Maximal Munch Search Engine - Final Report

April 22, 2019

Table of Contents

Team Maximal Munch

Brandon Kayes – CSE – Sophomore
280 IA, 281, 370 - Career in Industry

Ryan Wunderly – CSE – Junior
482, 445 - Robotics & Computer Vision

Alexander Raistrick – CSE / Math Minor – Junior
Extensive self study in machine learning – Career in Research

Austin Kiekintveld – CSE – Sophomore
281 IA, 370, 388 – Systems Programming & Security

Adolfo Apolloni – CSE – Junior
Industry Autonomous Vehicle Experience & 281, 388, 370 - Autonomous & Intelligent Systems

Daniel Hoekwater – CSE / Business Minor – Sophomore
281, 370, ENTR 390 – Career in Industry

Section 1 - Executive Summary

This report describes in detail the entire Maximal Munch search engine created by Adolfo, Alex, Austin, Brandon, Daniel, and Ryan in EECS-398: System Design of a Search Engine. It is laid out in sections beginning with the team organization, code contributions, and features supported by the engine in **Sections 2 - 4.** We then discuss the file system footprint, overall query engine performance, and overarching architecture in **Sections 5 - 7.** In **Sections 8 - 15**, we summarize the engine's overall architecture as well as the design and performance of each of its eight modules. We conclude with a holistic analysis of the system, its performance, known bugs, potential for future work, and reflections in **Sections 16 - 20.**

Overall, the Maximal Munch Team delivered a distributed search engine with a total index of over 2 TB distributed across 12 machines. The team crawled 150 million pages of the internet (~14 TB of HTML content) and indexed 116 million of those pages. From raw HTML to the index itself, we were able to achieve over 5x data compression. We developed a distributed architecture for the crawler and indexer processes as well as for the full end-to-end query serve engine. This distributed engine enabled scanning the index, finding matching results, ranking them, and serving them to the frontend to display to the user.

For many queries, we achieved a serve time of under one second from our full index using a network of fourteen machines. While some queries had substantially worse performance, our median response time of five seconds, considering our index size, was sufficient for a minimum viable engine. A more detailed breakdown of performance can be found in **Sections 7 and 15.**

The entire search engine, including our template library and search engine components, consisted of 26,887 lines of code developed over 4 months. The template library was vital to the success of our search engine, the most notable module being our hashmap, which had 3x the speed and ⅓ the memory overhead of the STL equivalent.

Despite some known bugs, documented in **Section 16**, and an extensive analysis in **Section 17** of potential future tasks, we believe the search engine is a minimum viable product in its current state. If it is running on enough distributed machines and has a relatively small pool of users, it is able to efficiently serve results. The code base is stable and rarely (if ever) crashes, providing the robustness necessary to serve the engine to end users.

To our instructors: Nicole Hamilton, Carolyn Busch, and Kevin Li, we understand this report is extremely long. **Sections 2-7** and **16-20** summarize the team dynamics, core results, and reflections on the project and the course as a whole. We recommend you certainly read those. **Sections 8-15** exist to do justice to the development process and features of each module including the challenges encountered during the process. Please read or skim those time permitting and at your own leisure (or just look at the figures).

Section 2 - Team Organization, Functionality, and Procedures

Our team consists entirely of members who knew each other prior to the class. Daniel, Brandon, and Austin attended high school together, while Alex and Adolfo met them through classes at the University. Alex met Ryan while working in the Michigan Autonomous Aerial Vehicle Vehicles - Guidance and Navigation team, and he was added to the group on the first day of class.

We chose to use an AGILE development process for this team project. The codebase was maintained on GitHub, and we used biweekly sprints to track progress. To ensure high code quality of our final codebase, all code required at least two peer reviews before being merged into the master branch. Additionally, we used Slack for all team communication and had bi-weekly team meetings regularly attended by all members.

Since we are all friends with relatively similar schedules, we largely coded as a group; this was particularly helpful when people found bugs or ran into roadblocks. Most modules were developed in teams of at least two to promote an environment where members could resolve blocking issues together.

In all, we managed to maintain a fairly even work load. Adolfo developed the HTML parser and stemmer modules. Brandon, Daniel, and Ryan were the principal crawler developers. Adolfo, Austin, and Ryan developed the index. Adolfo, Brandon, and Ryan developed the constraint solver and query serve architecture. Daniel developed the query parser. Alex wrote the Ranker. Brandon and Austin developed the initial Server. Adolfo and Alex extended the server to the Multi-server distributed architecture.

Section 3 - Code Contributions

This section summarized the total lines of code (LOC) written on this project as well as lines of code per team member by submodule. This is detailed in **Table 1** below.

Table 1: Lines of code summary table for the Maximal Munch search engine

Module	**Submodule**	**Total LOC**	**Adolfo Apolloni LOC**	**Daniel Hoekwater LOC**	**Brandon Kayes LOC**	**Austin Kiekintveld LOC**	**Alex Raistrick LOC**	**Ryan Wunderly LOC**
HTML Parser		880	880					
Crawler	Crawler Minion	1712	100	756	656	100		100
	Crawler Master	1710			1240			470
	Crawler	254		254				

	Aggregator							
Indexer	Stemming	670	670					
	Index Serialization	954				954		
	Index Build	784	666			118		
	ISRs	1027					243	784
Constraint Solver		350	175		175			
Query Compiler		522		522				
Ranker	Static	208					208	
	Url	206					206	
	Crawl	119					119	
	Dynamic	352					352	
	Utilities	390					390	
Server	Query Serve	348	174		174			
	Single Server	774			232	542		
	Distributed Serve System	611	183		214		214	
Frontend		695				109	586	
Utilities	Chunk Checker	369				369		
	Batch Rankers	331					331	
	Command Line Engine	102	51	51				
Team Library[1]	Hash Map/Set	1346				1346		
	Vector	505				505		
	Deque	725					725	
	Heaps	482				482		
	Math	907					907	

[1] The library was built before Ryan Wunderly joined the team thus he did not have the opportunity to work on it and would not be expected to have code contributions in this section.

	Libraries							
	Functions[2]	1588	357	450	144	250	387	
	Other Data Structures[3]	1287	459	200	228	400		
Other		141			141			
Test Cases		5888	800	925	925	875	1514	849
Makefiles		407		70	207	100	30	
Experiments		243		24	75		144	
Total		**26887**	**4515**	**3252**	**4411**	**6150**	**6356**	**2203**

Section 4 - Search Engine Functionality Checklist

Below is a summary table of our search engine's supported functionality. This outlines all features supported and not supported by our engine

Table 2: A summary of all features supported by our search engine

Module	**Feature**	**Supported**
HTML Parser	Fully parses HTML pages in a robust manner	Yes
Crawler	Hash Map & Hashing Function Chosen	Yes
	Manages a frontier of URLs and prioritizes them	Yes
	Is polite (robots.txt, does not DDoS anyone, etc.)[4]	Yes
	Automatic Recrawl	No
	Crawler is crash resistant, automatically restarts, and does not lose data on crashes	Yes
	Retrieves documents over both HTTP and HTTPS and also handles redirects	Yes
	Parallelized by Multi-threading	Yes

[2] These include string functions, conversion functions, sorting functions, etc.
[3] This includes bloom filter, trie, our own unique pointer, etc.
[4] We initially had issues with accidental Distributed Denial of Service attacks. These have been resolved. We apologize for any complaints filed by other institutions early on.

	Identifies and suppresses loops, spam, etc.	Yes
	Is distributed across a network of machines	Yes
Indexer	Determined index file, format, numbering, what attributes were captured, and how to gather useful statistics	Yes
	Demonstrate that you can build a reverse word index as a file with a dictionary and a posting list for each token.	Yes
	Create an index stream reader class that can seek. Derive word and document ISRs.	Yes
Constraint Solver	Create a derived and working AND, OR, Phrase ISR and support parenthesis	Yes
	Demonstrate working TDRD parsing and compile a query into a structure of ISRs	Yes
	Support Stemming & stop word elimination	Yes
	Demonstrate compiling and running a query producing unranked results.	Yes
	Derive and demonstrate container ISRs	No
Ranker	Rank using a bag-of-words technique.	Yes
	Rank using static page attributes.	Yes
	Rank using heuristics or other method considering proximity or ordering.	Yes
	Demonstrate ability to produce a useful 10 best search results.	Yes
	Create and use a training set.	Yes
	Rank using a neural net or other ML technique. (In the static ranker)	Yes
	Support PageRank or similar. (A custom ML trained static ranker considers neighboring static / url ranks)	Yes
Server & Frontend	Create a simple command line interface.	Yes
	Create a simple HTTP server as a wrapper UI.	Yes
	Report title, clickable URL.	Yes
	Serve results from a distributed system consisting of a network of	Yes

	machines each with a subset of the index	
Additional Features	Index off of a distributed system using several machines	Yes
	Associate anchor text with the document it describes in the index	Yes
	Create a snippet to go with reported hit	No

Section 5 - File System Footprint and Index Statistics

Below, **Figure 1,** explains the layout of our filesystem. Our file system has three primary components. First, Google Drive for aggregation and and long term storage. Second, Alex's desktop which contains locally files for testing and an 8 TB drive with a full copy of our index. Third, our digital ocean cloud platform which has the entire 2.2 TB index distributed across 13 minion machines. Together this is our total file system footprint. Note that each index directory contains a collection of .index file which follow the index format in **Section 10.** All crawl files contains aggregated pages stored in .agg files.

