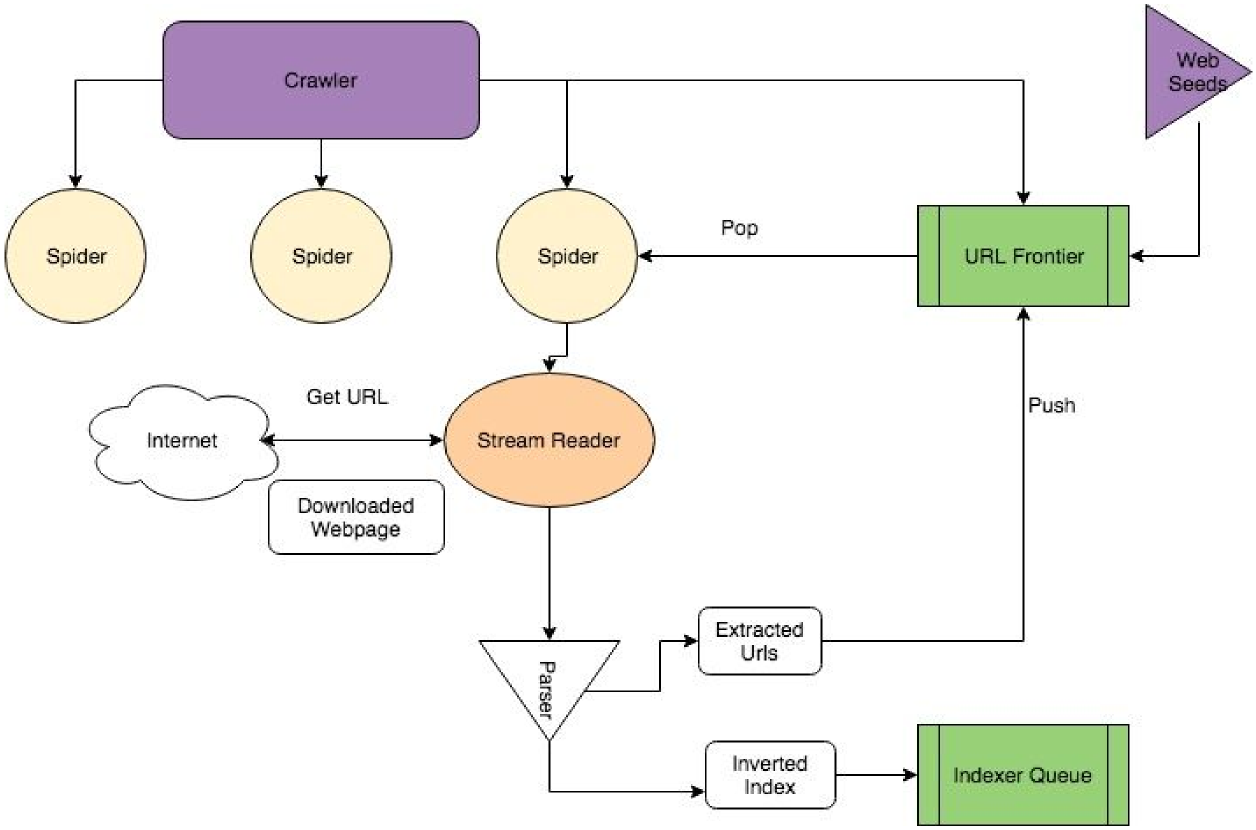
We found the speed to be linear in time with the number of hits. We believe we could have reduced this time if we had ran one thread per chunk while running the constraint solver, optimized some in-memory seek data, and stored more cached data on disk.

2. Diagram of filesystem


Master
Word -> ChunksList Frequency DocFrequency LastOffset
Chunk
Posting List
Seek List
#word-seek
[List of Posts] [List of Doc Ends
word -> seek offset in posting list
[absolute location, seek offset >
Absolute position , delta, delta, delta
[url, title, doc end position, # of words

- Master.txt (disk hash table)
    - Contains word as key, and chunk lists, frequency, doc frequency, and last offset for the value.
    - Also stores statistics such as number of words, number of unique words, number of chunks, etc, decorated with “=”.
- Chunks are comprised of three files, #.txt, #-seek.txt, and #-wordseek.txt
    - #.txt is the actual postings list. The first offset for each word is the absolute location.
    - #-seek.txt (disk hash table) is the seek list for every word in the chunk, also stored as a disk hash table. The seek location is simply stored as the value, and all the constraint solver needs to do to parse the file is search up the term as the key, and get the seek location on the actual postings list.
        - Also stores statistics such as the number of words, number of unique words, and number of documents, etc, decorated with “=”.
    - #-wordseek.txt (disk hash table) is the dictionary/”synchronization point” for frequently used words. Each word is simply stored as the word appended with a number, with the number acting as a partition. The key value used is a listing of <absolute location, seek offset> where absolute location is the location in the entire corpus, and the seek offset is where to seek in the postings list to get to that location.
    - One thing that was experimented with was making just a single wordseek file, eliminating the seek list and perhaps lowering the amount of space required for a single chunk. However, this resulted in significantly larger space per chunk, because there were many terms with only one or two postings. We decided to continue with the current working file structure.
    - Chunk size: can be adjusted per build in the IndexerConstants.h file to any size desired.

