

EECS 440 Final Report
25 April, 2021

Tristan Kellar (tdkellar)
Senior, Computer Science

Henry Meng (hjmeng)
Junior, Computer Science

Sebastien Rivest (sarivest)
Senior - Computer Science

Joseph Schuman (jmschu)
Senior - *Computer Science*

Ian Woodfill (imw)
Senior - Computer Science

Pedro Zamora (pedroz)
Junior - Computer Science

TABLE OF CONTENTS

Approach **2**
- Project Management 2
- Decision Making 2
- Sprints 2

Engine Architecture **3**
- LOC of Major Components 3
- Diagram of Distributed System 3
- Diagram of Filesystem Footprint 4
- Diagram of Index Builder 5
- Diagram of Query Side 5

Engine Functionality **7**
- Checklist of Functionality 7
- Statistics 10
- Examples of Good/Bad Queries 10
- Known Bugs 11
  - Manager - Memory Leaks 11
  - Worker - Memory Leaks 11
  - Worker - Other 11

Future **12**
- What would we do differently? 12
  - Index Tiering 12
  - Alter our Frontier Flow 12
  - Robots Complexity 12
  - Improve Dynamic Ranking 13
- What would be the next steps for our search engine? 13

Reflections **14**
- Workload 14
- Effect of the Pandemic 14
- Improvements to the course 14
  - Pages Crawled were too highly emphasized 14

Approach

Project Management

For version control, we maintained 5 private Github repositories for individual components of the project, which belong to a GitHub Project. For the first half of the semester, we also maintained a KanBan board with individual coding goals for each team member. We attempted to emulate the professional project management that we have experienced during our assorted internships, using one week sprints with mid-sprint standups and a story-point system. We decided to drop the KanBan system because our constant communication and varying schedules made it difficult to assign the right workload to each person for a sprint.

We did not explicitly assign roles to team members, although sub-teams of one or two often took charge of components of the engine and would dictate the work on those components. For example, Joseph Schuman took charge of the index, wrote a majority of the code and generally was able to dictate which code others should write for that component. Each team member had similar responsibilities for other pieces of the engine.

Decision Making

We maintained a regular schedule of meetings, at least two a week, and made the majority of design decisions early on. There were few implementation disagreements, and we generally would discuss a decision if there was disagreement and come to a consensus.

Sprints

Our team had two main development phases throughout the semester. Both phases were composed of many sprints concentrating on implementing specific components necessary to achieve the high-level sprint goal. Our first phase focused on populating our index. This consisted of building the robust and distributed crawling, parsing, indexing, and communication mechanisms for the engine. Our second phase was focused on the query lifecycle. This consisted of building the query compiler, the constraint solver, the index stream readers, ranking, our web server, and an aesthetic frontend.

Engine Architecture

LOC of Major Components

	Team Members							
	Tristan	Pedro	Henry	Sebastien	Joseph	Ian	TOTAL	(%)
Crawler	200	50	150	150	200	400	1,150	15.55%
Parser	0	250	250	0	0	0	500	6.76%
Index Builder	0	50	70	0	680	0	800	10.82%
Index Reader	200	0	0	0	300	0	500	6.76%
Query Compiler	0	0	0	340	0	0	340	4.60%
Constraint Solver	0	0	0	5	0	0	5	0.07%
Ranker	0	50	50	0	300	100	500	6.76%
Front End	250	50	0	50	0	0	350	4.73%
Manager	200	500	200	300	0	200	1,400	18.93%
Utilities	400	200	200	50	600	400	1,850	25.02%
TOTAL	1,250	1,150	920	895	2,080	1,100	7,395	100.00%
PERCENT (%)	16.90%	15.55%	12.44%	12.10%	28.13%	14.87%	100.00%	