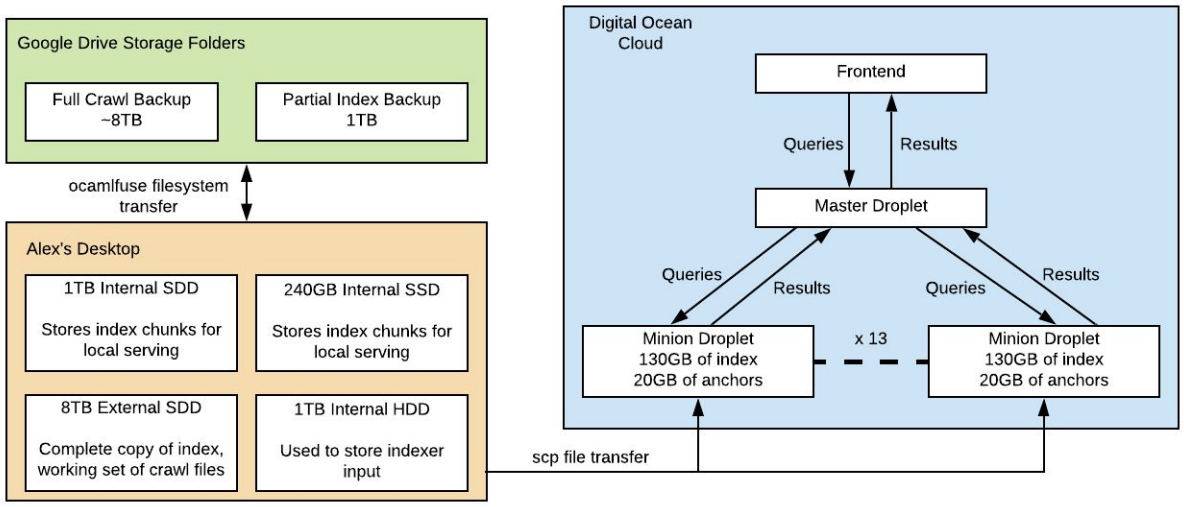

Figure 1 - Filesystem Footprint Diagram

Section 6 - Query Serve Performance Statistics Summary

When serving queries on a single host (a Dell XPS laptop running 32 GB of RAM and an 8th generation Intel i7 processor) with 11 million pages, we found relatively performant results. The median query time was 360 milliseconds across 16 test queries that returned an average of 2713 ranked results. We expect this benchmark to be representative of a fully distributed system, which would consist of 10 million pages per machine. This provides a performance benchmark for each minion machine on modern controlled hardware.

When serving the whole index (116 million documents) on a distributed system of 13 machines we found the search engine had a median query time of 5 seconds. This performance drop off was in part due to the older and less capable hardware running on the machines and in part due to imperfect load balancing across the cluster. Despite this fact, these numbers are promising considering the size of the corpus, and we would expect improved performance with a larger cluster of machines with more computing power.

For further analysis of query performance, see **Section 16.**

Section 7 - Search Engine Architecture

Our search engine consists of 5 overarching modules: the parser, the crawler, the indexer, the query engine, and the frontend.

The parser is the first major module. It accepts raw HTML as input and produces a compact digest form suitable for crawl storage and a more processed form for indexing.

The crawler module is composed of a master and a pool of minions. Master is a process which maintains a frontier as a priority queue of URLs to crawl and is also responsible for removing duplicates, performing URL filtering, and ensuring request politeness. Minions send requests for work to master, and receive a set of URLs from the frontier. Each minion maintains a thread pool of workers, which crawl each page and save the digested HTML to Google Drive for storage. Once a chunk is processed, the minion sends back to master any links found on the pages, along with a request for more work.

The index is the heart of the engine. On the build side, it consumes crawl record files, parses them, and compiles the inverted index and document information necessary for queries. This data is serialized to disk and deserialized on query side, providing a barebones interface for the query engine.

The query engine, composed of the query compiler, constraint solver, and ranker, attempts to distill the essence of what a user is looking for out of a text query, perform a mechanical search process over index chunks, and sort results to find the most relevant to the user's query constraints.

The frontend accepts queries as input, forwards them through the distributed server, and receives a set of results, which are displayed for user interaction.

Tying this all together is our library, known as Square One. It includes templates for any relevant data structures and algorithms used in the other modules of the engine.

Section 7.1 - Architecture Diagram

This section provides an architecture diagram for both index build and query serve.

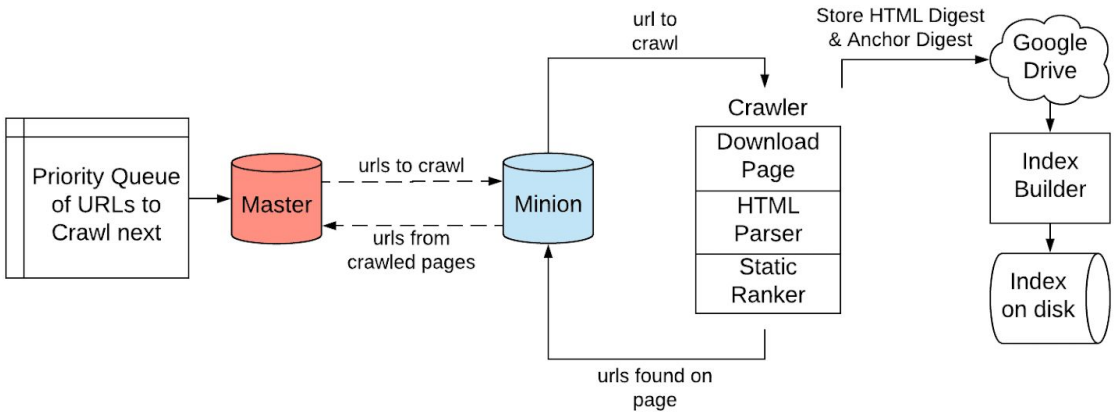

Figure 2: Architecture diagram of the crawler and index build submodules. Note the distributed nature of both the crawler and the index build system.

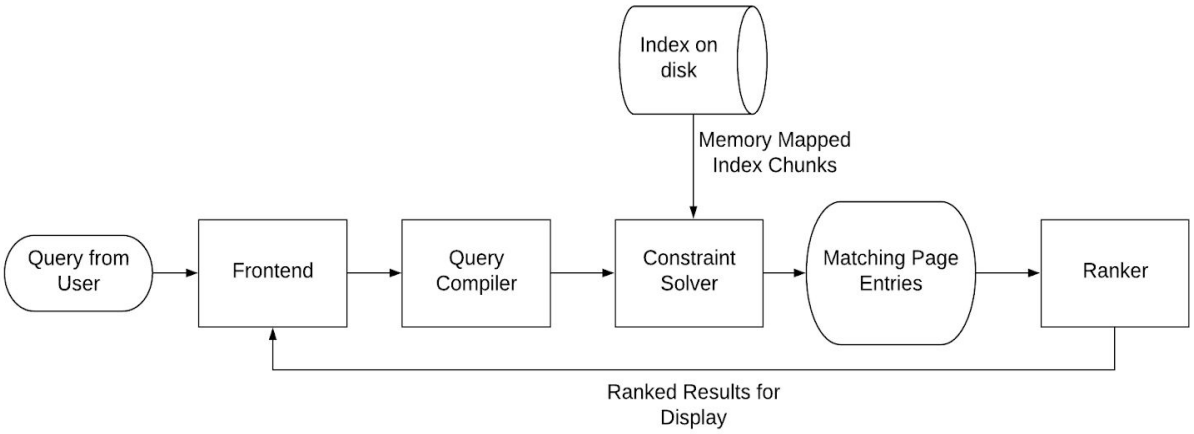

Figure 3: Architecture diagram of the query engine displaying submodules, data storage units, and data pathways.

In addition to the pipeline shown above, it is worth noting that the entire system is distributed across multiple machines each with a subset of the overall index. See **Figure 1** for a more detailed diagram of the filesystem.

Section 8 - HTML Parser

The HTML Parser supports two distinct modes of functionality: initial parsing for the crawl record and final parsing for the indexer. The initial parsing for the crawl record is critical as it allow us to efficiently

store a record of all pages crawled without recording needless information such as scripts, style comments, and images. It also extracts urls and anchor text from the page, which are sent back to the crawler. The complete parsing mode strips all HTML, leaving only plain text for the indexer. Additionally, it extracts key features that are used by the ranker: anchor text, title, URL, headings, and emphasized words.

The HTML parser uses recursive descent parsing and processes HTML in a single pass without any copying. The parser is designed to safely error on invalid pages (including deeply recursive tag structures) which avoids crashes during the parsing process, allowing the crawler to throw out invalid pages. Supporting two modes of operation allows us to perform two independent tasks without adding much code complexity, since functions such as tag extraction, tag comparison, and data extraction are shared between the modes.

Section 8.1 HTML Parser Performance

The parser itself was robust enough to successfully parse ~90% of pages we crawled. Considering the degree of "tag soup" and invalid page formation in our crawl space, we found this impressive.[5]

By using a preliminary parser to strip useless content out of pages before storing the crawl archive, we were able to reduce page size by 50% before storage. This reduction was crucial in order to eliminate wasted storage and disk/network I/O.

Section 9 - Crawler

True to its name, the crawler is responsible for crawling the internet and processing web pages, which are later used in the index of our search engine. The crawler itself is a distributed system that will be covered in detail in the sections below.

Section 9.1 Crawler Architecture

We designed a multiprocess, distributed crawler consisting of a master process and a pool of minion processes. Master's role is to maintain a frontier of pages to crawl, ensure we do not crawl pages that have been crawled before, and ensure politeness across the domains that we are crawling. Each minion receives URLs, downloads and processes the content, stores the digest to Google Drive, and gives master any links found on the page with blacklisted links removed.

The multiprocess design allows us to distribute a fleet of minions across independent machines, including cloud servers. Master and minion communicate over TCP sockets to achieve low latency communication between processes. The message protocol is defined in **Appendix C**.

[5] https://en.wikipedia.org/wiki/Tag_soup

Section 9.2 - Crawler Master
Master is responsible for labor management and distribution. To coordinate this, Master maintains a priority-queue frontier of URLs, a set of crawled URLs, and a pool of Minion connections.

Master was designed to be as efficient as possible to enable support of a large number of minion connections, which means that master interacts with URLs in a very minimal way. Thus time intensive processing that needs to occur for a given URL, such as robots.txt and blacklisting, is done by the minions. This enables us to better utilize the CPU resources available in our system.

The frontier is a max binary heap ordered by Crawl Rank (see **Section 13.3**). To minimize memory overhead, we cap the frontier size at 3 million links, archiving low quality links to disk. This reduces memory overhead and ensures the frontier and hashset are able to fit in RAM.

To keep track of crawled pages, master maintains a hashset of every link ever discovered by a minion. At the end of our crawl, this hashset contained 1 billion distinct entries. We developed new strategies during our crawl to reduce hashset memory usage; instead of storing url strings, we generated a 64-bit hash of each url and stored that hash as an identifier. This reduced our hashset memory by a factor of 10 to a total of 8.6 GB at the end of the crawl.

On startup, master creates a socket on a given port number. Minions make a connection to the port for each batch of urls they would like to receive. Master assigns a thread to each connection.

Master detects minion crashes by looking for closed connections. In the event of a crash, master re-assigns the batch to a new minion. Master is made crash tolerant by backing up its frontier and seen set every 30 minutes. The backup operation is atomic, eliminating worst-case risk of losing all state. The entire system makes the crawler crash tolerant, preventing loss of pages and ensuring we can seamlessly resume crawling in the event of any crashes.

Section 9.3 - Crawler Minion
Minion is a multithreaded worker process which performs tasks assigned by master. It processes each batch by following a producer-consumer relationship. To do so, it maintains its own queue of work and a collection of URLs it has gathered from pages. These resources are shared among several threads: a gatherer thread, a writer thread, and a pool of processor threads, known as “munchers”.