3. **Diagram of index build process**

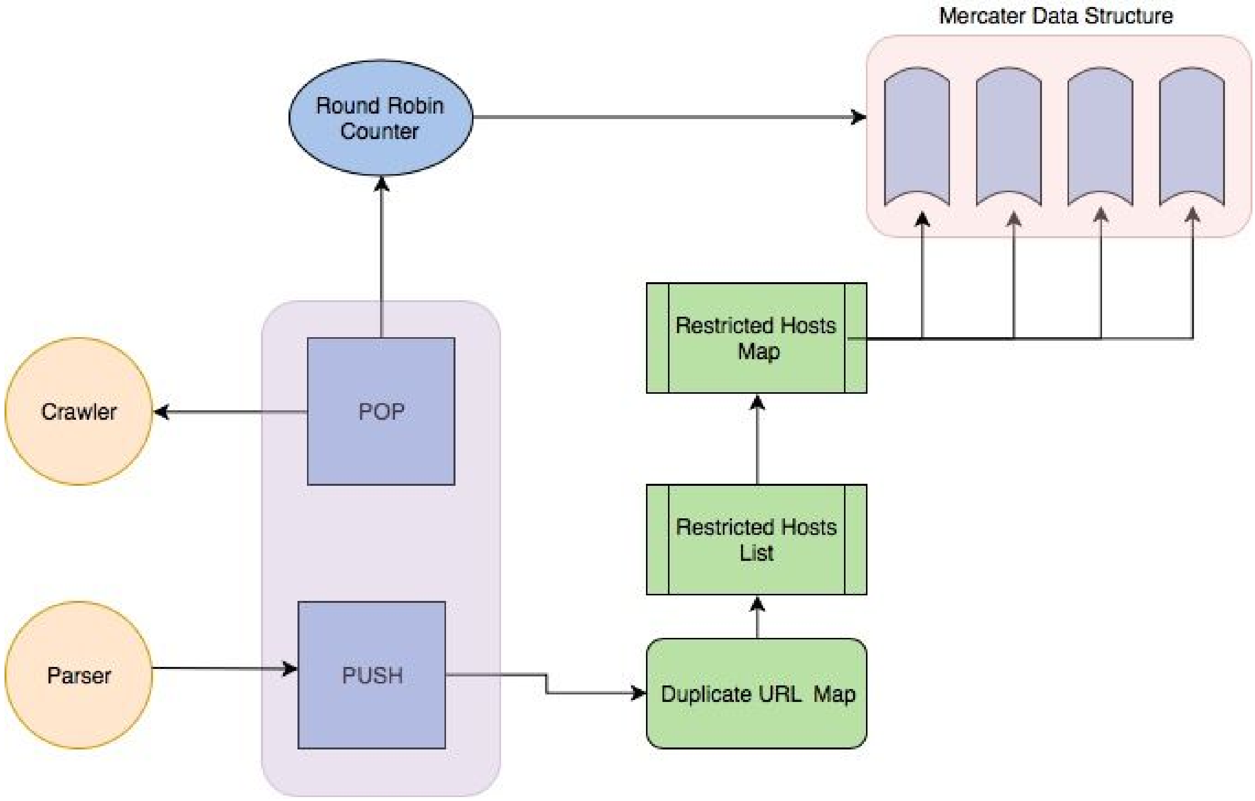


(Detailed Diagram at Build Index Process)

Our build process was able to run with a dynamic amount of crawling threads and a single indexing thread. The spiders and the indexer were each given a pointer to a shared memory producer consumer queue. On start up, the crawler would spawn the chosen amount of spiders threads. Each spider had its own stream reader and Parser class. The spider would retrieved a new url off of the Url Frontier, download the web page, pass it to the Parser whom would push the final inverted index to the Indexer Queue. The Indexer would pop an index off the queue and process it in memory. Once the in memory data structure reached a determined size, the chunk would be written to disk

Strategy

For modularity, we implemented a Url Frontier class which handled the web crawling strategy. This allowed us to give the spiders and parsers, push and pop API's and easily changing the strategy. Our goal for the crawler was to find a diverse set of the highest quality websites possible. Also, we wanted to obey some form of politeness and not spam a host with many requests at a time. When we original began, we utilized a single queue, and found that there were many weird/irrelevant sites getting indexed. Our attempt to fix this was but adding a blacklist for domains we wanted to ignore. However, we quickly realized hand picking all of the bad sites was not scalable. Then, we tried a Whitelist and implemented a time buffer for each domain so the queue would prioritize the least recently seen domains. This was better but we were still getting stuck with a lot of the same sites and was still not diverse enough. Finally, we came upon the mercator round robin structure which provided us with the best diversity in crawl as well as a form of politeness. This strategy utilized a priority queue for each one of the whitelisted domains.