Diagram of Distributed System

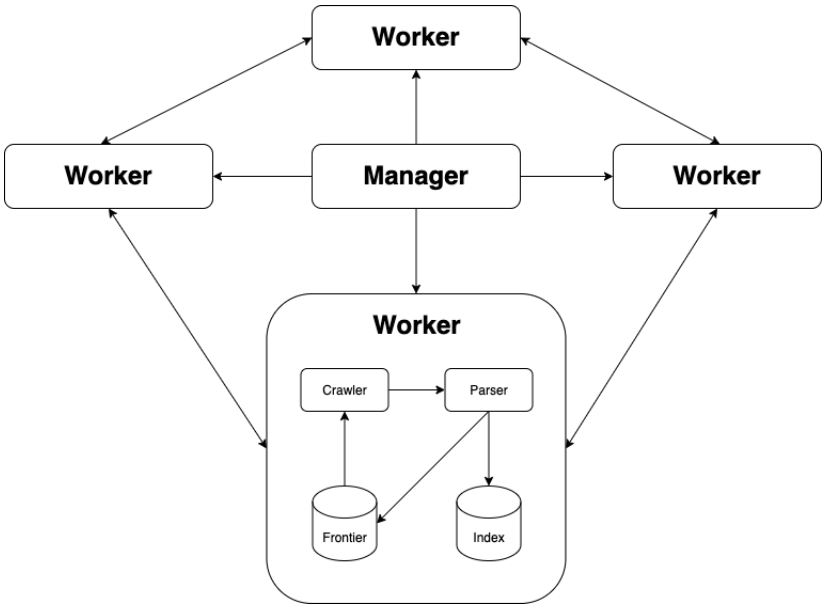
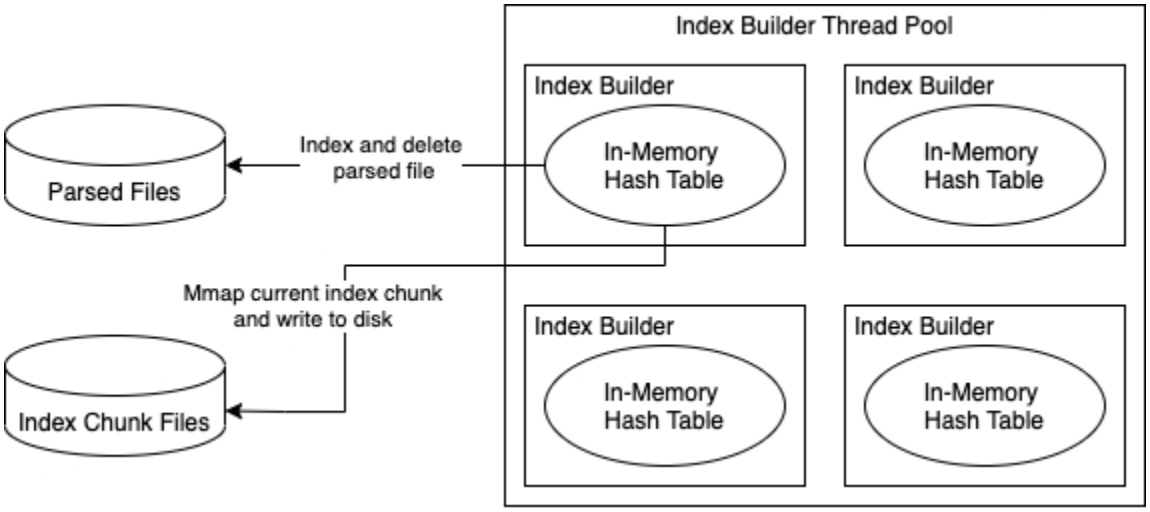


