Information Retrieval – Programming Assignment 2

Alexander Thomas Mol Holmquist
alexthomasmol@gmail.com

ACM Reference Format:
Alexander Thomas Mol Holmquist. 2022. Information Retrieval – Programming Assignment 2. In *Proceedings of PA 2 (Information Retrieval).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Previously, we built the first part of a web search engine: a web crawler, capable of crawling 100,000 pages. The second part of a web search engine is the indexer. Given a set of crawled pages, the indexer produces an *inverted index*, a dictionary of inverted lists. For every word in the corpus, the inverted list contains pairs of integers for every document in the format $(docid, frequency)$. The third essential part of a web search engine is called the query processor. This part is answers user queries based on the inverted index. In this work, we present an implementation of an indexer and a query processor using classic algorithms and heuristics.

2 INDEXER

The indexer was tasked with indexing a corpus of size 3GB. This corpus contains the text of almost one million web pages. The indexer has to follow a set of constraints:

(1) Perform stopword removal and stemming.
(2) Execute in an environment with limited random access memory available.
(3) Execute across multiple threads or processes, to speed up the indexing process.

Most of the problems found implementing the indexer concerned the above constraints. Therefore, we go over each of them below.

2.1 Stopword removal and stemming

After the page text undergoes tokenization, these are the preprocessing steps we perform over the raw document text. We used predefined sets of stopwords of the Portuguese and English languages from the Python library nltk [3].

(1) If the word is one of the predefined stopwords, discard the word.
(2) If the word has less than 3 characters, discard the word.
(3) If the first character in the word is a punctuation such as ".", "," etc, discard the word.
(4) Stemize the word using the Portuguese snowball stemmer from the nltk library [4].
(5) If the word has more than 20 characters, discard all characters after the 20th.

2.2 Limited memory

This was by far the hardest constraint to satisfy, largely because in Python the programmer has less control over the program's memory state than in languages like C. To satisfy the constraint, we performed several optimizations to the program. The numbers below consider that we have 1024MB of RAM available. The *Indexer* class mentioned below is implemented in `indexer._internal.indexer.indexer.py`.

- The corpus files are read at a maximum of 131072 bytes at a time. That is reasonable upper bound, and at the same time is a multiple of the amount of bytes the warcio library [5] archive iterator reads (16384).
- We delete dead objects and force garbage collection as much as possible. See function `Indexer._run` for an example
- We write partial indexes to the temporary directory `subindexes`, and other metadata (e.g. mapping docid -> URL) to `urlmapping`.
- When the program finishes executing, we perform an external merge sort of the partial indexes, limiting the number of UTF-8 characters we read from each file to 8,388,608, or 8M. This effectively limits the number of bytes read to about 34MB. The actual implementation in `Indexer._merge_index` seems to be more complicated than necessary, but we did not want to change it to avoid breaking the program.

As will be described in 2.3, to effectively parallelize the program, we had to use multiple cores of the host machine, which in Python translates to a need for different processes. The memory of different processes is completely isolated from each other; a behavior enforced by the operating system. Therefore, to make sure the program never crosses the memory limit given, the first thing we do in the target function of each process is enforce the memory limit (see `Indexer._run`).

The indexer is divided into four steps: *streamize*, *tokenize*, *preprocess* and *produce_index*. The steps that led to memory overflow were *streamize* and *tokenize. streamize* loads page WARCIO data into volatile memory. It creates string objects and list containers, which consume memory $O(n)$ where n is the size of the document. The *tokenize* step runs over every document, tokenizing the text into words using the library function `nltk.word_tokenize`. The problem is that if a document is large, the function `nltk.word_tokenize` consumes too much memory. This lead to an overflow a few times before the final version of the algorithm was complete.

Finally, we note that at some points in the code we had introduced a call to a debug message in the logger, but the Python f-string that needed to be built before printing the log was too large, which caused the memory limit to be exceeded. We initially thought that the string would not have to be computed if the log level was larger than DEBUG, but we were wrong.

