EECS 398 Final Report

Anirrudh Agarwal, Chandler Meyers, Jaeyoon Kim, Jin Soo Ihm, Grant Reszczyk, Sharon Ye

December 14, 2019

1 Group Introduction

Name	Major(s)	EECS Classes	Goals
Aniruddh Agarwal	CS	281/381/388/482/489	Work in Software
Chandler Meyers	Math & CS	281/388/477/574	Work in Software
Jaeyoon Kim	Math, Physics, & CS	281/381/482/575	Grad School
Jinsoo Ihm	CS	281/442/445/475/477/490	Grad School
Grant Reszczyk	CS	281/445/482/484	Work in Software
Sharon Ye	Math & CS	281/388	Work in Software

Five of us know each other from the math department. Grant interned with Jaeyoon at Citadel.

2 Team Dynamic and Work Division

We met as a complete group nearly every week for a few hours, and 4-5 times per week towards the end. On a daily basis, it was almost certain that some subset of us saw each other (we all hang out in the same place). Most design decisions were discussed and written down on a chalkboard in East Hall.

	Aniruddh	Chandler	Grant	Jaeyoon	Jinsoo	Sharon
HTML Parser	✓			✓	✓	
Crawler	✓	✓		✓	✓	
Index			✓			✓
Constraint Solver			✓			
Query Language	✓					
Ranker		✓		✓		✓
Front End					✓	

3 Project Architecture and Statistics

3.1 Statistics

Number of pages crawled	
Number of unique words	
Number of pages indexed	
Number of bytes of source material	
Crawl speed	
Query serve time	

3.2 Crawler

3.2.1 Seed List Generation

We initially created a seed list of about 10k URLs by crawling the Wikipedia homepage. This introduced a number of issues; for instance, many of the URLs in the resultant seed list were in a foreign language, and our parser only supports English characters. In the end, we ended up using 400k URLs from a list found online of the most visited websites on the internet (and filtered out the non-English pages).

3.2.2 Parser

The parser is written in a simple text based approach. the HTML downloader gives a String object to the parser. There are custom string comparison and seek functions implemented in the parser to safely traverse through the string without a segmentation fault. Except for very few exceptions of extracting substrings, the parser reads each character in the string once only. The parser removes unnecessary contents such as tags, scripts, and styles, and only leaves out alpha numerical characters of the content that the user can see in a page. Some additional information is recorded, such as whether a word is from a title, header, or emphasized. Initially, we extracted URLs and associated anchor texts, but we decided to remove it due to other bottlenecks.

3.2.3 Frontier

Our design of the frontier aimed to obtain high concurrency, and transational push/pop from the frontier for crash safety.

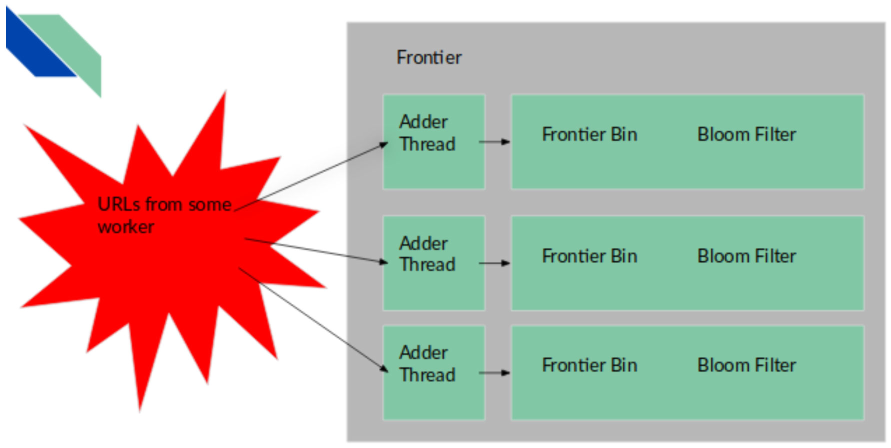
In order to increase concurrency, the frontier itself is separated into 13 bins. Each URL belongs to a frontier bin based on its hash $\mod 13$. When a new group of URLs is to be added to the frontier, each URL is hashed, and is added to a one of 13 local vectors. Once all of the URLs are added to the local vectors, these vectors are added to a queue of URLs to add corresponding to each frontier bin. We add an entire vector by r-value instead of adding one at a time to reduce the time with a mutex.

An adder thread is associated with each frontier bin. An adder thread is responsible taking URLs from the queue of URLs to add, and add them to the frontier bin. In order to reduce time with the mutex, adder thread will construct a local queue, and swap the local queue with queue of URLs to add. In order to check for potential duplicates, each frontier bin has a bloom filter (more on bloom filter later). For each url, adder threader will ensure that this url has not been seen before using the bloom filter, and then add it to the frontier. Each frontier bin is a DiskVec, a type which will be described later.