The gatherer thread is responsible for refilling the frontier when it is empty; when it does so, it pulls a new batch from master and places this batch on the local frontier for processing.

Each batch is sequentially processed by a muncher thread. For each link, the muncher connects to the website, checks the domain’s robots.txt, and follows any redirects to land at the final page. It then parses the webpage and stores the digested HTML. When the batch is complete, the writer thread serializes the digested pages to disk then requests a new batch from master.

To minimize network overhead, a minion only writes results back to disk and to master after completing an entire batch of work; this process is coordinated by the writer thread.

Through these behaviors, the master and minions actively discover and process pages. Because of the number of processed web pages, it is imperative to be intentional about the information that is stored.

Section 9.4 Aggregator

When storing a large number of downloaded web pages on one machine, it is crucial to devise a file storage strategy that is easily human-readable and does not bog down the file system. Our system stores each page as readable text concatenated in a file. This allows us to store data from an arbitrary number of scraped pages in one file by simply concatenating the page files together. To manage this, we wrote an aggregator program, which combines the page data from a fixed number of files into one aggregate file. In this manner, we can store all the relevant information for thousands of web pages with the system management overhead of a single file.

Section 9.5 - Storage Strategy

In order to coordinate a distributed web crawler, it is necessary to manage storage across machines. Our method of choice for handling this concern was with Google Drive, which offers virtually limitless storage space and access to data across platforms. All machines running minion processes mounted Google Drive as a directory, and automatically uploaded their aggregated pages to be further processed.

Section 9.6 - Rate Limiting, Diversity, & Denial of Service Challenges

We initially believed that, with a sufficiently large URL frontier, crawling would naturally diversify. This proved to be incorrect when we crashed the servers hosting the website **soup.io.**

We attempted to prevent future incidents like this by penalizing repeat domain occurrences within each batch sent to a minion on a per-batch basis. Unfortunately, this proved ineffective in preventing high rates of traffic against a single site. While no site ever again received the volume of traffic directed at soup.io, the improved system still directed suspiciously high levels of traffic to University of California - Merced's login pages. This problem was reported to us by ITS (we had executed 300,000 queries over the course of 20 hours). We mitigated this by developing a global tracking system for the number of times we visited each domain name. This global tracking system would begin penalizing pages when they hit 0.1% of the last million pages crawled along a custom score which reduced the page's rank to 0 by the time it hit 5% of the last million pages crawled. This upgraded system reduced the peak hit rate (generally on domains with extremely high url ranks) by more than 10x.

These changes, however, did not solve all denial of service issues. We were reported one more time to ITS: this time for crashing the Duke Law School course login and registration databases. In this case, we executed an average of one query every 2 seconds against Duke's databases for 5 hours. Because these were database queries with high amplification factors (large return for a single query), this brought their servers down entirely.

To prevent any further complaints, we added a randomization factor of ± 20% to our frontier ordering to prevent any domain-based URL clumping. This randomized high-ranking links in a virtually identical manner to random sampling, preventing clumping and thus any further denial of service complaints.

Section 9.7 - Multi-Threading & LibreSSl Challenges

During our crawl campaign, we noticed that LibreSSL's libtls appeared to cause a double-free on an internal object when many threads were running at once. Austin filed an issue on the LibreSSL GitHub repository. A few days later, the developers responded that the context structure contains a reference-counted default config object shared across all instances. Multiple threads were having a data race on the reference count, resulting in multiple frees of the object. The developers recommended locking around the initialization and freeing of the objects. Following this change, we saw a steep drop in crawl performance due to contention. After reading through LibreSSL source code, we discovered it is safe to have a single context object per thread each with their own thread-local config. As a result, we only needed a single lock around the thread local initializations that occur once on program launch, and the rest could then be lock-free. As a result, we eliminated the data race and significantly increased crawl performance due to increased stability and zero contention around TLS downloading.

Section 9.8 - Crawl Results

Over the course of our crawl, we downloaded and stored 150 million pages. This included crawling 8.4 million unique domains. After parsing out all unnecessary information from the page (scripts, style comments, etc), this lead to a total crawl archive of 7.4 TB of data. During our crawl, we discovered a total of 1 billion unique URLs. Our peak crawl rate was 25 million pages per day, and the crawl ran for a total of 23 days. This can be seen in **Figure 4** below.

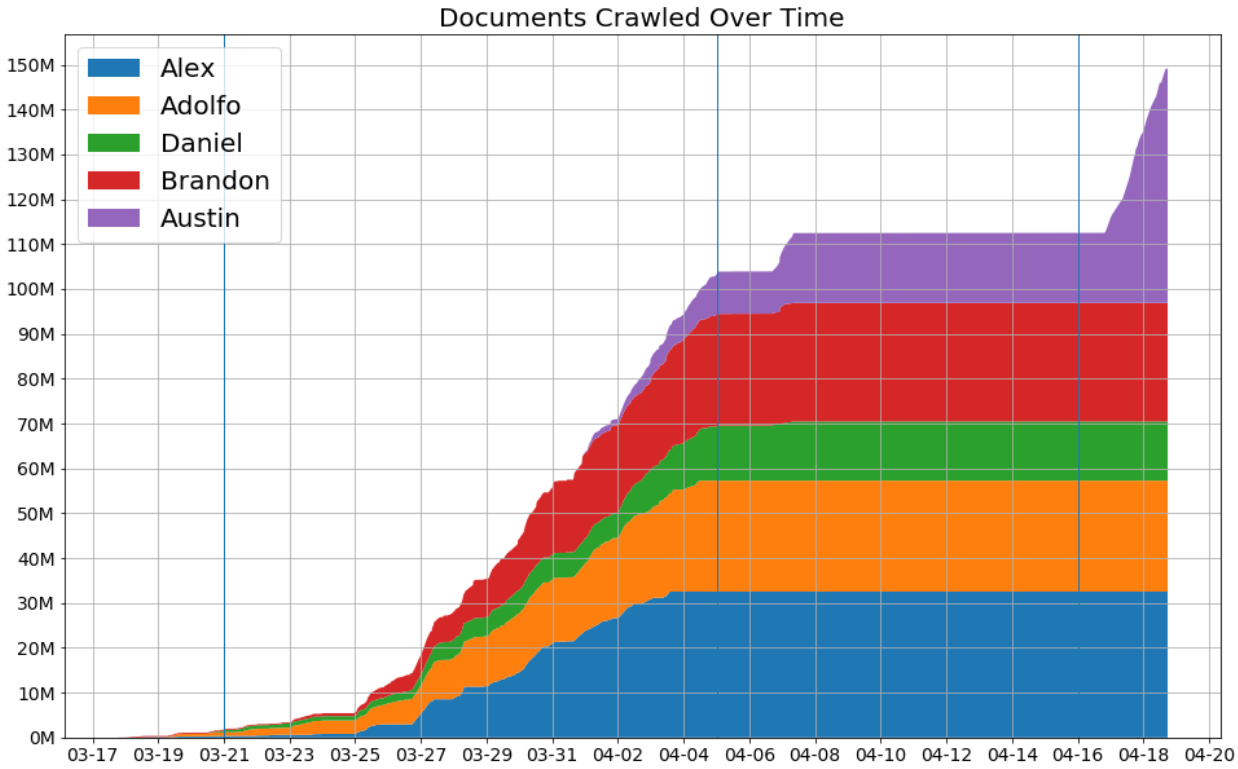

Figure 4: Our distributed crawl by user's unique name. Note that in many cases one username represents multiple machines.

This figure summarizes total pages crawled over time by host username. It shows both our crawl rate and distribution across various machines. The following pie chart in **Figure 5** below breaks down crawling by host machine.

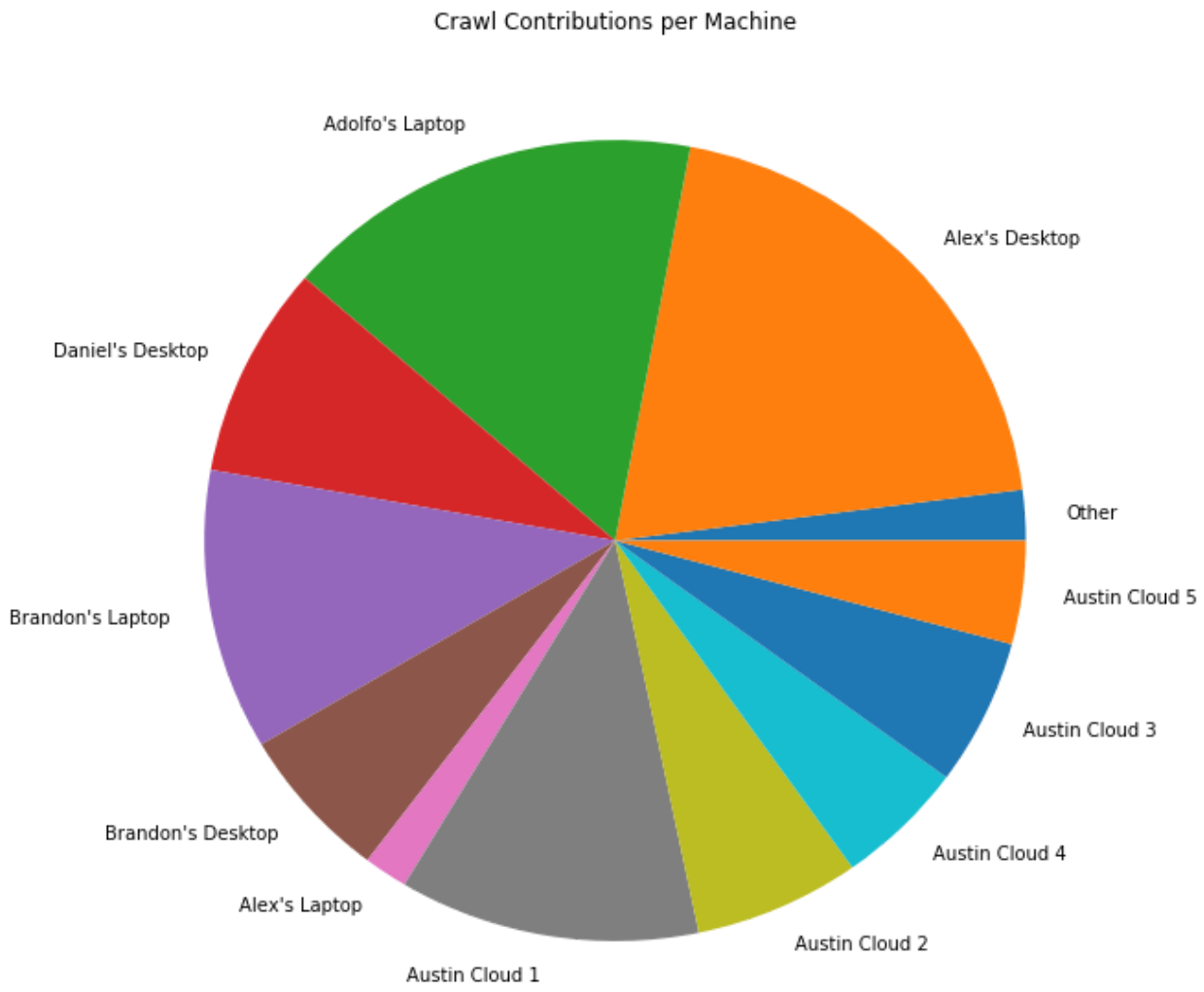

Figure 5: Pages crawled as a percentage by host machine. This shows how many different machines we used in our distributed crawl and the importance of a distributed system in achieving our document corpus.