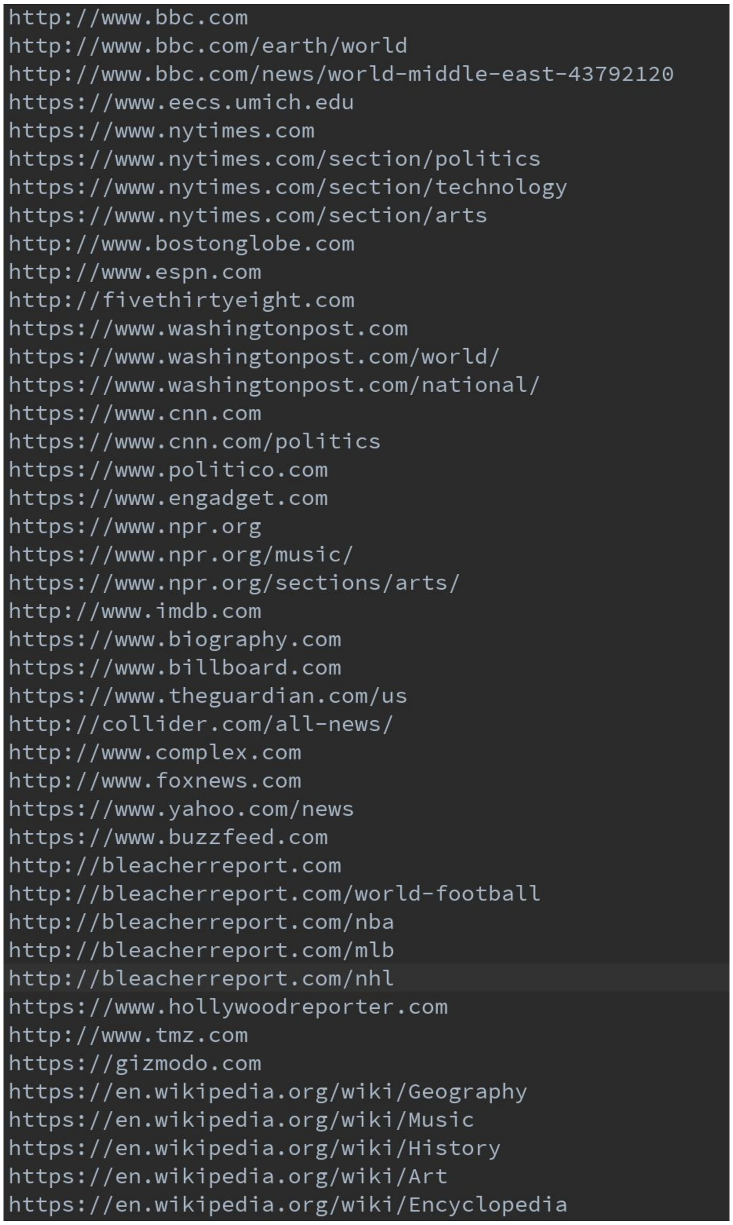


(Diagram of Url Frontier and Multi Queue Structure)

Seeding

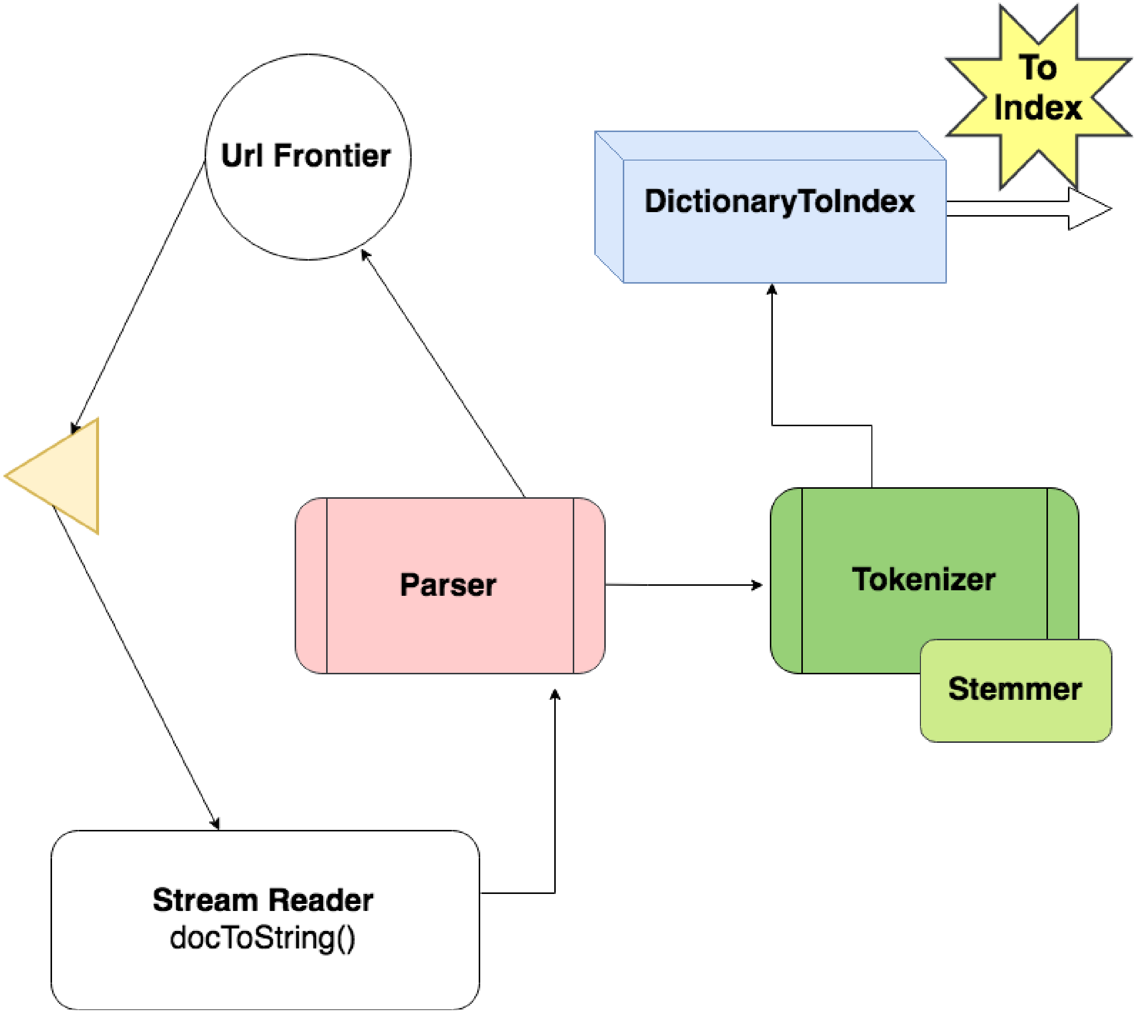
We determined our seeds by running multiple Google searches and recording which sites were consistently in the top ten. We removed sites from the seed list that we found were not easily crawled, or prevented us from downloading their pages.

