CIS5550 Project Report: ByteNet

Ahmed Muharram, Matthew Kuo, Eleftherios Lazarinis, Joon Luther

December 2024

1 Introduction

ByteNet is a straightforward search engine that allows users to visit the search page, input queries, and retrieve URL results. ByteNet was developed by 4 contributors, loosely following the pipeline approach outlined in the course project description. Ahmed was resonsible for crawling. Joon was responsible for the data processing of the crawl. Matthew was responsible for information retrieval and ranking. Eleftherios provided the implementations for webserver, flame, and kvs, and was responsible for refactoring the components and creating the frontend. Roles were not strict and all team members supported each others tasks.

2 Features

2.1 Integrated Webserver, KVS, Flame

The Webserver, KVS, and Flame libraries implemented in the homeworks have been refactored to support the demanding tasks of the seach engine. For example, multithreaded versions of flatMapToPair and saveAsTable have been added to optimize performance. Batch puts and gets allow for faster reads and writes of the tables. The main reason why we are doing batches is because as the kvs tables get bigger, it will become increasingly slow to write or read values from it. These enhancements represent just a fraction of the work, as many of the smaller refactors were crucial to the performance of the system.

2.2 Crawler

The crawler runs on a c6i XL instance and follows the links from a set of seed urls to discover more web pages and store their contents in a table called pt-crawl. The main challenges with crawling are maintaining a corpus of high quality documents while ensuring a variety of pages and not crashing on broken links, unresponsive pages, and other edge cases. The crawler limits the amount of urls from the same host to prioritize overall breadth of the crawl. The crawler does not get stuck when there are issues with a particular page, such as a broken link or a long response time; these pages are stored in a different table, pt-error, and the crawl resumes with the next url on the frontier.

2.3 Processing

Bytenet uses PageRank, term frequency, and inverse document frequency to determine which urls are the most relevant to a user query. Processing tasks such as PageRank, Term Frequency, and Inverse Document Frequency are executed on separate c5 4XL instances. Each task uses the table produced by the crawler to output a table of its ranks/scores corresponding to terms and urls.

2.3.1 PageRank

PageRank is used to estimate the relative importance of web pages in a subgraph of the web based on incoming and outgoing links. The corpus contains "dangling" links, which are documents with no outgoing links, as a result of having a limited crawl size. In order to prevent these dangling links from absorbing most of the rank, their cumulative rank is distributed evenly across all links in the corpus. This aligns with the random surfer model because it mimics the way a user may switch from one page to another without following outgoing links. PageRank is computed iteratively, with each additional iteration requiring memory to store the state tables and the transfer tables. Java processes can quickly run out of memory and be killed by the OS, thus saving state tables between iterations of PageRank ensures that progress is saved even if a kvs worker runs out of memory.

2.3.2 Term Frequency

Term frequency is used to identify documents where a term from the query appears frequently, thus making the document more relevant. Documents contain words that have the same root but different suffixes. From observation, storing the same roots with different suffixes in an inverted index adds complexity with little benefit to search results. To cut down on potentially redundant entries of the table, all terms in the pages and queries are stemmed.

A stopwords list is used to further filter terms that may be very frequent in a lot of pages but unhelpful in searching. Stopwords are simply stored in a hashset to quickly determine whether a term in a page should be included in the inverted index.

A maximum number of words to index per page is introduced in order to more aggressively filter page content and obtain the most relevant terms. The intuition is that the most relevant terms to a page occur earlier on the page, near the top of the document. Setting the maximum to the first 2000 words on a page ensures that a substantial amount of terms can be captured, but processing won't spend an arbitrarily large amount of time on any particular page due to its size.

2.3.3 Inverse Document Frequency

Inverse document frequency considers the total number of documents in the corpus when placing a score on certain terms that are more niche. Like TF, IDF uses the strategies of stemming, stopwords filtering, and maximum words per page to cut down on potentially unnecessary/redundant entries in the inverted index.

2.4 Frontend and Backend

The frontend of ByteNet is hosted on a t2 medium instance and features a page with a text box for users to input queries, and displays a page of results when a query has been entered. When a search is inputted, the frontend submits a flame job that queries the backend so that results can be obtained.