From this chart we can see the impact of a distributed crawl; no machine contributed more than 20% of our total 150 million crawled pages. We would not have reached these numbers without our peak operation of 8 separate minion machines crawling in parallel.

Section 9.9 - Crawling from the Cloud

Another advantage of distributing our crawler was that we were able to put our crawler on the cloud. Through Digital Ocean's cloud platform, we set up five virtual machines (or "droplets"), which crawled with a total of 1000 minion threads. These droplets crawled 60 million pages and achieved a peak crawl rate of 25 million pages a day. The total cost of the cloud resources used for the crawl was $15.00 (or ~25

cents per million pages). This was enormously successful and demonstrated the feasibility in scaling crawling indefinitely with cloud machines.

Section 10 - Indexer

The following sections detail the indexer module. We cover the index format, how we serialize and deserialize the index, the index build system, and the overall index build performance.

Section 10.1 - Index Format

The index format was designed to be trivially parallelizable, both on the build and serve side, by separating it out into a series of “chunk” files. Each chunk is fully capable of serving queries for the documents it contains and was built independently of all other chunks.

The file format begins with a prefix, including version number, to verify compatibility. The prefix contains all length fields necessary to locate other sections of the file.

Next are the reverse-index and the term storage segments. The reverse index segment is a series of fixed-size key-value pairs, and we store all terms concatenated together with no delimiters. The key is an offset into term storage, and the value is of an offset intoto a seek table. Given more time, we would modify this format to serialize the hashmap directly into the file for memory efficiency on the serve side.

The seek tables are an intermediate step between the reverse-index and the posting lists. Each seek table contains a count of posting list entries and a constant size table from absolute location to byte offset into the posting list, and each seek table is stored sorted by absolute location. Our largest posting lists rarely exceed the 32 memory pages required to make larger table sizes worth the space overhead to reduce paging. This size incurs an overhead of 512 bytes per posting list in each chunk however, and it makes up a significant portion of our index size overall (~62% - see **Figure 6)**. To reduce waste, we distribute the seek entries evenly across the posting list while serializing. Given more time, we would move to a dynamically sized table to reduce paging further. **Figure 6** shows the distribution of memory used by each portion of the index.

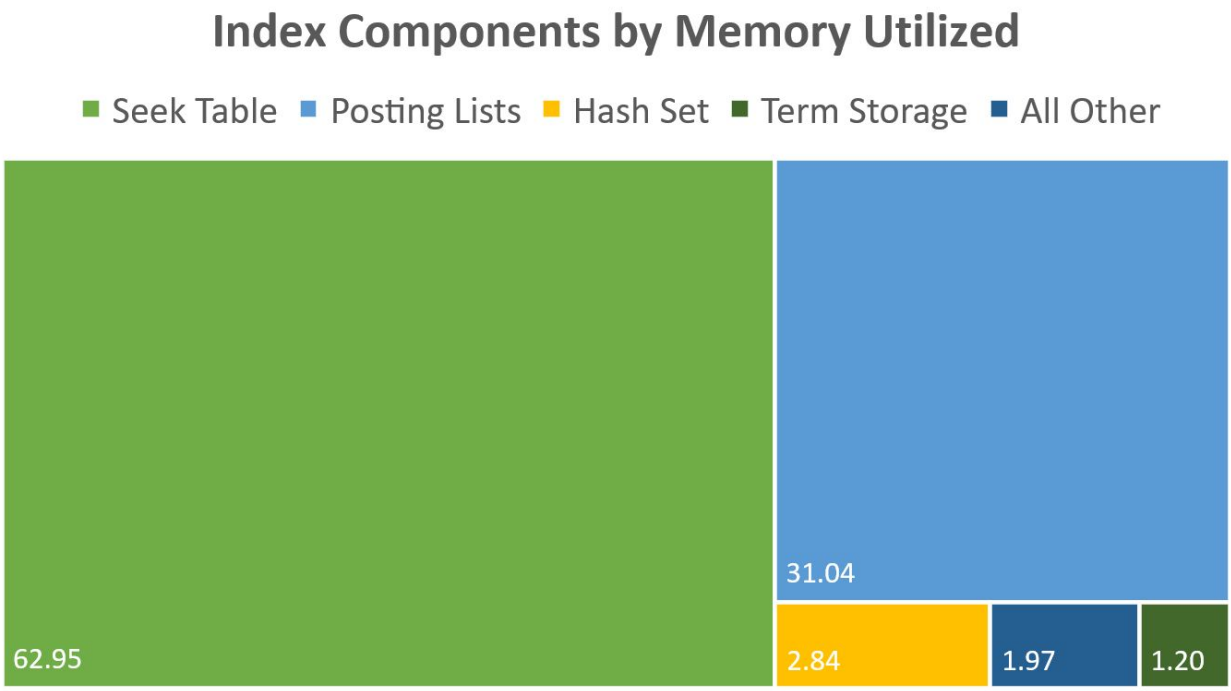

Figure 6: Memory utilization (as a percentage) by entry type in our index

Clearly seek tables have by far the most memory allocation and require the most work in terms of further optimizations. The posting lists themselves are stored extremely efficiently. Each post location is serialized as a delta from the previous post location, which is encoded using Variable Length Quantity, commonly used in the MIDI file format. VLQ encodes integer byte sequences in base-128, leaving off leading zero bits and using the high bit of each byte to denote the end of an integer. This results in our posting lists being a small portion of our chunk size (~31%), and helps to reduce paging.

The remaining two segments of the chunk are the document index and storage. These segments are what allow us to look up a document for a given location/post, and retrieve information about a document. Document storage stores variable length fields, while the document index stores the fixed field information. Storing the document index sorted by start location lets us binary search for lookup.

Section 10.2 - Index Serialization/Deserialization

The serialization step takes in a mapping from term (string) to posting list (sequence of absolute locations in the index), and a list of documents from the build system. It then translates the data into the previously mentioned file format. This is done by iteratively building up a representation in memory using dynamically allocated containers, truncating a file large enough to store the full size, and directly writing the bytes through a memory mapping. This process could potentially be made more efficient by writing to multiple growing segment files, and then merging these into a full chunk rather than building a copy of the whole file in memory.

On the deserialization side, the prefix segment allows us to create a lightweight representation of an index chunk that only stores pointers into the different segments on disk. The shared data types between

serialization and deserialization ensure that the data is interpreted exactly as it was written, and the version number and prefix fields sanity checks the shared interpretation. Deserialization does not perform the expensive step of re-building our reverse index hashmap in memory, it only provides access to the key-value array. A higher-level chunk abstraction exists which is able to perform this conversion at construction, and still avoids bringing the term storage from the file into memory. This keeps the cost of reading a chunk as low as possible. All other information is stored directly in the file, and does not require copies or writes while reading the index chunk.

Section 10.3 - Index Build System

The index build system itself is a producer-consumer system. It takes in the aggregate (.agg) files from disk and parses them into a document structure with key information stored in fields. These structures are produced by a thread pool while the vector of generated documents is not full. Meanwhile, a separate thread consumes these documents and writes them into our three index chunks: the good chunk, the bad chunk, and the anchor digests chunk. Our index is partitioned to improve query serve by separating documents by static rank into a good index and bad index. When a given chunk reaches a predefined max capacity, it is serialized to disk. This entire process is summarized in the **Figure 7** below.

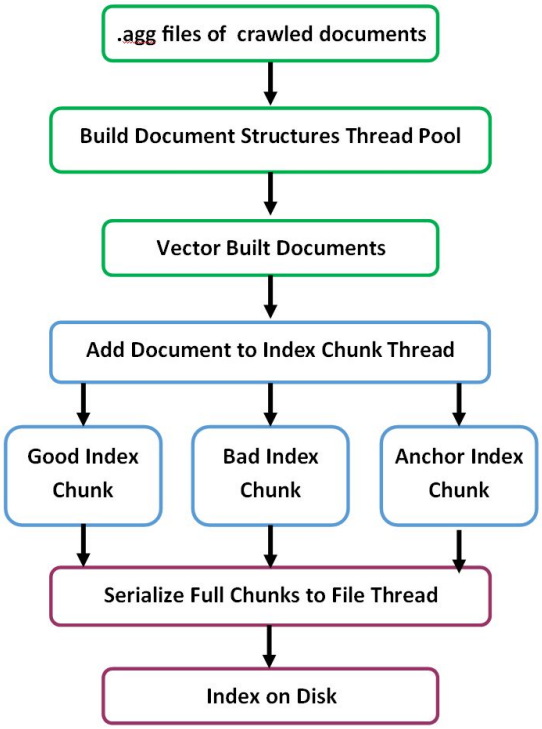

Figure 7: The index build process. This figure summarizes the process for computer aggregated crawled documents files to index chunks.

Section 10.4 - Index Build Distribution

Because index chunks can be built entirely in parallel from separate .agg files of crawled pages, we could trivially distribute index build. Our index build was distributed across three separate machines, substantially increasing the index build rate; in a production search engine, this could be extended even further. Moreover, because all files were available on Google Drive, physical data transfer on disk was unnecessary. Instead, each index build machine could pull the necessary files to index directly from Google Drive.

Section 10.5 - Index Performance

Our multithreaded index build supported a build rate of 15 million documents a day per machine. We completed index build on a network of 3 machines over the course of 5 days. Additionally, our index serialization yielded high compression rates for our index; our crawl archive was compressed by a factor of 3x when building our index. This is shown in **Figure 8** below.