An example list of the seeds we used:



```
http://www.bbc.com
http://www.bbc.com/earth/world
http://www.bbc.com/news/world-middle-east-43792120
https://www.eecs.umich.edu
https://www.nytimes.com
https://www.nytimes.com/section/politics
https://www.nytimes.com/section/technology
https://www.nytimes.com/section/arts
http://www.bostonglobe.com
http://www.espn.com
http://fivethirtyeight.com
https://www.washingtonpost.com
https://www.washingtonpost.com/world/
https://www.washingtonpost.com/national/
https://www.cnn.com
https://www.cnn.com/politics
https://www.politico.com
https://www.engadget.com
https://www.npr.org
https://www.npr.org/music/
https://www.npr.org/sections/arts/
http://www.imdb.com
https://www.biography.com
https://www.billboard.com
https://www.theguardian.com/us
http://collider.com/all-news/
http://www.complex.com
http://www.foxnews.com
https://www.yahoo.com/news
https://www.buzzfeed.com
http://bleacherreport.com
http://bleacherreport.com/world-football
http://bleacherreport.com/nba
http://bleacherreport.com/mlb
http://bleacherreport.com/nhl
https://www.hollywoodreporter.com
http://www.tmz.com
https://gizmodo.com
https://en.wikipedia.org/wiki/Geography
https://en.wikipedia.org/wiki/Music
https://en.wikipedia.org/wiki/History
https://en.wikipedia.org/wiki/Art
https://en.wikipedia.org/wiki/Encyclopedia
```

Parser



(Detailed Diagram for Parsing Process)

Overview

The parser, once it received a StreamReader object from the Crawler, would call docToString() on the object in order to retrieve the entire html in one string. The parser used various custom string processing functions in order to find the text within the html that we deemed important.

Tag Detection

Once the parser found a "<" opening tag, it continued to index the string, looking for what type of tag it was encountering, calling various "extract" functions to extract full strings from within the tags. Tags the parser handled include body, paragraph, anchor, and title tags. Each extract function would pull out the full string and pass it to the Parser's Tokenizer instance. Any tags that did not have detectable closing tags were ignored, such as “meta” tags. Any detected urls found in anchor tags were sent to the crawler after a checking for validity (i.e. no pdf files).

Tokenization

The tokenizer accepted a string, and a decorator char. If the string was a url (determined by the decorator), the Tokenizer split the text based on any symbols found. For all other strings the Tokenizer split the text on white spaces, removed symbols such as “!”, and case folded all chars so the string was entirely lower case. Stop words were removed and all other words were stemmed. The individual tokens were concatenated with their decorator at the beginning of the token, then stored in a document inverted index (an unordered map that lived on the heap), with the token as the key, and a list of offsets as the value, relative to the beginning of the document. Once all strings in the document had been parsed and tokenized, a pointer to the dictionary was sent to the Indexer, who was responsible for merging this dictionary with the master Index, normalizing offsets as relative to the beginning of the index, and deleting this dynamically declared document index.

Stemming

https://tartarus.org/martin/PorterStemmer/

One important design decision was to incorporate a stemmer into our tokenization process. We knew we would have a limited about of memory, and stemming would reduce the length of our master Index, by storing similar words in the same location. Stemming should also increase recall for our search engine, since stemmed queries can be mapped to more terms.

For example, the query “why are cats so happy” after stemming and stop word removal will yield to “cat happi” and search results such as “Happy Cats” will also be processed to “happi cat”. In this was more relevant document will be returned by the constraint solver.

We used the Porter Stemmer algorithm to accomplish this. The algorithm removes the endings of words incrementally through a 5 step process. Word endings such as “s”, “ing” and “ment” are removed, and only if that word’s measure (the count of Consonant-Vowel-Consonant patterns) were greater than 1.

Query Language

Overview

The query language was designed to read in a query from the user, extract the important words, and generate a parse tree from that query. We chose to insert decorated nodes into our tree, rather than have the constraint solver generate the decorated token. We support queries that are nested with parentheses and support both ORs and AND. If not specified, we assume words are ‘AND’ed together. We support the following for AND: ‘and’, ‘ ’, ‘AND’, ‘&’, and ‘&&’ and the following for OR: ‘or’, ‘OR’, ‘|’, ‘||’.