Information Retrieval, June 2022, Belo Horizonte, MG
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.3 Parallelization

We used the `concurrent.futures` module of the Python standard library to parallelize the indexer with a process pool. Initially we had tried a thread pool, but that led to no perceptible speedup. Processes cannot share a file. Therefore, we submit jobs to the process pool called `Subindex` in the code, where each `Subindex` contains a subset of the input WARCIO files, and tracks a document ID. Each process receives a specific `Subindex` to work with. This guarantees that there is no file sharing between them.

The process takes one of the files in the received `Subindex` and reads at most 131072 bytes from it, which amounts to about 50 documents in average. Then, it proceeds to *tokenize*, *preprocess*, etc, and at the end it writes the partial index to a temporary file in the directory `subindexes`. The fact that the process only works with a small piece of the corpus at a time guarantees load balancing.

We imposed the limit of 6 processes to the process pool. This was done purely to make the program work for memory restriction specified in the assignment, 1024MB. At the moment, the program scales with memory only in the number of bytes each process can read from a file at any given time. For example, if 2048MB is assigned to the process, it will read $131072 * (2048/1024)^2 = 524288$ bytes from the WARCIO files instead of 131072. We found that this way of scaling it works well in practice. Of course, it is possible to scale the number of processes used if the computer has more cores, but we did not have time to implement this functionality.

2.4 Performance analysis

We collected the running time for different corpus sizes, as can be seen in Figure 1. The algorithm seems to scale linearly with the corpus size. In fact, the indexing part of the algorithm is linear ($O(n)$), where n is the size of the corpus in bytes. For every word of every document, we perform a number of $O(1)$ operations, such as stemming, which amounts to $cO(1)$ with c constant.

The external merge algorithm implemented in `Indexer._merge_index` runs in $O(n\ log\ n)$. We start with a queue containing all of the files. Then, we pick two files, and of these two one will be deleted. Therefore, after traversing the queue, the number of files reduces in half. Hence, the number of times we traverse the list is upper bounded the minimum k such that $lceiln/2^k\rceil = 1$. It is easy to see that $k = O(log\ n)$. The total complexity of the algorithm is $k * O(n) = O(log\ n) * O(n) = O(n\ log\ n)$.

Notes:

(1) In our implementation of the external merge algorithm, when we traverse the files in the queue, we never read the same byte twice. Then, we only perform $O(1)$ operations per byte. This is necessary to ensure that the complexity of traversing the queue mentioned above really is $O(n)$.
(2) We saw no need to *parallelize* the external merge algorithm, specially because of the memory limit. The algorithm scales– reads more characters per file– as the memory limit is loosened.

2.5 Statistics

As required per assignment specification, here are some metrics collected when indexing the full corpus of 800,000 pages, which are given below. The distribution of sizes is displayed as a histogram

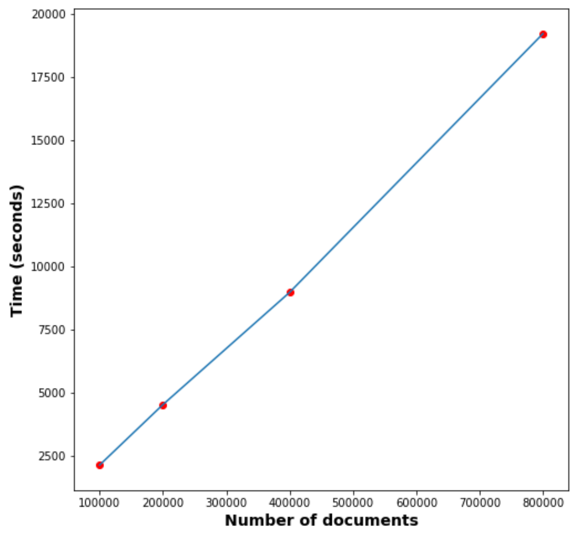

Figure 1: Execution time of indexer over corpus of size varying sizes. The maximum corpus size is 800,000 pages. Each size is twice the previous. The indexer process pool used at most 6 processes.

in Figure 2. It was necessary to display this graph in log scale due to proportions. It is clear that most of the inverted lists have small size.