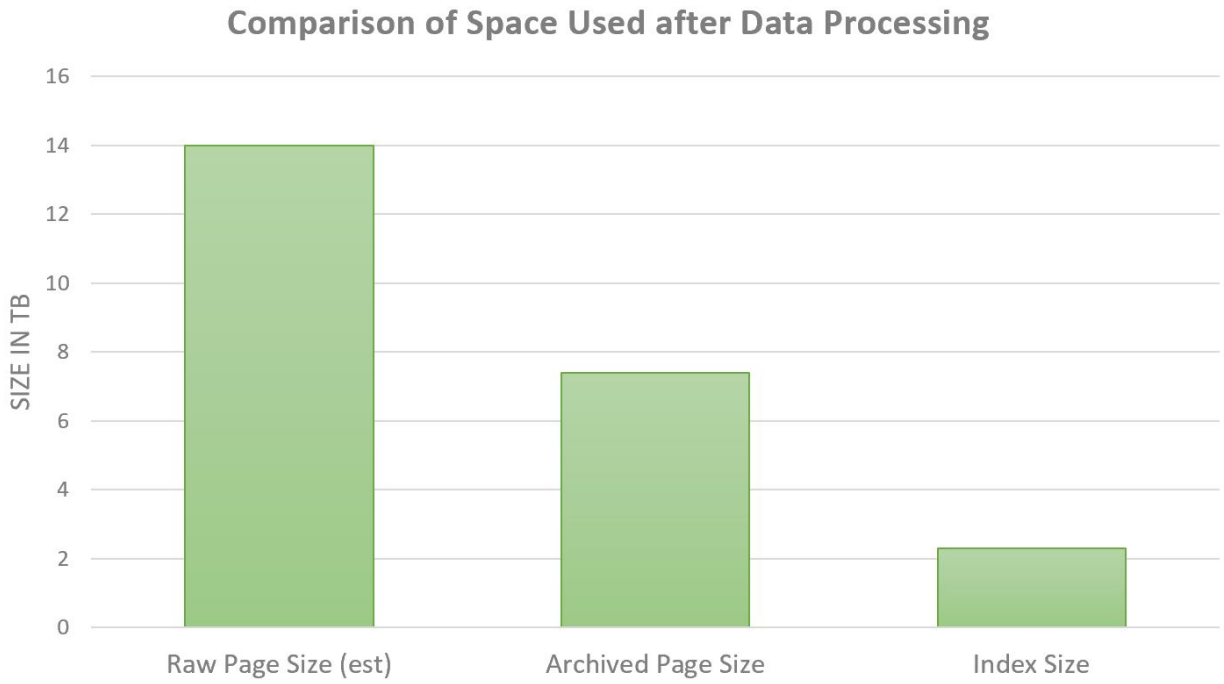

Figure 8: Total size of crawled data compared against archived page size and index size

Here we see the high level of compression from raw HTML to the index itself. The HTML parser removed ~ ½ the content before archiving the page. The index then further compressed this data by factor of 3. This allowed us to convert an approximately 14 TB of raw crawled pages into a 2.2 TB index. This was critical to search engine performance as there was substantially less data on disk to traverse during query serve.

Section 11- Query Parser

The query parser translates a search into tokens bound by a relationship given by a specific grammar parse tree. It is implemented as a top-down recursive descent parser, a style of parser known for its modularity and elegant simplicity. The query parser that we implemented in our project is a combination of Google query language and the language specified by Nicole Hamilton. It supports all recommended language features as well as numeric ranges (denoted by ##..##), stop word elimination, stemming, string cleaning, and intersection (ANDing) of adjacent words.

Section 12 - Constraint Solver

The constraint solver takes the results from the query parser and uses logical stream readers to return pages from the index that contain the keywords. An example would be the search *the University of Michigan*. The Query Parser would return the ISR parse tree [(univers OR univers*) AND (michig OR michig*)] (note stemming, casing, stopword elimination, and decoration to search for anchor text), and the constraint solver would traverse the index with this ISR tree finding all matches.

Section 12.1 - Constraint Solver Design

The constraint solver is implemented as a multithreaded process, which handles several chunks in parallel. It clones the ISR tree from the parser into an arbitrary number of copies, runs each across a subset of index chunks, and combines the results. Since our index is partitioned, we initially only solve over the “good index”. If we do not generate at least 1000 matches on the good index (on each distributed machine), we then go to the index of remaining pages to generate additional matches. Since the “good index” only consists of 20% of total pages this allowed us to increase performance by 5x on many common queries.

The constraint solver takes an index (a vector of memory mapped chunks) by reference from the high level query serve process. Then, in separate threads, it scans the ISR trees over the chunks in the index and merges the results from each thread together into a final set of matches. The constraint solver takes care to consolidate anchor text matches for a single URL into a single object used by the ranker. These matches are then returned to the query serve engine, which passes them to the ranker.

Section 12.1.1 Thread Pool Design

For the constraint solver, we experimented with thread pools of varying size, ranging from two threads total to one thread per index chunk. We saw no improvement by using one thread per index chunk, largely because disk I/O was the limiting factor. In the end, we settled on a thread pool of ten threads total, which maximized performance in testing.

Section 12.2 - ISR System Design

Index Stream Readers (ISRs) are the core of the constraint solver because they find the desired tokens or a combination of child ISR matches in the index. Our system uses Word, Intersection (AND), Union (OR), Not, and Phrase ISRs. These ISRs are derived from a base ISR class; this fact allowed for the

parser-generated ISR tree to take advantage of a polymorphic interface of token retrieval. ISRs interfaced with the index to access posting lists and document metadata. The interface, *index-serve*, defined iterators to move around that index so that ISRs were not dependent on index implementation. This design decision made ISRs robust to changes in index content and also allowed for parallel development.

Section 12.3 - Constraint Solver Performance

Initially, page table lookups in the operating system were the bottleneck for the entire query serve engine. Because our loaded index was a 2.2 TB memory mapped file, the entire index was never paged into physical memory; not even the index hashmaps could fit entirely in memory. This lead to continuous thrashing and horrible performance. The operating system would thus have to go to disk, locate the appropriate pages that matched virtual memory and load all the necessary 4K pages into RAM. This load process took over 99% of total constraint solver time. We observed this behavior by the fact that repeated queries (even those that occurred several hundred queries and over an hour later) were served in 1/100th of the time. We resolved this issue by distributing query serve across several machines (documented in **Section 14.1**), each hosting a ~170 GB index with ~10 million pages. Page faults were no longer the constraining factor, and constraint solver performance was improved by a factor of 100. The constraint solver on average takes ~1000 ms to generate all matches for a given query on the entire 116 million document index.

Section 13 - Ranker

The overall purpose of the ranker is to prioritize the ordering of results shown to the user. In effect, it takes all matches from the constraint solver and develops an ordering over them specified by the query. In designing the rest of our search engine, we noted that elements of the ranker could also be quite useful for the task of prioritizing crawl order. We decided to divide the ranker into 4 components: the URL, Crawler, Static, and Dynamic Rankers, which will be explained in the coming sections. Division into modules enables reuse between webpage ranking and crawling, promotes good encapsulation, and makes tuning of individual ranker performance substantially simpler.

Each ranker has a function to extract features from an input then a function to produce a rank from these features. By convention, ranker features are all non-negative. Negative values are used to indicate that the ranker input was invalid and should not be considered. The separation of feature extraction is vital in situations such as index build where we wish to only store the dense feature.

Section 13.1 - Url Ranker

The URL Ranker gives the quality of a url. It is used as a submodule of the Static and Crawl rankers.

First, we implemented the obvious features of protocol and extension ranks by hand, designing values as shown in **Appendix A.** To scientifically determine what features might be useful, we scraped urls from Reddit and extracted an excessive number of features from them, detailed in **Appendix B.** Relative

importances of these features was determined using an Extra Trees Regressor (ETR) and can be seen in the chart below.

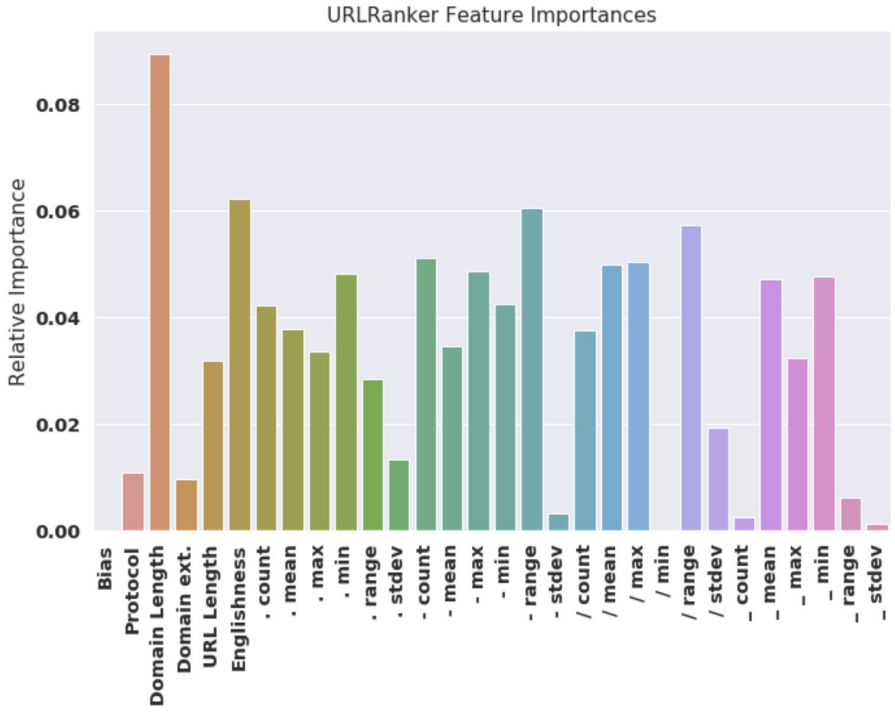

Figure 9: Extra-Trees Importance of Experimental URL Ranker Features

Clearly, the ETR determined that many features other than the leftmost "obvious" features are important. We further refined these features over the course of ranker development, eventually deciding on a final feature set that can be seen in **Figure 10**:

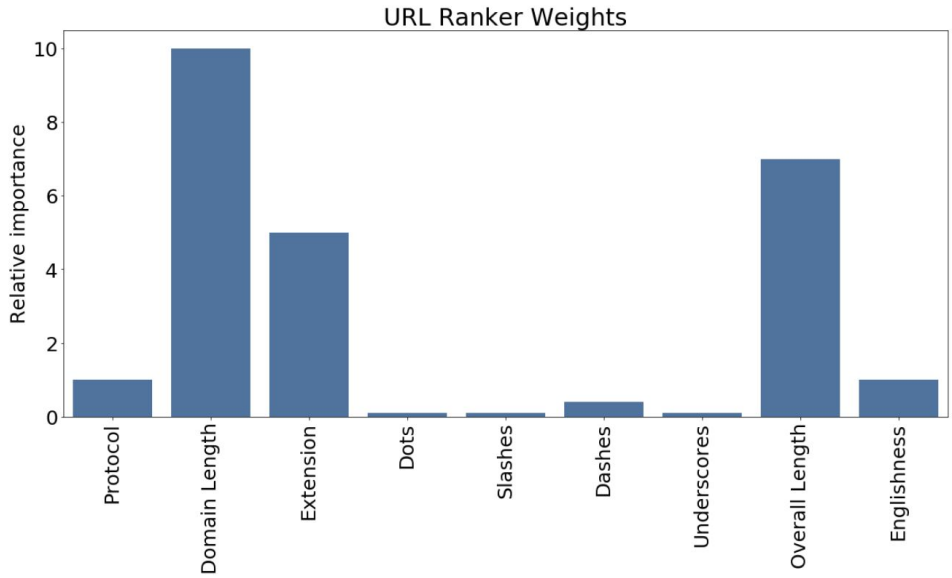

Figure 10 - Final URL Ranker Features and Weights

Initial versions of the url ranker used a direct linear combination of these features, but we eventually created hand-designed piecewise nonlinear functions, which we apply to the features before weighting. In Python, we experimented with deep learning systems to automatically and empirically learn these nonlinearities, but we were unable to deploy these models due to the time investment that would have been required to implement the necessary matrix math.