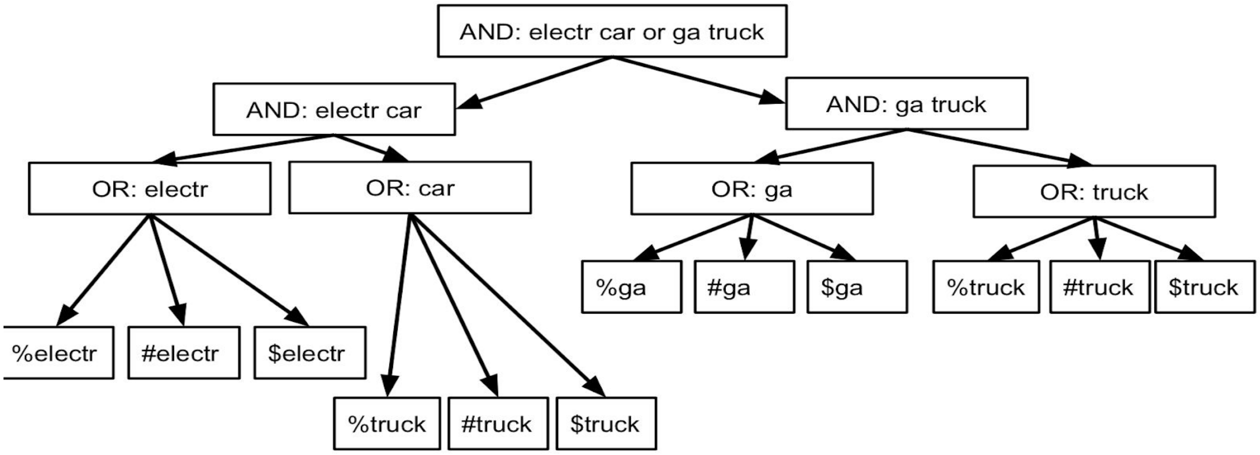
Query Preprocessing

We support the following methods of cleaning our queries:

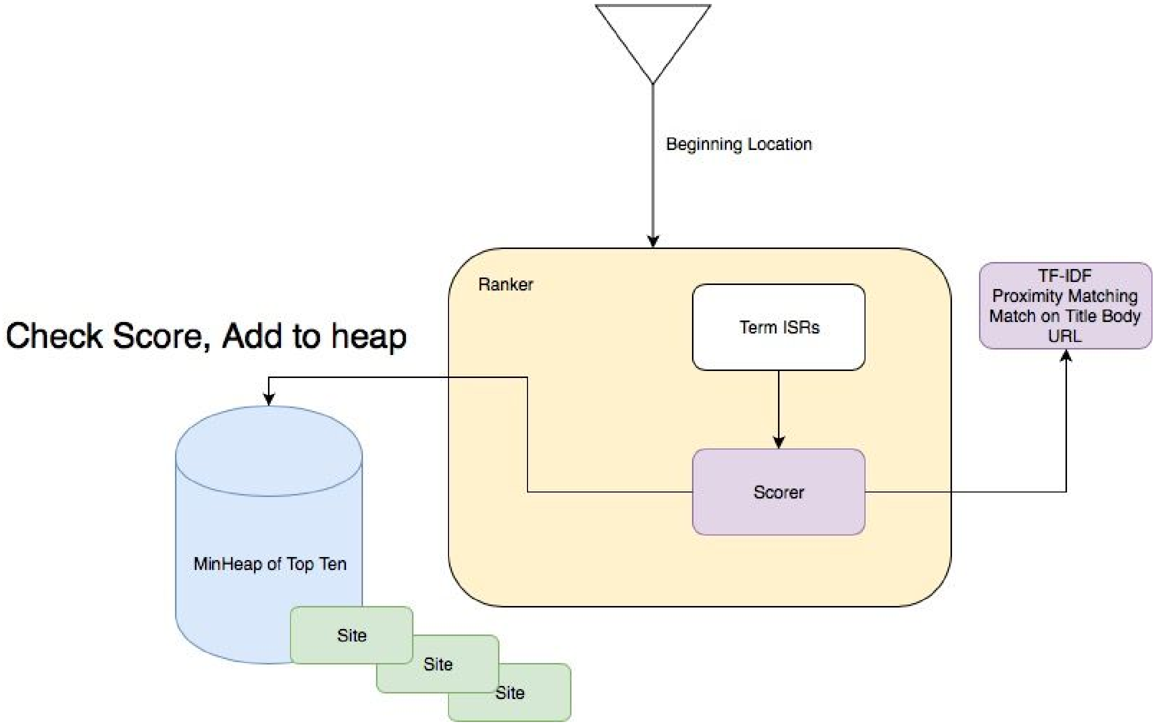
1. Bracket cleanup: the query language detects either '(' or ')' symbols, and places a space before and after. Since we separate on whitespace, we avoid any errors in assuming "(word" is one word since it is transformed into " (word" in the preprocessing stage.
2. Removing stop-words: our query language removes the same stops words that we remove in the text parser. Removing words that don't add any extra context ensure our system is searching for the most relevant keywords in a given query. Also, these words do not appear in the index, so we would return no search results if we ANDed a stop-word. The only modification is that we include 'or' and 'and' since they are needed for context in the query language.
3. Lowercase all words: this complies with the parser
4. Stem the query words to match what's stored in the index
5. Remove symbols: we remove all symbols. For the following : '|', '||', '&', '&&' we replace them with OR, OR, AND, and AND respectively.