The backend of ByteNet is hosted on an Amazon EC2 t2 medium instance. The backend contains tables for PageRank and a table that combines term frequency with inverse document frequency. These tables are accessed when a user inputs a query to calculate scores and identify the most relevant urls.

2.4.1 Extra Credit

The infinite scrolling feature has been implemented. The top 20 urls for a query are displayed after a search. Scrolling down submits a job to fetch the next 20 highest scoring urls.

3 Ranking

To compute scores for URLs, ByteNet utilizes a combination of term frequency, inverse document frequency, and PageRank. A bias of 0.3 is applied toward PageRank and 0.7 is applied toward the combined TF and IDF scores. This weighted approach means that the results should prioritize TF/IDF but will still be somewhat influenced by highly authoritative pages. To improve the efficiency of search operations, the precomputed scores are stored in a table.

4 Evaluation

Table 1 shows the KVS/Flame benchmark results of the following configuration: 3x t2.medium machine A: 1x KVSCoordinator, 1x FlameCoordinator machine B: 1x KVSWorker, 1x FlameWorker machine C: 1x KVSWorker, 1x FlameWorker.

Results for 'content' table:
Number of files: 356
Average file size: 403688.06 bytes
Average putRow() time: 55.09 ms
Average getRow() time: 7.75 ms
Estimated time for operations:
Estimated time for 10,000 putRow(): 550.95 s
Estimated time for 10,000 getRow(): 77.47 s
Estimated time for 100,000 putRow(): 5509.47 s
Estimated time for 100,000 getRow(): 774.74 s
Estimated time for 1,000,000 putRow(): 55094.66 s
Estimated time for 1,000,000 getRow(): 7747.42 s
Results for 'pt-content' table:
Number of files: 356
Average file size: 301561.33 bytes
Average putRow() time: 53.99 ms
Average getRow() time: 6.40 ms
Estimated time for operations:
Estimated time for 10,000 putRow(): 539.85 s
Estimated time for 10,000 getRow(): 63.99 s
Estimated time for 100,000 putRow(): 5398.53 s
Estimated time for 100,000 getRow(): 639.86 s
Estimated time for 1,000,000 putRow(): 53985.27 s
Estimated time for 1,000,000 getRow(): 6398.58 s

Table 1: Benchmark Results

5 Lessons Learned

5.1 Challenges

Creating the ByteNet search engine came with many challenges, alluded to in the previous section.

1. Choosing the right instance for a task: Determining the most suitable instances for tasks like crawling, processing, and hosting the frontend involved some trial and error.

2. Processing task failures: Handling a large corpus from the crawl often consumed significant memory and CPU resources. This sometimes led to workers being terminated by the operating system, causing entire jobs to fail.

3. Calculating scores efficiently: The initial strategy involved maintaining three separate tables for TF, IDF, and PageRank scores. However, this approach was inefficient because performing Flame operations to hash into each table immediately after a search was very time consuming. Instead, the updated approach combines these three metrics into a single structure, enabling a streamlined lookup process after a query is entered.

5.2 Future Improvements

The following improvements can be made for more accurate and efficient search..

1. Phrase search: Implementing phrase search would allow users to find results containing exact sequences of words, improving the precision of search results. Instead of treating a query as a collection of individual terms, the system would match documents containing the exact phrase in the specified order.

2. Store tables efficiently: when a search is entered, scores are read from persistent tables storing combined tf/idf/pagerank scores. The pagerank table is relatively small compared to the other metrics, so it is possible to store it in memory for faster lookup.

3. Normalize term frequency: We didn't add the Eucledian normalization for our TF. This feature would have allowed us to have a fair comparison between documents of varying lengths and so the search engine will do a better job.

4. Combine results without a Flame job: Flame jobs are sometimes slow especially for getting the url scores based on the queries. The professor suggested that we should've used kvs operations to do this part of the task instead of using flame jobs.

5. Depth of the crawled urls: To ensure breadth of our crawled pages, we put many different types of urls in our seed urls list. Thus, we have a pretty wide breadth of the types of urls we crawled. However, one thing we could have improved is to increase the depth of particular topics so we will have more relevant results.