Diagram of Filesystem Footprint

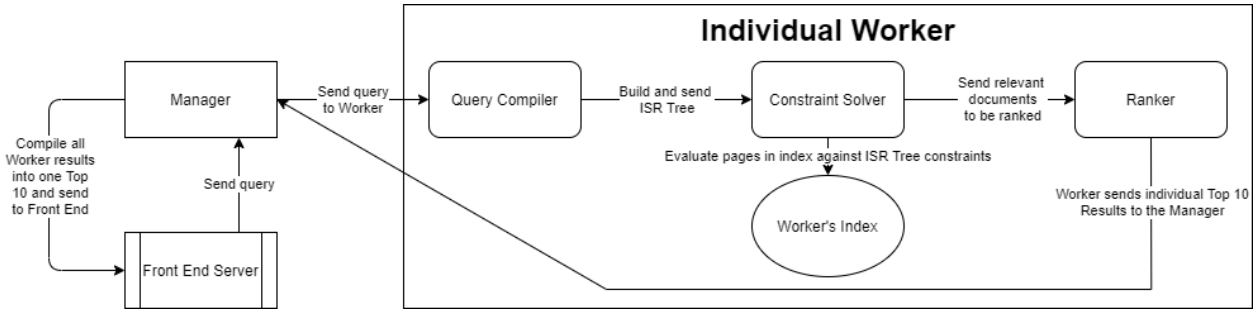
Worker	
metadataWorkerCommunicator.dat	A memory-mapped struct containing a list of the network addresses to all the other workers in the system.
metadataWorkerState.dat	A memory-mapped struct containing persistent worker state like *isCrawling*, *isIndexing, LinksParsed, PagesCrawled* and *PagesIndexed*
INDEX_CHUNKS/	Each index chunk on disk is a maximum of 150MB. It consists of a hash file as we learned about in class and wrote in the HashFile homework.
ROBOT_CHUNKS/	This is the folder where cached Robot rules are stored. The chunks are 16kb each. It contains a HashFile like we learned in the homework, as well as a list of hostnames that are contained in the chunk.
URL_POOL_CHUNKS/	This is the underlying filesystem representation of our frontier. Each URL Pool chunk is a 40KB struct that consists of a large buffer for storing URLs in the frontier. The number of URLs in each chunk is stored at the front of the chunk, as well as a list of the start indices of each URL.
PARSED_HTML_FILES/	When we crawl and parse a URL, the title words and text are printed to a new file in this directory, waiting to be consumed by the IndexBuilder.
Manager	
metadataManagerCommunicator.dat	A memory-mapped struct containing a list of the network addresses to all of the workers in the system.

Diagram of Index Builder



An IndexBuilder class instance is responsible for consuming parsed files written to a directory from our crawler threads, adding the content to an in-memory hashtable, and finally writing the in-memory hashtable to disk once it surpasses a threshold size. Each worker is able to maintain a variable sized thread pool of IndexBuilders to cope with varying load.

Diagram of Query Side



The Front End sends the query message from the user to the Manager, which initiates the query side processes. The Manager will send the query to the Query Compilers in all of its workers, where an ISR Tree is built. Documents in each Worker's Index are evaluated against that ISR Tree and sent to the Ranker if they match. The Ranker creates a Top 10 and passes it to the Manager in a JSON payload. Once all Workers have responded to the Manager, the best results from all of the Top 10s are sent to the Front End to be displayed for the user.

Engine Functionality

Checklist of Functionality

Level 1: Parse HTML files	
Parse an HTML document to extract words and links.	X
Hash function needed to build the index picked.	X

Level 2: Build a crawler	
Retrieve documents by HTTP and HTTPS.	X
Manage a frontier of URLs to be explored.	X
Crawler is restartable.	X
URLs on the frontier are prioritized.	X
Uses a seed list. (Describe source.)	X (source: https://majestic.com/reports/majestic-million)
Polite. Obeys robots.txt and does not DOS anyone.	X
Deals with redirects and common errors.	X
Parallelized by multithreading.	X
Able to identify and suppress loops.	X
Crawler recovers automatically from crashes.	X

Level 3: Reverse word index	
File format, strategy for merging index files defined.	X
Index structured for size, performance and usefulness.	X
Decide how to number locations across documents. Accumulated useful statistics. (List.)	X
Decide what attributes are captured. (List.)	X (title words, urls, body text)
Demonstrate that you can build a reverse word index as a file with a dictionary and a posting list for each token.	X
Uses delta encoding and a synchronization table or index to support seeking.	X
Create an index stream reader class that can seek. Derive word and document ISRs.	X

Level 4: Create a user interface	
Create a simple command line interface.	X
Create a simple HTTP server as a wrapper UI.	X
Report title, clickable URL.	½ X

Level 5: Constraint solver and query parser	
Create a derived AND ISR.	X
Demonstrate ability to find next matching document satisfying an AND.	X
Derive and demonstrate OR and phrase ISRs.	X
Derive and demonstrate container ISRs.	X
Compile a query into a structure of index stream readers.	X
Demonstrate TDRD parsing of a query string with syntax allowing AND, OR, NOT, phrases and parentheses.	X
Supports stemming and stop words.	
Demonstrate compiling and running a query producing unranked results.	X