Section 13.2 - Static Ranker

The Static Ranker returns the quality of a given HTML page without respect to any query. The Static Ranker uses the URL Ranker as a submodule. Our Static Ranker makes the additional assumption that only English pages should score highly.

Most static ranker features are ratios of the lengths of certain portions of the HTML digest. These are covered in detail in **Appendix D**. We also implemented some novel features to achieve our goal of detecting non-English or spam websites.

Stopwords are not stripped from HTML digests until the index build phase, so we count their number in the Static Ranker. We use a nonlinear function of this count to detect spam pages, as pages with too few stopwords are clearly keyword spamming. Our exact list of stopwords can be found in **Appendix A**.

To improve crawl performance, we implemented Englishness detection in the Static Ranker. To do this, we count the frequency of each A-Z character and compute its difference from a pre-determined distribution using Jensen-Shannon Divergence. This allows us to detect any pages which do not use characters with approximately the correct frequency, which helps to filter out any script pages, stylesheets, or foreign language pages which make it through our other methods of filtering A graph showing effective splitting of English and non-English pages can be found below in **Figure 11**:

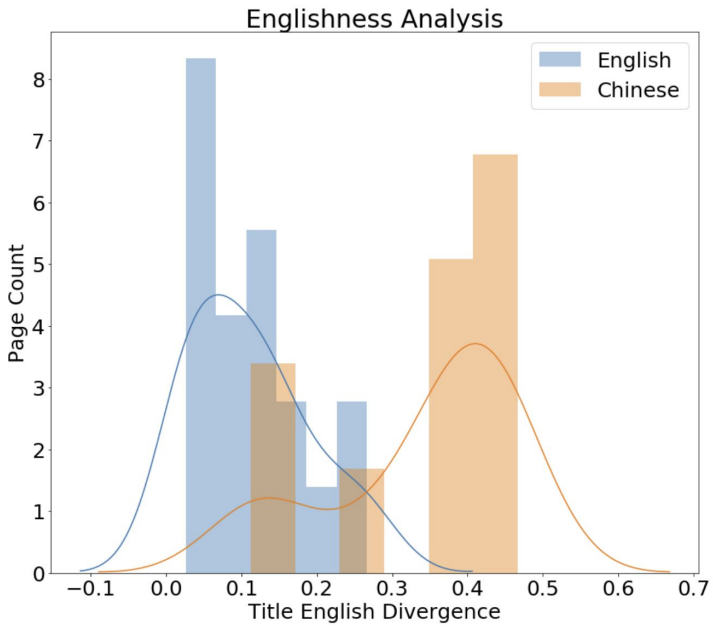

Figure 11 - Distribution of English and chinese pages

From the figure above, we see distinct curves for English and Chinese pages with little overlap, allowing effective detection and elimination of foreign pages (particularly Chinese). We also developed our static ranker weight and features which are shown in the plot below:

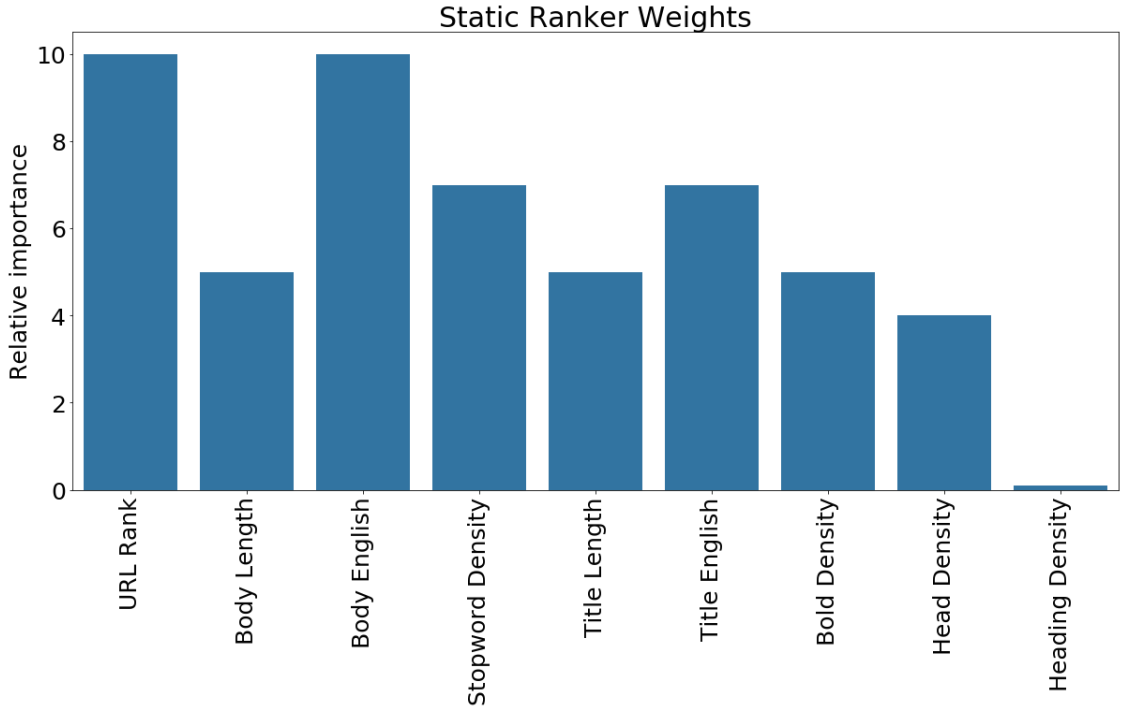

Figure 12 - Final Static Ranker Weights

In a larger system, we think the Static Ranker could be improved dramatically. All the features we have extracted are attempts to infer the quality of a page, however in a system with actual users we would be able to directly measure this by seeing what domains are most commonly clicked on the frontend. Other teams used Alexa domain rankings to emulate this, however we initially avoided this. It was eventually implemented as a Dynamic Ranker feature for reasons which will be discussed in **Section 13.4**.

Section 13.3 - Crawl Ranker

The Crawl Ranker prioritizes the order in which links are crawled by scoring them before insertion into master's frontier. Since master also periodically prunes its priority queue, the scores given by the Crawl Ranker indirectly determine which links are crawled and which are not.

We designed our Crawl Ranker to minimize the overhead of content sent over IPC. It accepts the URL as input as it is very rich in features and introduces no overhead since it is required by other parts of Master. We decided to add one additional feature to be sent over IPC - the static rank of the source page a link was found on. This feature is very dense since it encodes all the Static Rank features discussed above. We make the "PageRank assumption" that URLs found on high static rank pages will generally be higher quality than those found on low rank pages. Computing these Static Ranks on Minion means we save the cost of sending the full HTML page to master. It also naturally distributes the CPU load of computing the Static Ranks across our thousands of minion threads.

Towards the end of our project, the Crawl Ranker also took on some responsibility for politeness. We added a random noise feature with an approximately 20% weighting which served to break up clusters of URLs from the same site to prevent over crawling of any particular domains and accidentally Distributed Denial of Service attacks (DDos).

Section 13.4 - Dynamic Ranker

We decided that our dynamic ranker would operate on four decorated streams of information from the index - URL, Title, Headings and Body text. It breaks down any query into a list of stemmed tokens, then extracts the number of matches for these streams. For each stream it computes the number of OR, Phrase and Rarest Word Intersection matches. We augment these index based features with the Static Rank. This results in 12 analyzed features.

The features extracted by the Static Ranker are stored in the index as a dense vector so that we can include Static Rank at this step without access to the raw HTML.

We also include a score for the domain extracted from Alexa Top Pages. As mentioned in **Section 13.2** we initially avoided directly taking another company's website ranking, instead opting for a purely ML static ranker. At the point we added it it was too late to add it to the static ranker since its features were already baked into the built index. Thus it was added to dynamic ranking, which also has access to page urls.

Below are the weights used in the linear combination of these features.