- **Number of documents indexed:** 799,997
- **Number of inverted lists:** 21,773,688
- **Average size of inverted list:** 13.5
- **Average execution time (three runs):** 19,609 seconds – 5h26m49s
- **Number of tokens:** 980,360,550

3 QUERY PROCESSOR

The query processor is the second part of this work. It takes as input the index generated as described in Section 2. These are the constraints imposed upon the query processor:

(1) Stopword removal and stemming
(2) Must perform document-at-a-time (DAAT) matching
(3) Must provide two scoring functions: TFIDF and BM25
(4) Must be parallelized to run across multiple threads or processes

In the sections that follow, we first go over some important details of implementation. Then, we directly address each of the constraints above. Finally, we show a performance analysis and characterization of the ranks.

3.1 Implementation details

It is important to remark that the query processor heavily relies on statistics or metadata produced by the indexer. If the final index file

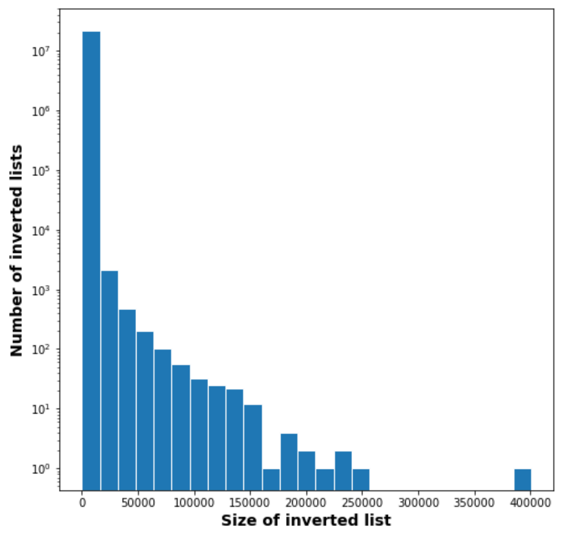

Figure 2: Distribution of number of postings per inverted list.

only contains document identifiers, terms and frequencies, then the query processor has no way to compute the scoring functions, and no way to relate document identifiers to the actual URLs. For this purpose, the index file was enhanced with two sections: the *URL mapping* section, that contains for each document ID the number of characters in that document, and the URL it relates to. The second section is the *index metadata* section, which contains information such as the highest document identifier. The highest document ID is needed when performing DAAT matching.

Initially, we had tried to consume all the queries in the given query file, and produce a partial index containing all the unique words in the given queries. However, this partial index ended up being too large to work with. Additionally, memory caching was heavily impacted, because the index is a map, which is a random access data structure. See how this problem was solved in Section 3.5.

3.2 Stopword removal and stemming

The procedure used to tokenize and pre-process a given query is exactly the same as the one used in the indexer when pre-processing the corpus. In fact, they use the same functions. You may find the implementation in the file `common/preprocessing/normalize.py`.

3.3 DAAT matching

Our implementation iterates over every document ID. For each document ID, it iterates of the words of the given query. For each word in the query, it retrieves the postings from the index and computes the score. The implementation can be found in the function `Ranker._score`, in the file `processor/_internal/processor/ranker.py`.

We use a simple max-heap ordered by score. The implementation of this heap can be found in the file `processor/_internal/processor/score_heap.py`.

3.4 Scoring policy

We computed TF as:

$$\frac{weight}{|D|}$$

IDF as:

$$ln\left(\frac{N-|P|+0.5}{|P|+0.5}+1\right)$$

Where N is the number of documents in the corpus, and $|P|$ is the number of postings related to the given term. And TFIDF as:

$$TFIDF = TF * IDF$$

The definitions for IDF and BM25 were extracted from Wikipedia [2]. The hiper-parameters used for BM25 were $k_1 = 1.5$ and $b = 0.75$. The choice between TFIDF and BM25 is decided in the function `Ranker._score`.