Level 6: Build a ranker	
Rank using a bag-of-words technique.	
Rank using static page attributes. (List.)	X (Query words in URL, URL domain length, top level domain like .edu, HTTPS vs. HTTP, Prioritize fewer slashes in URL)
Rank using heuristics or other method considering proximity or ordering. (Describe.)	X (Spanning)
Demonstrate ability to produce a useful 10 best search results.	X

Level 7: Advanced features	
Crawls from multiple machines without overlapping. (Describe.)	x (hashes by domain to assign each URL to a unique machine)
Indexes on multiple machines.	x (hashes by domain to assign each URL to a unique machine)
Queries parallelized across multiple machines. (Describe.)	X (Each machine has its own small portion of the index to itself. Each machine searches and ranks its own index and generates a top 10. The manager then chooses the best results from each machine's top 10 to form its own final top 10.)
Words in the URL are indexed.	X
Matches in the URL considered in ranking.	X
Ranking tuned with a training set.	
Ranking uses a neural net or other ML technique.	
Create a snippet to go with a reported match.	X
Indexes and searches anchor text.	
8Supports PageRank or similar.	
Identifies and suppresses spam, porn, non-English content. (Describe.)	X (does not parse/index words starting with non-english characters)
Uses shingles or other technique to suppress duplicates.	
Long-running queries terminated. (Describe.)	X (cap off number of matches per machine)
Does automatic recrawl.	

Statistics

Pages Crawled	228.4 Million Pages
Peak Crawling Rate	~3,800 Pages per Second
Pages Indexed	144.1 Million Pages
Peak Indexing Rate	~11,000 Pages per Second

Folder	**Average Size Per Machine**
INDEX_CHUNKS/	2.25 GB
URL_POOL_CHUNKS/	6.43 GB
ROBOT_CHUNKS/	2.25 KB
PARSED_FILES/ (remaining after indexing)	33 GB
Worker Total Footprint	43.45 GB

Worker		**Manager**
48 Worker Machines		1 Manager Machine
Google Compute Engine		AWS EC2
(12 x 4) e2-standard-2 (2 vCPU, 8GB RAM)		(1 x 1) t2.xlarge (4 vCPU, 16GB RAM)

Examples of Good/Bad Queries

Our large Index size began to pose a problem when it came to finding good results for queries. Our initial build of the constraint solver and ranker put no limit on the number of possible pages it would search through and rank to get results, and we found this approach to take multiple minutes to process queries. We had to modify our ranker to only rank the first 10,000 pages before stopping and returning results in order to combat this problem. This sped up our query time to a much more acceptable number for most queries, but it did cause some of our queries to not have great results.

Since the ranker stops after looking at 10,000 pages, queries that have many possible relevant pages produce underwhelming results. Examples that we had found to be poor were queries like “wikipedia” or “apple”. While our ranker returned pages that had minor relevance, it never simply returned the front page of these websites in the top ten results. This is likely due to the fact that these queries have many relevant pages in the index, and the ranker stops before it even sees a page like “wikipedia.org”. On the contrary, the more specific queries tended to perform better. An example we tried that worked reasonably well was “General Motors”. Most of the top results were articles about the company or relating to the company. Since this query likely has many fewer relevant pages in the index that a query like “wikipedia” has, it makes sense that the ranker would actually rank most of the relevant pages before hitting the 10,000 result limit per thread. These results overall are unsurprising, as even on commercial search engines queries that are very general tend to take longer than queries that are very specific.

Known Bugs

Manager - Memory Leaks

Our Manager also would crash occasionally due to memory leaks. We ran out of time to debug our manager, and because the crashes were infrequent and the necessary manager data was persistent, we decided to manually restart on crash and not spend further time debugging it.

Worker - Memory Leaks

Our worker also contained memory leaks. Tristan and Ian spent days bug-hunting in the worker, but in the end we still had a number of issues with our worker in the real-world. Our program would crash often because of memory leaks, and we believe this is what may have caused our google cloud VMs to become completely unresponsive and need to be restarted (won’t respond to SSH, won’t accept commands from manager). To deal with our crashes, Tristan wrote a bash script to restart boxes on crash. To combat the ‘dead’ boxes (needing restart), Pedro wrote a Python script that used GCP’s remote ssh command to restart specific boxes and run commands to start our worker without opening a terminal into that worker.