AND / OR Logic

After the string has been preprocessed, the QueryParser object separates the string on whitespace, creating a vector of each individual word or symbol in the query. The parser then separates the words in the query based on any OR not within brackets, and then recursively calls itself on that subset of the original query. If no ORs are found, it splits on ANDs, and if none of those are found then it resolves the remaining query as an individual string. During the process, if any parentheses are found, the function searches for the matching parenthesis and calls itself on that subset of the query. Here is an example of a query tree generated from the Query "What are some electric cars or gas trucks?"



Ranker



Overview

The Ranker object consisted of a list of ISRWords. Each document that was returned by the constraint solver was read into a Site object. Each Site represented one potential document to be returned, and had a dictionary object that mapped each word in the document and query to a list of relevant information, such as frequency of word in document and offsets, that was obtained from the ISRWords for that document.

Each Site was sent to the Scorer, an object managed by the Ranker. Our Scorer objected ran through four different heuristics to rank a document: Static ranking, TF-IDF, Proximity Matching, and Location scoring. Each score was multiplied by its coefficient, determined from our linear regression model. After a final score was computed, it was normalized by dividing by the sum of coefficients for each heuristic. This ensured that all of our scores were relative and fell within the range of 0-1.

Static Ranking

Static ranking took into account the TLD of the url. We assumed that urls that contained TLDs such as ".com" and ".edu" would contain more relevant information that other TLDs. We used the Wikipedia article "List of Internet top-level domains" to support our scoring of each TLD.

TF-IDF*

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

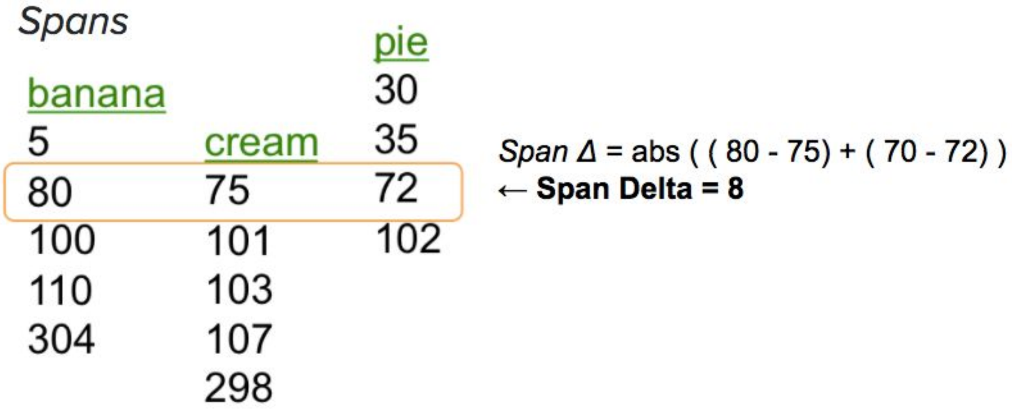
TF-IDF, term frequency times inverse document frequency, is a bag of words model, which does not take into account the order or context of words in a document. We used this scoring method to balance the raw term frequency query terms that appeared in the document with its respective rarity in the corpus. We multiplied the term frequency (normalized by the number of words in the query) by the log of the total number of documents in the corpus divided by the document frequency of that particular word. We summed up the scores for each word, then divided it by the number of words in the query to get an average tf-idf score for each document. This score correlates to how often rare words that are in the query are also found in that document, and scores accordingly.

*We were unable to fix a bug in the TF-IDF function that was causing the score to sometimes result in negative infinity. This resulted in use having to leave this scoring function commented out in production due to lack of time.

Proximity Matching

Spans

For a proximity match score, used the heuristic that words that were near each other in the query would also appear in the same location in a relevant document. Documents that contained exact phrases that matched the query phrase should also be weighted higher than other documents. In order to find exact phrases, and also take into account near-exact phrases, we used the "spanning" algorithm taught in lecture. To do this we first found the rarest word in the query (the term with the least amount of occurences in the document that was being scored). Next we found all other terms in the query that were also in the document, and found the minimum offset distance between each term and the rarest term. We then calculated the delta, or average distance, between each term in the “span”. A span consists of the query terms that are closets in order of appearance in the document.



(Image representation of a span, span delta calculation)

We then took the sum of each span delta and divided this by the number of total spans in the document. This value represented the average length, or window, of terms that would be necessary to include all the query terms in that window.
The lower this number, the higher the score for that document should be, so we took the reciprocal and multiplied it with the number of unique terms in the query. We then multiplied this result by alpha, a weight determined by the linear regression modeling. This was then added to the number of exact phrases in the document, or span deltas of one less than the number of words in the query, divided by the number of spans. This number too was multiplied by alpha prime, which was determined in the linear regression modeling.

$$\text{score} = \alpha \frac{|\,query\ words\,|}{\frac{\sum_{i\,\epsilon\,Scans}^{|\,scans\,|-1} abs(Scan\,\Delta_i - Scan\,\Delta_{i+1})}{|scans|}} + \alpha' \frac{|\,exact\ phrase\,|}{|\,scans|}$$