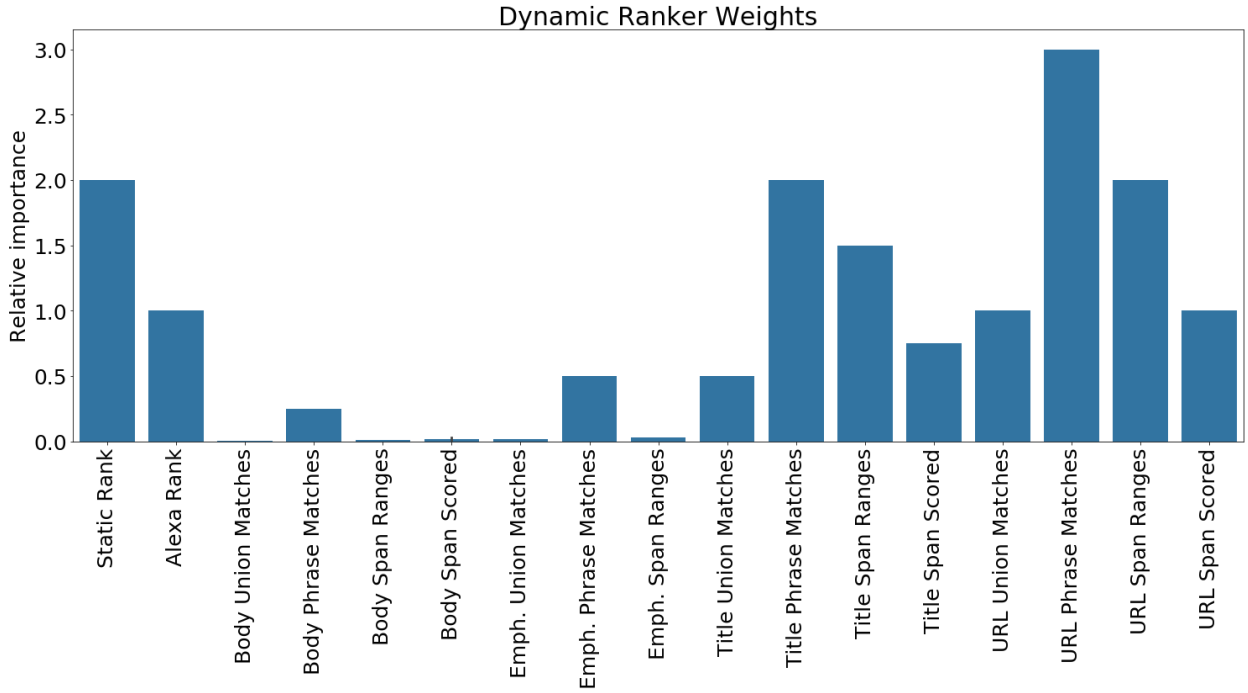

Figure 13 - Dynamic Ranker Weights

Note that body features appear very low weighted, however they often have significant contribution to rank due to the high values of the features themselves.

Section 14 - Server

The server is the final C++ module of our search engine. It interfaces with the frontend to receive user queries and then calls the query serve function itself to return ranked results back to the user. The server receives the query as JSON from the frontend. It then parses this JSON to get a query from the user and a particular page of results they want. It then calls the query serve function which returns a sorted vector ranked results. The server can then run the safe for work filter (SFW) as needed to generate final results. It generates the set of 10 results for the particular page requested, encodes it as JSON, and returns it to the frontend for display.

Section 14.1 - Distributed Serve System

The above architecture lays out how a single server system would function however to increase query serve performance we created a distributed server to work with a network of machines. Each minion machine has an index of ~1/13th of the pages we have indexed and only by aggregating the results back from all of the machines can the multi-server gain a complete set of results. We chose to implement the distributed system at the server level with a multi-server and a minion server.

Each minion machine operates a minion server. It is a simplified server that returns all results in JSON for any query received. When the multi-server receives a query it then queries all minion servers over the network with the query received. They return their vector of ranked results as JSON following the general server process listed above. The master server then aggregates all of these results, it sorts the results and then returns appropriate pages of results to the frontend.

The master server is a very lightweight process while the minion servers do all the work. The only computation involved on the master server is caching recent query results and sorting the aggregated vector of results from the minion server. By creating the distributed serve system we were able to reduce average query time on the full index by a factor of ~200 by eliminating most thrashing.

Section 15 - Frontend

An initial frontend was developed using Bootstrap and Vue.js. The frontend used in our live demo implemented with Webpack, Vue.js and Bulma. It features an “About Us” Page, Landing Page and Results Page. It was developed using a global store model but did not use libraries such as Vuex to manage this store. In **Figure 14** below is an image of the Munch home page.

Figure 14: The Maximal Munch home page.

The results page contains a variety of features for customizing and displaying search results. It features user friendly messages for cases where results returned no results. It has an interactive page selector for querying additional results in increments of 10 per page. It has a google-inspired tagline at the top of each set of search results which displays the number of results, time taken, and index size. It also has controls for the Safe For Work (SFW) mode which screens for explicit content. The SFW mode also warns users when > 20% of their results were filtered incase they would like to turn it off and see the full set of query results. In **Figure 15** below is an image of a sample query in the Maximal Munch Engine.

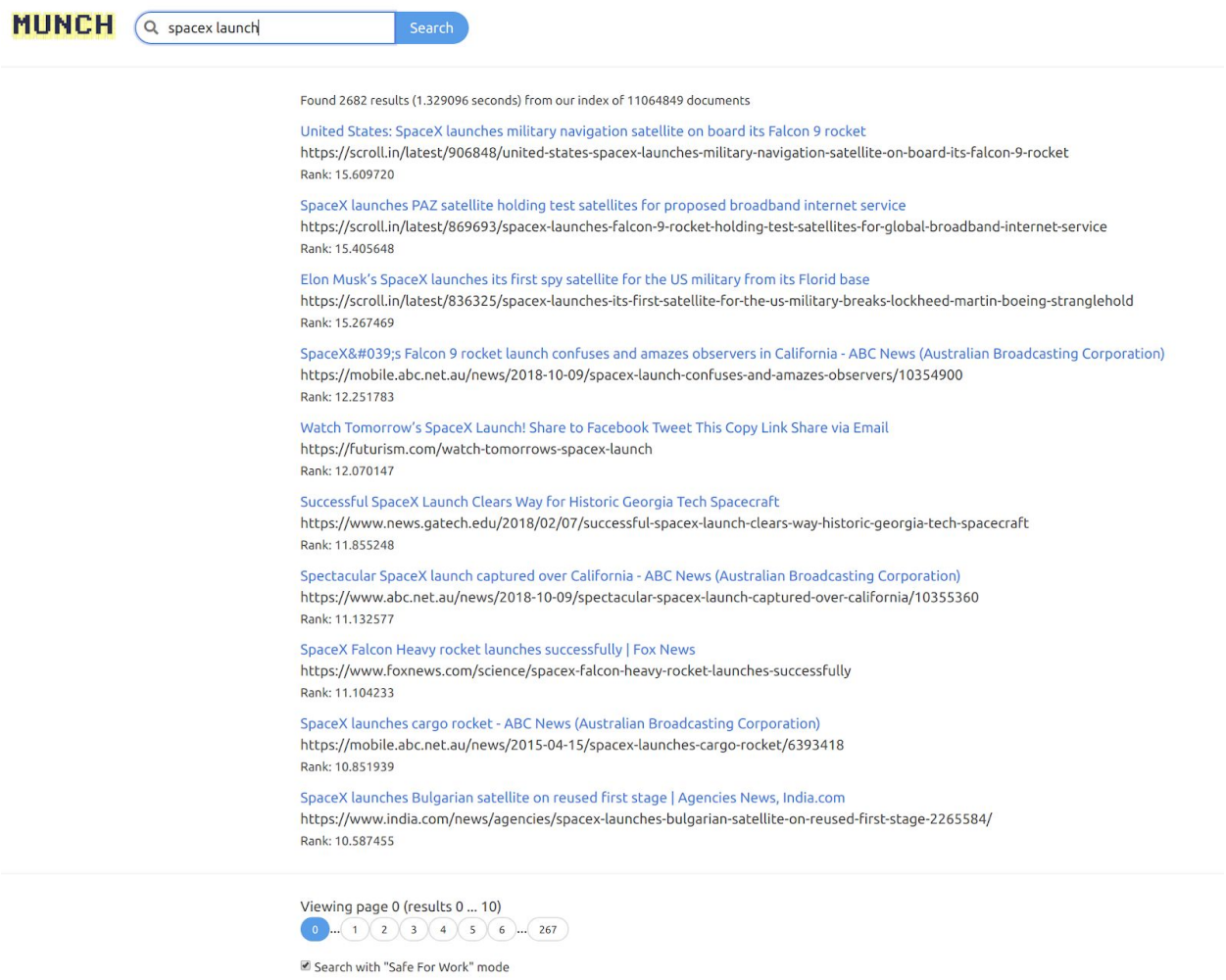

Figure 15: The results page of the Maximal Munch engine. Note the page selector and "Safe for Work" mode check box.

The frontend also validates queries to ensure they are nonempty, and prevents users from submitting new queries until their previous one returns information. This was essential to supporting a semi stable live class demo serving to multiple users as otherwise users inevitably DDoS our servers by repeatedly querying without a delay.

All of the above listed features are supported by fields in the JSON response sent by the backend. The frontend requests results from the backend by prepending its get request with a "search?" token then appending parameters. For example "server.com/search?page=eecs+398&page=0&sfw=true" requests results for "eecs 398" on page 0 with safe search enabled. Queries are "url encoded" to support this by a javascript library, then are parsed on the backend side by splitting on & characters.

Section 16 - Example Queries & Performance

While the query engine performance was summarized in in **Section 6**, we will provide a more detailed breakdown of performance here. Below is a table of sample queries run on a single machine (representative of the performance of a target minion process). This machine served an index of 11 million documents which is approximately the target size for each machine in the distributed query serve system.

Table 3: Summary of query result performance for a single machine serving 11 million pages.

Query	# of Results	Time (ms)	Time per result (ms)
"detroit tigers"	1925	54	0.03
"Nicole Hamilton"	2	61	30.50
"spacex launch"	469	76	0.16
cppreference	127	104	0.82
"blueberry pie" AND recipe	84	209	2.49
"ford pickup truck"	71	210	2.96
"alfa romeo"	4038	288	0.07
spacex	4003	331	0.08
spacex launch	2713	356	0.13
parker solar probe	284	750	2.64
"Cherry Pie"	677	900	1.33
"Michigan Football"	6583	1120	0.17
"vladimir putin"	8384	1126	0.13
"vladimir putin" AND "donald trump"	3323	1732	0.52
"university of michigan"	20,256	2947	0.15
wikipedia	142,369	13558	0.10
wikipedia linux "operating system"	2797	14,887	5.32
Average	**11653**	**2277**	**2.80**
Median	**2713**	**356**	**0.17**
1st Quartile	**284**	**209**	**0.13**
3rd Quartile	**4038**	**1126**	**2.49**

From the table above we can see a couple of insights. In our distributed system, serving ~11 million pages on each machine in the cluster seems reasonable as we have a median query time of 0.36 seconds. We can also take note of a couple of outlier results. Particularly ***wikipedia linux "operating system"*** which had a query time of ~15 seconds. This was due to the complexity of the query. The ranker currently takes substantially longer to rank long and complex queries than simple ones. From the rightmost column you can see the general trend that response time per result rises as number of searched words increases. Additionally we can see the ranker also has performance which is linear with number of matches. The ***wikipedia*** query exemplifies this performance as it took a very long time due to over 100,000 results ranked.

The next table illustrates performance on our distributed system. This was a cluster of 13 machines each running with 16 GB of RAM which served and index of 116 million documents.