When worker requests more URLs, a socket will open up. It is assigned a frontier bin to pop URLs from. Thus 13 workers can be grabbing more URLs from the frontier at any time. In order to further increase concurrency, the function for popping URLs has to be very fast. Before locking the frontier, 4000 random numbers are generated. We will use the first 3 random numbers to sample 3 URLs from the frontier bin and obtain their maximum ranking. Assuming uniform distribution on ranking, this has expected value corresponds to 75-percentile ranking. This approximation of 75-percentile ranking is used as a cutoff for obtaining URLs. If we were to completely randomly pick URLs, we will have horrible spatial locality since we had many billions of entries on the frontier. In order to maintain politeness and increase spatial locality, we randomly choose a random region in the frontier bin, and then select 4000 URLs from the restricted region.

A bloom filter is a hashing based data structure which implements a seen set with relatively small memory usage, which was vital for maintaining a frontier at the magnitude of our project. The trade-off for this memory usage is a small probability of false positives, that is, the bloom filter will sometimes claim to have already seen a URL which we have not yet seen. However, this is a well worth it trade-off, because we do not need to guarantee that every page is crawled.

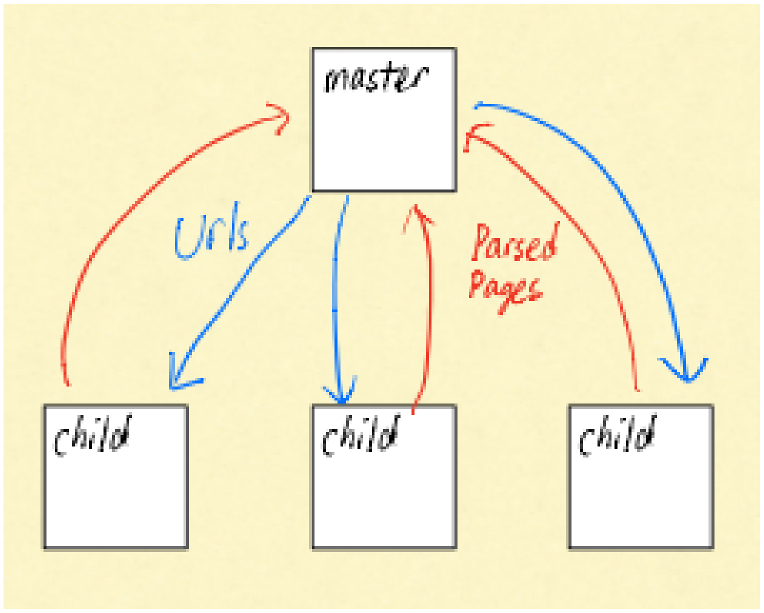
Figure 1: The frontier setup



3.2.4 Distributed Crawling

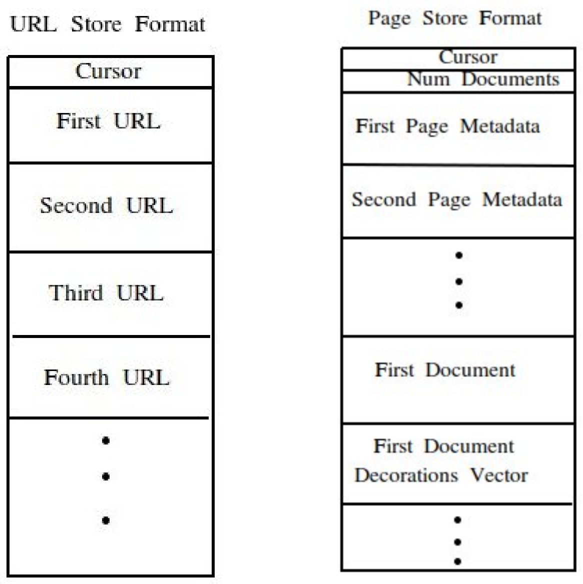
We decided to create a distributed crawling system with a central “Master” computer and many “Worker” computers. The Master computer hosted global information, including the frontier, the list of seen URLs, and associated URL info. Originally, the Master also stored a global adjacency list as well as anchor text associated to each URL, however this required us to constantly access disk with very low spatial and temporal locality. Thus these features were eventually abandoned, as they were massive bottlenecks.
We created a protocol in which Worker computers could connect to Master anonymously, and at any time. This meant we could arbitrarily connect and disconnect Workers without disturbing the Master process. At peak, we had 25 Workers connected to the Master.
Upon opening a connection to the Master, a Worker could request URLs. The Master would then take a collection of URLs from the frontier, and send them to the Worker. The Worker would then process these URLs, including downloading the pages, parsing the pages, and storing the parsed pages on its own disk. While parsing, the Worker would detect new outgoing links from the page, resolve them to absolute URLs, and send the relevant information back to the Master. The Worker can now request URLs again. If the master received any invalid TCP message, it will close the connection. This further increased the robustness of the system. Any Workers that had its sockets closed, has the responsibility to re-establish connection with Master.

Figure 2: The distributed crawling model



3.2.5 Storage and File Formats

We created a class called DiskVec, which is an abstraction of a memory-mapped file which stores an array of data. It uses an atomic cursor to keep location in the file, which allowed multiple threads to concurrently write safely to the file. One example instance of a DiskVec was the URLStore file. This file on the Master computer stored the global list of all URLs which had been seen. Another example, on the Worker computer, was the PageStore files. The PageStore files store parsed pages, along with some relevant metadata about those pages, including word decorations and the lookup address for the associated URL.