Max Score
We wanted to normalized this number by dividing the score by the highest score possible for a document. At first we assumed that the best document would only consist of query terms and would only consist of exact phrases.We realized this yield an unrealistic score and resulted in low scores for even high ranking documents. To counter this, we still assumed that the average span delta was one less than the number of unique terms in the query, multiplied by alpha. Instead of adding alpha prime multiplied by 100% exact phrases, we found beta, which represented a more realistic percentage of exact phrases that are found in a relevant document.

$$maxScore = \alpha \frac{|\,query\ words\,|}{|\,query\ words\,|-1} + \alpha' * \beta$$

To find beta, we wrote a python script, goldStandardDoc.py, to run random phrase queries in Google. We then took the top five results returned from each query, parsed the document and found the number of exact query phrases in the document's text. We divided this number by the number of N-grams in the document, N equal to the number of terms in the query. This yield a percentage of exact phrases found in the document. We then took the average of each percentage of exact phrases found in each of the five documents returned for each query. Through this subset, we found that a top-ranked result on Google consists of about 0.06% exact phrase matches on average. This was number we used to multiply with alpha prime. If the score we calculated ended up being higher than our estimated exact score, we set it equal to our exact score and divided by itself, so our scoring was still normalized and fell within the bounds of 0-1.

Word Locations

The Scorer finds the number of instances of each of word of interest in the query in each respective location, and multiplies the frequency/totalsize with a corresponding weight. This weights documents with the query words in the URL and title very highly, but also gives some weight if the words of interest occur many times in a document.

Machine Learning

We found that our hand tuned coefficients were working generally well, and ranking was returning relevant and quality results. However, we decided to create a simple machine learning tool in order to improve the results. We used a linear regression model, trained on google results to produce the coefficients we wanted.

Training Set

Our training set consisted of the top result on each of the page of some queries google result. If a Google query returned 10 results per page, we extracted page 1, 11, 21, 31...etc. Since the top Google results are extremely relevant, and have little differentiation, we chose to choose results with more variations in features to help our program learn what the best pages look like.

$$\tau = \frac{P - Q}{P + Q}$$ [4]

Loss Function

To calculate the error in each step of the gradient descent, we use a evaluation metric called the Kendall's Tau function. The metric evaluates how similar two ordered lists are. In our case, we perform recursive descent where we increase the coefficient that increases the Kendall's Tau score the most. To evaluate the cost you create a nxn matrix where n is the number of documents to rank. The binary matrix has the value 1 at (i, j) if r(i) > r(j), and a zero otherwise, where r(i) is the i-th document in the set of documents r. You compare two matrices (ordered set of documents) by counting P (number of elements that agree), and Q (number of elements that disagree), and calculating tau.

Potential Improvements

There are some things we'd love to improve about this machine learning approach that we think would greatly improve its accuracy:

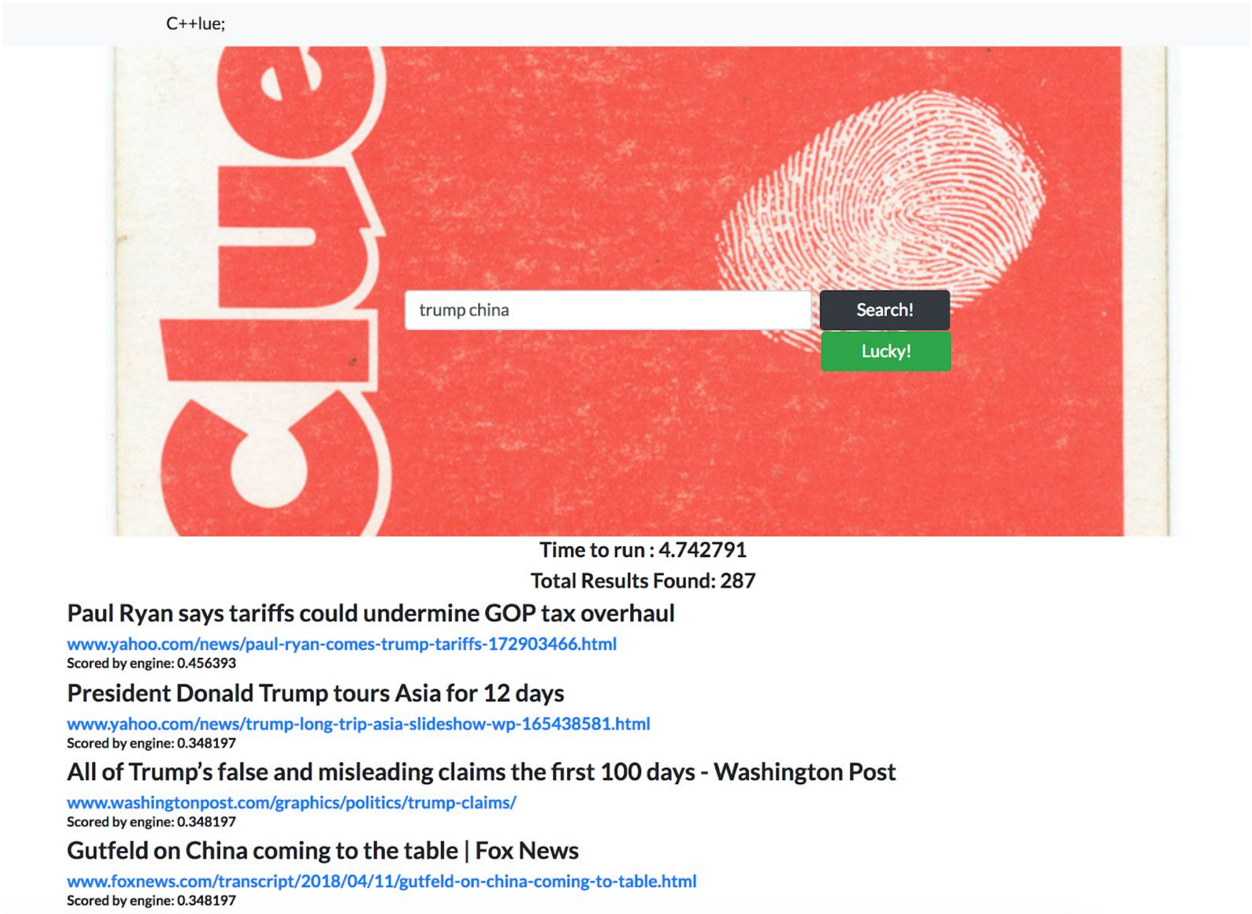
- Fix gradient "jump" with Kendall Tau: With each step in gradient descent the ranking of the documents may not change, so the loss function might not register a "good" increase in a docs score. Therefore, the gradient calculated will "jump" when there's a change in ranking.
- More data: In our case, it was very difficult to extract features from all of the pages, so getting enough training data was tough, and I think increasing data would have made this really powerful.
- Non-Linear regression: I think it would have been useful to experiment with logistic regression and other types of non-logistic regression. It's not likely to have a set of ranked documents be scored in a linear way.