Table 4: Summary of query result performance for a distributed cluster serving 116 million pages

Query	# of Results	Time (ms)	Time per result (ms)
"detroit tigers"	21267	5302	0.25
"Nicole Hamilton"	76	568	7.47
"spacex launch"	9346	1733	0.19
cppreference	1797	970	0.54
"blueberry pie" AND recipe	1048	2497	2.38
"ford pickup truck"	1099	3148	2.86
"alfa romeo"	53213	10019	0.19
spacex	41393	12069	0.29
spacex launch	34835	5481	0.16
parker solar probe	3625	4451	1.23
"Cherry Pie"	7320	3431	0.47
"Michigan Football"	11454	4332	0.38
"vladimir putin"	40497	9613	0.24
"vladimir putin" AND "donald trump"	31604	5977	0.19
"university of michigan"	189,319	32804	0.17
wikipedia	295,821	23385	0.08
wikipedia linux "operating	29325	24,078	0.82

system"			
Average	**45473**	**8815**	**1.05**
Median	**21267**	**5302**	**0.29**
1st Quartile	**3625**	**3148**	**0.19**
3rd Quartile	**40497**	**10019**	**0.82**

Here we can see that, by distributing query serve, we maintained decent performance even as our index size grew. The performance did drop by a factor of ~10. This is in part because the hardware on our distributed cluster was worse both in terms of CPU and RAM. Additionally, performance dropped because we didn't perfectly load balance our index across the distributed machines. You are forced in the distributed architecture to wait for the slowest machine to return results so this imperfect balancing substantially hurt performance. There is certainly room for improvement on these performance numbers; this can be done with a combination of code improvements and a larger cluster of distributed machines but we have successfully demonstrated a system capable of delivering a minimum viable search engine over a multi-terabyte corpus.

Section 17 - Known Bugs

Below is a list of known bugs in the search engine. These would all need to be resolved before it could be turned into a product

1. In the last 4 days and 45 million documents of crawling, Master segfaulted once of unknown origin.
2. Because the Ranker is linear in time-complexity with number of search results to rank. It struggles on queries that return a large number of matches. This less of a bug and more of a performance issue that should be resolved by speeding up the Ranker module.
3. The Not ISR is not fully implemented. It is in the codebase but has a known bug causing the engine to sometimes return zero results. As a temporary solution we have disabled it in the query language.

Section 18 - Future Work and Extensions

While we have demonstrated a fully functional search engine with a large index and efficient crawl process, there is still room for improvement and extension. Particularly, there is a list of extensions that we would need to make before delivering a production version of the Maximal Munch engine. This section summarizes key extension areas including: automatic re-crawl, shingling, index sorting, early exit and other performance enhancements such as sharding and cutting off slow machines during query serve.

The crawler needs two key enhancements before delivering a close to production variant: automatic re-crawl and shingling. The first is relatively easy: we just need to timestamp index chunks, then develop a module to pull all of the pages out of them and have the crawler recrawl from that list of urls. Shingling is more complex but necessary to provide relevant and unique search results. At the moment we frequently crawl close to duplicate pages. For example, we have crawled the same pages from Wikipedia's mobile and desktop versions. This is redundant information (wasted crawl time) and also leads to somewhat odd query results where we display both the mobile and the non-mobile variant. Shingling should resolve this, increasing the diversity of both pages crawled and pages served.

The subsequent three improvements relate to performance of query serve. Creating a distributed system has lead to a much faster query serve side; however, to support volume requests, we need to increase both throughput and decrease response time. Index sorting and ranker early exit would decrease response time. At the moment, we rank all pages found by the constraint solver. When a large number of results are found, the ranker can take several seconds. By sorting the index by quality (static rank) we could ensure that, in most cases, the first constraint solver matches would be the top ones. Then, we could implement a ranker cutoff system where, if no top 10 or top 25 results were found in the next n pages analyzed, we cut the ranker off. With a sorted index, this would significantly reduce response time without dramatically reducing. We initially implemented this method on our search engine; however, the degrade in recall was too high as our index was not well sorted. The final suggested improvement is sharding. Since we have a distributed system we could serve off of several banks of machines (each bank containing a whole index). Our master server could shard the results and send it to one of n banks of machines. This would parallelize serve across multiple queries increasing throughput. By combining all 3 of the above improvements, we believe with enough machines we could deliver an initial production search engine.

Section 19 - Reflection & What We Would Change

In retrospect, there is a clear difference in difficulty between working within your sandbox and working with the outside world. Thus, the two modules that required the most work in terms of robustness and debugging were the crawler and HTML parser because they had to interface with the rest of web standards and rules that are rarely followed. They had to handle complications such as slow servers, bad connections, malformed HTML, and early terminated connections. The query serve side was much easier in this regard, as we had complete control over all interfaces and inputs. The query serve side was by no means easy, but it was not plagued with months of unexpected bugs and crashes. Despite these challenges, our search engine was quite successful and stable; however, it took substantially more development time on the edge of our sandbox to complete than we anticipated.

However, if we rebuilt our search engine from scratch, we would make several design changes. Safe for work mode would be stored as metadata in index build. We could analyze the entire text in this situation without performance implications and, since this is static data, there is no need for it occur on the serve side. Alexa ranking would be part of static rank; this would require an index rebuild, which is why it didn't occur. We would also constraint solve and rank at the same time to take advantage of memory locality (a producer-consumer relationship). At the moment, the constraint solver runs sequentially after the ranker. That is to say the constraint solver finds all matches before forwarding them to the ranker

This is not cache-local and leads to an increased rate of page faults when compared to simultaneous constraint solving and ranking. By changing this architecture, we would significantly increase performance. Together, these modifications would provide substantial performance enhancements over the current architecture.

Section 20 - Reflection on the Course

Overall, we found the course to be a superb exercise in designing a complex system from the ground up and in developing a massively parallel and distributed system. Before the class, most of our team had close to no experience writing multithreaded or multi-machine code. This course was an important learning opportunity in that area. Additionally, we all improved in our ability to architect and write data structures from scratch because we built our own template library. The course was incredibly enjoyable and fulfilling because we were able to deliver a functional system at the end of it. It certainly met the three goals of teaching us multithreaded and distributed systems, teaching us from-scratch architecture, and allowing us to create a final product to talk about in interviews and on our resume.

We would suggest a couple of improvements for the course as a whole. First, we would recommend being more upfront with which parts of the STL are allowed and which are not. In the end, we believe we followed the instructor's intentions, which were to require us to build important data structures from scratch (hash map, heaps, sorts, etc.) while allowing us to use portions of the STL which were not part of the core search engine code. Like std::cerr for error logging, and other very basic templates. However, we think that this was not easy for every group to realize. We would also recommend getting groups started on the engine earlier. Even with our team working hard from the start, there was still a massive time crunch to deliver the final engine. Lastly, we suggest instructing everyone to randomize their crawl at the very beginning so that teams do not receive complaints on politeness. We take full responsibility for the complaints filed against us, but requiring all teams to randomize their crawl would likely reduce the odds that future teams crawl too aggressively.

Appendix A - Ranker Configuration Files

Domain Extension Ranks:

Domain Extension	Value
com	0.7
org	0.7
edu	0.85
gov	0.85
biz	0.3
io	0.8
net	0.5
default	0.35

Security Protocol Ranks:

Protocol	Value
http	0
https	1
default	-1

Stopwords: the, be, to, of, and, in, that, have i, it, for not, on with, as, at, an, so

Words considered to be "useless" if found in anchor text: here, link, this, click, go, follow, see, load, email, article, page, download

Appendix B - Experimental URL Ranker Features

0. Bias feature - fixed to 1. Acts as an offset or intercept value.
1. Protocol rank - Determined from the config table in **Appendix A**
2. Domain name length
3. Domain extension rank - Determined from the config table in **Appendix A**
4. Length of total url
5. Englishness of url - logistic cross entropy of the character distribution of the url with a predetermined english character distribution. Values indicate distance from english.
6. . count - number of '.' characters in the input
7. . sum - sum of the lengths of the '.' delimited sections of the url
8. . mean - mean of the lengths of the '.' delimited sections of the url
9. . max - length of the longest '.' delimited section of the url
10. . min - length of the shortest '.' delimited section of the url
11. . range - (max - min) of the lengths of the '.' delimited sections of the url
12. . var - variance of the lengths of the '.' delimited sections of the url
13. - count - number of '-' characters in the input
14. - sum - sum of the lengths of the '-' delimited sections of the url
15. - mean - mean of the lengths of the '-' delimited sections of the url
16. - max - length of the longest '-' delimited section of the url
17. - min - length of the shortest '-' delimited section of the url
18. - range - (max - min) of the lengths of the '-' delimited sections of the url
19. - var - variance of the lengths of the '-' delimited sections of the url
20. / count - number of '/' characters in the input
21. / sum - sum of the lengths of the '/' delimited sections of the url
22. / mean - mean of the lengths of the '/' delimited sections of the url
23. / max - length of the longest '/' delimited section of the url
24. / min - length of the shortest '/' delimited section of the url
25. / range - (max - min) of the lengths of the '/' delimited sections of the url
26. / var - variance of the lengths of the '/' delimited sections of the url
27. _ count - number of '_' characters in the input
28. _ sum - sum of the lengths of the '_' delimited sections of the url
29. _ mean - mean of the lengths of the '_' delimited sections of the url
30. _ max - length of the longest '_' delimited section of the url
31. _ min - length of the shortest '_' delimited section of the url
32. _ range - (max - min) of the lengths of the '_' delimited sections of the url
33. _ var - variance of the lengths of the '_' delimited sections of the url

Appendix C - Master/Minion Message Protocol

Each message between master and minion (except for work_request) consists of a request header followed by the message type. The header defines the necessary data to parse the message and is standardized, while the message contains a variable sized list of urls. All messages are null terminated. Minion first issues a work request to connect to master. Master sends a CRAWL_THIS message which contains a batch of urls. For each page that is crawled minion sends a PAGE_URLS message so that master can log the new urls that were found.

Request Header

WORK_REQUEST: WORK_REQUEST\0
CRAWL_THIS: CRAWL_THIS <NUM_URLS_IN_LIST>\0
PAGE_URLS: PAGE_URLS <SOURCE_PAGE> <PAGE_RANK> <NUM_URLS_IN_LIST>\0

Message Formats

CRAWL_THIS: <URL> <URL> … <URL>\0
PAGE_URLS: <URL> <URL> … <URL>\0

Appendix D - Static Ranker Features

1. Bias feature, pinned to 1
2. Body length
3. Body english divergence
4. Stopword density per body length
5. Title length
6. Title english divergency
7. Heading density
8. Length of <head> divided by body length