Assignment 4 - Information Retrieval

Luiz Philippe Pereira Amaral

[1]Departamento de Ciência da Computação - Universidade Federal de Minas Gerais (UFMG)

Abstract. *This project is an indexer for web pages that can be used as a tool for information retrieval in large collections of documents. The program discussed on this document is built so it can work with collections that do not necessarily fit in primary memory.*
Keywords: *C++, Chilkat, web crawler, indexer, information retrieval*

1. Implementation

The code is written in C++ 17 and utilizes C++'s Standard Template Library (STL) in addition to the Chilkat, Niels Lohmann's JSON for modern C++ and Gumbo parser libraries. While Gumbo parser and Niels Lohmann's JSON library is included in source, instructions for installing Chilkat can be found at `https://www.chilkatsoft.com/downloads_CPP.asp`. The complete source code can be found at github.com/LuizPPA/web-crawler.

1.1. Collection Reading

The program works with collections that are formatted as JSON objects separated by line breaks. Each line represents a document, and each object contains two keys, *url* and *html_content*, representing the document link in the web and content respectively, as shown in the example below.

```
{"url": "www.example.com", "html_content": "<html>...</html>"}
```

Once the documents are organized properly, the indexer reads the collection file one line at a time. For each file, it uses Niels Lohmann's JSON library to parse the JSON content and uses Gumbo Parser to extract the plain text from the HTML content. This is done sequentially from document *1* to *n*.

1.2. Indexing Terms

The indexing steps is done in batches, each batch of documents generates an index file that is later going to be merged to the other indexes. A batch is a group of documents with a maximum size defined through a constant value on the code (for the results presented here, the maximum batch size used was 4096). During this step, the index for every document on the batch is built and loaded to main memory, therefore, a larger batch size results in a larger memory requirement, here the process for building the indexes is much like the previous assessment of this assignment which can be found at *github.com/LuizPPA/web-crawler/docs/Information_Retrieval_Assignment_3.pdf*. After processing each batch, it's index is stored in a temporary file on the secondary memory.

1.3. Merging Indexes

Once every document has been processed, we need to merge the obtained indexes. To do so, we will use the fact that every index is ordered within itself, thus, the merge strategy becomes simple. On the following paragraph, we will refer to the files where the indexes were saved simply as indexes, and to the act of loading an index entry to the memory simply as reading it.

We will begin by reading the first entry of each index. If two or more of the read index entries refer to the same term, we must merge those entries by inserting into the first entry all the occurrences present on the second. Then we store those entries as a tuple $(indexes, entry)$ where $indexes$ is an array of identifiers to the indexes used to built that entry.

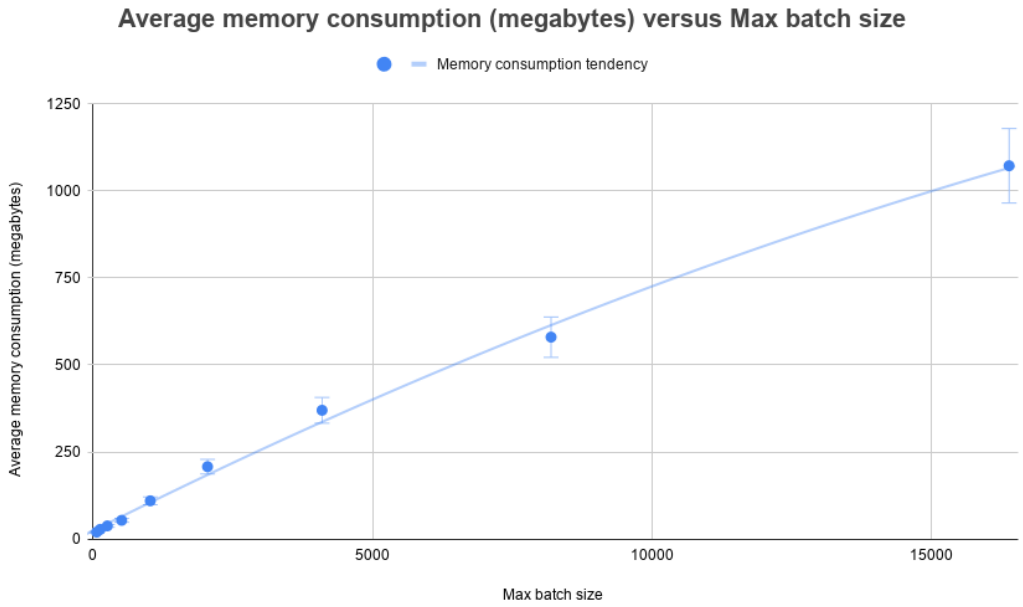
Since all the indexes are already ordered, on each iteration we can simply order the read entries, select the first among them and save it to the merged index. Then we read one more entry from each index present on the indexes list of the tuple we just consumed and add that entry to our structure. Finally, we remove the consumed entry from the structure and continue to the next iteration. We finish the execution once every entry from every index has been consumed and added to the merged index.

1.4. Compiling and running

You can use make to build the project by simply typing ”sudo make install && make”on your terminal. The output is an executable file inside the build directory. You will need, however, to install the Chilkat library on your environment. Use ”make run”to execute the crawler with a predefined seed then build the index for that collection, or run ”./build/web-crawler -b [COLLECTION PATH]”replacing [COLLECTION PATH] with the path to the file containing the collection of documents to be indexed (you may also leave it blank if the collection is in a file named ”collection.jl”inside the ./output folder) to run the indexing step only.

2. Results

To test the program, we used a collection containing 1.000.068 documents. Building an index for a collection with this many documents in-memory at once would occupy dozens of gigabytes, but since we are only loading in memory the index for a partition of those documents in any given time, we do not need this much resources. On our scenario, the memory usage was never above 370 megabytes of data, however, this value may vary depending on the content of the documents present on the collection, the architecture of the machine executing the program and, mainly, the value set to the max batch size. Generally, a smaller batch size will lead to less use of memory at the cost of more use of secondary memory, this is due to the fact that while we are loading less documents to the memory at a time, we will be creating and storing more indexes on disk. The graph below shows the tendency of the relation between batch size and memory consumption.



- Maximum batch size: 4094
- Number of documents: 1.000.068 documents
- Number of indexes produced: 245
- Collection size: 87.852.357.474 bytes (approximately 81,7 GB)
- Index size: 6.217.392.492 bytes (approximately 5,79 GB)
- Size of the vocabulary: 2.738.510 terms
- Memory consumption: 370 mega bytes (approximately)
- Index building time: 20320 seconds (approximately 5 hours and 38 minutes)
- Index merging time: 2662 seconds (approximately 44 minutes)
- Execution time: 22983 seconds (approximately 6 hours and 20 minutes)

Note that building the indexes is much more time consuming than merging them, since when building, we have to parse the entire HTML content (which is much larger than the index it produces) into plain text, sanitize each word and only then save it to an index, while in the merge process, all data is already processed, all we have got to do is join it. It is also important to remember that when reading from the documents, each word appears multiple times, but when merging, each one will appear at most n times, where n the number of indexes produced.

Even though we had a very large collection, we were able to apply a an affordable strategy to building it's index, most modern ordinary personal computers have more than 370 megabytes of primary memory. We can also point that even for much larger collection, our memory consumption would still be roughly the same.