Results

Our engine is able to return good relevant results for terms that are very common, however it is slightly slower because there are more pages to rank. Terms that less common return faster results but less relevant. 'AND' and 'OR' queries are relatively fast as well, although not as fast as simple word queries. We don't anticipate this is a huge issue given the fact that queries are on average 2.3 words.

Server

We built a web server in C++ using sockets to send the results via JSON format. The front end was implusing jQuery and HTML to parse and display the top ten results including links, scores and url. Below is a screenshot.



trump china

Time to run : 4.742791

Total Results Found: 287

Paul Ryan says tariffs could undermine GOP tax overhaul
www.yahoo.com/news/paul-ryan-comes-trump-tariffs-172903466.html
Scored by engine: 0.456393

President Donald Trump tours Asia for 12 days
www.yahoo.com/news/trump-long-trip-asia-slideshow-wp-165438581.html
Scored by engine: 0.348197

All of Trump's false and misleading claims the first 100 days - Washington Post
www.washingtonpost.com/graphics/politics/trump-claims/
Scored by engine: 0.348197

Gutfeld on China coming to the table | Fox News
www.foxnews.com/transcript/2018/04/11/gutfeld-on-china-coming-to-table.html
Scored by engine: 0.348197

Bugs

a. One known bug is that sometimes the title of a page returned is: “301 Forbidden” or “404 Not Found”, even though the link below is valid. We believe this is due to parsing these returned values incorrectly in some rare cases.
b. We weren’t able to get Tf-Idf functionality working, we think this may be because method of retrieving document frequency is buggy
c. Front End fails when JSON isn’t returned properly
d. Linear regression training data only includes a subset of the features, so some of the coefficients are hand tuned

Production

There would be a fair amount of work required to get this product to production. While this product would never be able to compete with modern search engines like Google or Bing, it could fill the need for a local file system searcher. In order to adapt our engine to be a pure local file system searcher, it would require another few weeks or refactoring our code to be local file specific. These changes would include changing our parsing logic to get every word, instead of looking for specific html tags, and detecting which files are binary files by looking at how high the byte values are. Another option would be to focus our search engine to a certain “world” of data, such as a sports search engine, or a search engine solely for Wikipedia.

4. **What would we do differently:**
   a. Crawler:
      i. Add an ability to start/stop the crawler.
      ii. Obey robots.txt
   b. Constraint Solver:
      i. Have a thread per chunk of the index to query reduce speed
      ii. Not and Phrase ISR’s
   c. Indexer:
      i. Introduce more compression to the file system (a lot of room on disk is dominated by our seek and wordseek disk hash tables).
   d. Query Language
      i. Phrase matching
      ii. NOT functionality
      iii. More error checking for mismatched brackets and missing words
   e. Ranker
      i. Fix TFID
      ii. Include a large set of heuristics
   f. Machine Learning
      i. More training data
      ii. Non linear regression
      iii. Better loss function

Appendix:

https://nlp.stanford.edu/IR-book/html/htmledition/the-url-frontier-1.html
http://cloc.sourceforge.net/
https://tartarus.org/martin/PorterStemmer/
[4] Hung-Mo Lin & Vernon M. Chinchilli (2011) A MEASURE OF ASSOCIATION BETWEEN TWO SETS OF MULTIVARIATE RESPONSES, Journal of Biopharmaceutical Statistics, 11:3, 139-154, DOI: 10.1081/BIP-100107654
https://en.wikipedia.org/wiki/List_of_Internet_top-level_domains