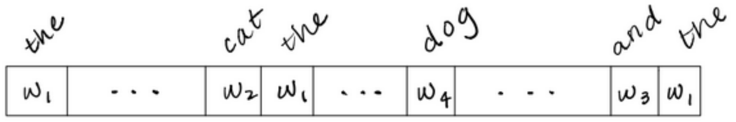
4 Index

Our index is fairly straightforward, and is very similar to the index structure described in lecture. Our index supports a variable size encoding for each post between 1 and 8 bytes. The first two bits of each post represent data about the word that post refers to (e.g. bold, italic, header, etc.) and the remaining bits represent the numerical value of the offset to the previous occurrence of the word. Furthermore, our index supports a variable size skip table for each posting list. The size of the skips is determined by the length of the posting list, so longer posting lists will have bigger skip intervals.

5 Ranker and Other Advanced Functionality

Initially, we had wanted to do Pagerank. However, it became too inefficient to store adjacency lists while crawling, so we did not end up doing Pagerank. Instead, we decided to go with the basic TFIDF scheme described in lecture, as well as something we have dubbed "snippet rank". The idea behind snippet rank is to find the biggest cluster of the most important words (based on the query) in some given document. If the most important words in our query are clustered close together in some document, that document will be ranked higher than another document in which the words are further apart. Additionally, this computation has the bonus effect of telling us where in the document we can find a good snippet, which is sent back to the main server and displayed to the user alongside top results. Additionally, we use this time when reading the stored documents for snippets to also reconstruct the title of the page. After running these computations, we send the top 100 results back to the master computer where the URLs are stored, and the master computes a fast ranking heuristic based only on the URLs. This is added to the rank returned by TFIDF and snippet rank, and that comprises the total rank of a page.

Figure 3: The motivating image behind snippet rank



5.1 Frontend

The frontend is written with simple HTML, CSS (using Bootstrap), and Javascript. Some of the Javascript functionality is adapted from online tutorials. There is a title page, search results page, and invalid page. In the search results page, default of 10 results are shown, and if the user wishes, the user can see the next 10 results. It is also possible for the user to go to the previous 10 results. Clicking the previous button right without ever pressing the next button leads the user to the invalid page. Also, if the user searches with empty query, invalid page is shown.

6 A Fantastic Bug, and Where We Found It

When retrieving search results, we noticed that in many cases, the displayed title of the page was missing its final word. Upon further investigation, this turns out to be due to a bug in the parser. When the final word of the page's title and the closing title tag are not separated by a space, the parser fails to mark the final word as being part of the title. As a result, the title will be one word too short when it is generated during the query. Because our downloaded pages are stored without all tags, it is impossible to recover this information without re-parsing the page, and thus starting over crawling from the beginning.

7 Project Reflections

In retrospect, one of the biggest mistakes we made during the project was basing decisions on estimates obtained from erroneous assumptions. In general, it was very hard to accurately determine statistics without empirical data. For instance, we tried to estimate the size a skip table in an index file with 10k documents would need to be. After obtaining empirical data, we realized that over 80% of posting lists were less than 10 posts long, eliminating the need for a skip table in the vast majority of cases. We then had to sprint to implement a more memory efficient, variable sized skip table.

Given the opportunity to do things over again, we would have enforced deadlines for ourselves throughout the semester. The vast majority of the project was completed in the last few weeks, and it has been immensely stressful for everyone in our group. If we had more time and resources, we probably would have tried to get more advanced functionality working (e.g. Pagerank, user personalized search results).

Furthermore, we had made incorrect estimates on how long certain components will take. We had the ambition to make the greatest search engine possible. We significantly underestimated the difficulty and the time required to make build a adjacency list and mapping of anchor texts.

8 Course Feedback

1. It is our general opinion that a background of just EECS280 would not be sufficient to succeed in this course. The levels of coding experience among our group members varied a lot, and it was difficult for some of our members to keep up with more advanced topics they have never been exposed to. Furthermore, the timing of the homework assignments and labs did not help us; we feel like these basic concepts should have been introduced and drilled sooner rather than later. By the time our homeworks and labs were due, our group had already written our own code to do whatever the homework assignment was, and the homework generally just took time away from our project.

2. The lecture material seemed very redundant. A lot of slides are repeated lecture to lecture, and it seems like there is not a rigid organization to what material we needed to get through by some concrete date. We feel like classtime was not used as efficiently as it could have been; for instance, introducing a lot of material in the beginning, and then converting class to office hours towards the end of the semester would have been more beneficial.

3. Intermediate deadlines for when we needed to have things done by (e.g. a functional crawler) should have been enforced. For most of the semester, we had no idea whether we were "on track" or not, and we based our assessments entirely on asking other groups where they were.

4. The midterm exam did not seem helpful. We would have preferred the exam be based on material that we would need to know to make a successful search engine; it felt more like

memorizing trivia. In particular, we would have preferred more material on system design and C++ concepts.