Worker - Other

Aside from our memory leaks, we have occasional bugs on Worker we did not have time to trace and didn’t cause enough interference to be engine-stopping. These stemmed

from the threading library, which we never looked further than the stack-trace, and also seg faults from our implementation of some of the stdlib data structures.

Future

What would we do differently?

Index Tiering

One of the main challenges our team faced was coping with a large Index size. When searching for simple queries like “wikipedia” or “reddit”, although we were able to generate results from “wikipedia” or “reddit” we rarely were able to present the base site in the results. In our presentation we asked Professor Hamilton how Google, Bing, or any other legitimate search engine is able to consistently search their massive indexes for the best sites. Appart from a more distributed system, one of the main implementation ideas she described was “tiering” our Index, keeping high quality, base sites in a primary, smaller index, and the many more niche, lower quality sites in a secondary Index.

Alter our Frontier Flow

After running our crawl for a few days, our team realized we made a mistake in the order of operations within the Frontier. The problem, though not detrimental, quickly became our bottleneck. For each link we found on a crawled page, we check its Robots.txt page to determine whether to add it to our frontier or not. This resulted in hundreds (or thousands) of GET requests per page crawled. Because we were unable to crawl every link added to our Frontier, the time spent checking the Robots.txt file for crawled links was time wasted. With more time, our team would change our implementation to adding links to our frontier without checking its Robots.txt until it is being processed by a crawler thread.

Robots Complexity

We did not expect the Robots component of our crawler to be as complicated as it was. Evidently, this was mostly self-inflicted. We decided to create a Prefix-Trie data structure that would allow for efficient searching and storage, but this only slowed us down and wasted an entire week of our valuable time. Successfully implementing the structure proved to be quite difficult. We quickly realized that this was an unnecessarily robust solution to this problem, and after some frustration, changed the implementation back to a more simple linear path search. With more time, our team would circle back and

overcome the implementation challenges with Prefix-Trie and improve the efficiency of our Robots.txt parsing.

Improve Dynamic Ranking

Our team implemented Spanning as our main dynamic ranking algorithm. We were disappointed by both its speed and contribution. With more time, our team would optimize our spanning implementation, implement doubles and triples, and continue to tweak the weights of each of the associated factors.

What would be the next steps for our search engine?

Given more time, we obviously would continue to improve dynamic ranking, as referenced in the previous section. Furthermore, we definitely would like to rework how the Robots Parser works. First, we noticed that almost every access to the Robots data structure was searching the same domain multiple times (ie. wikipedia wikipedia wikipedia reddit reddit reddit youtube youtube youtube). So, it was obvious to us that we should add a cache that stores the previous lookup to Robots. Lastly, as discussed previously, we also need to rework how Robots is accessed by the crawler.

Reflections

Workload

Overall, the workload of the class is amongst the highest in the CS department. However, the same can be said for the reward and experience gained from this class. *This class is one of the few (only) opportunities for students to accurately experience how a long-term, project-based team would operate in industry.*

Almost all aspects of the project were about as difficult as we expected, with exception to the Robots Parser (for reasons stated in the previous two pages), and maintaining the distributed machines while crawling (we ended up creating a macro script to do this for us in python).

Effect of the Pandemic

The majority of our team worked in a remote capacity for internships in the summer prior to this school year. We have each needed to build new habits to adjust for the changing work atmosphere. Our team is very proud about our high quality project management and effective communication habits. Slack, Discord, and GitHub were

each highly effective tools we used to organize our project. In addition, we made efforts to break away from a “strictly business” mindset and were able to really get to know each other. That went a long way in making the project work more enjoyable and fulfilling. One of our team members, Henry, worked from Seattle this semester while the rest of the team was in Ann Arbor. Being sensitive to the different time zones was an important consideration for our team.

Improvements to the course

Pages Crawled were too highly emphasized

This semester’s first lecture had various statistics about previous engines built in for the class. The most “shiny” of these statistics was evidently pages crawled / indexed. Some of the best teams in the prior semesters were able to crawl a few hundred million pages and it was our team’s goal to crawl more than that. In retrospect, building a large index while it sounds cool, can be expensive and even detrimental if your engine cannot efficiently search and effectively rank your data. To improve the class for future students, we recommend turning away from an emphasis on pages crawled and towards producing high quality results. This will allow teams to better align their focus with a more satisfying end goal.