3.5 Parallelization

We initially tried to use a process pool to parallelize the query processor. When using the query pre-fetching described in Section 3.1, this made sense, because no file access was performed after the ranking started. This process pool granted us no speedup. In fact, the effect was negative: to create the processes, the memory of the parent process had to be copied, which resulted in multiple processes using about 3GB of RAM each.

Then, we switched to a thread pool. Each thread processes a single query. And, instead of loading the entire index statically to main memory, we switched to a dynamic approach: each thread fetches from disk the subindex it needs to process a specific query. The subindex is discarded after the query is processed. A major optimization we performed to enable this dynamic fetching was to have a pre-processing step when initializing the ranker, where we mark the index file at every megabyte. That is, at every megabyte of the index file, we store the first word of that megabyte in a map. This map is called `self._marks` in the `Ranker` class. The function `preprocess_entire_index` returns these marks. The function `find_checkpoint_marks`, given a list of words and a *marks* map, returns a list of checkpoints in the index file where those words can be found.

Note that by having these checkpoint markers in the index file, we guarantee that each thread only needs to do one disk access operation for each term in the query it is processing.

3.6 Performance analysis

To benchmark the query processor, we extracted the 1000 most popular queries in the Google web engine in 2017 from [1]. We let the query processor rank all of these queries using different number of threads. The results can be seen in Figure 3. Note that we only record the time for ranking. The time for preprocessing (which we estimate to be roughly 300 seconds in average) is discarded in this analysis.

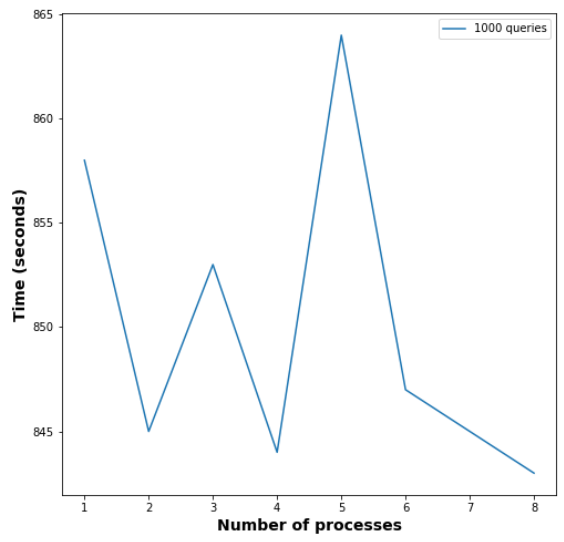

Figure 3: Execution time of ranking Google 2017 top 1000 most popular queries with varying numbers of threads.

As can be seen in Figure 3, the sequential ranking throughput is about 1000/858 ≈ 1.17 queries per second. We consider this to be quite fast, since the index is read dynamically from disk. The speedup achieved by using multiple threads is negligible. We have two hypothesis on why the speedup was so bad:

(1) CPU time is still the bottleneck of the ranking process (disk time is small). In this case, there is not much to do in terms of parallelization strategy. The best would be to improve the format of the index file itself, and the matching algorithm.
(2) The fact that all threads share the same files enforces the usage of mutual exclusion (mutexes). In this case, it could be useful to partition the index into multiple files as a pre-processing step.

To improve the speedup, we could multiple processes alongside the multiple user threads. This would require that the optimization suggested for hypothesis (1) above be implemented first.

3.7 Characterization

The assignment specification contains two sample queries we now use to estimate the effectiveness of the Query Processor. Below we show these two queries, as well as the tokens they map to after pre-processing.

1. *como fazer horta em casa* – faz, hort, cas
2. *o que é comorbidade?* – comorb

Next, we show some statistics of the score produced using TFIDF and BM25 scoring functions. Visually, BM25 seems to perform better for these queries, since the score distribution is more widespread.

3.7.1 TFIDF. For query 1, there were 211,289 documents matched (documents that contained one of the words). The distribution of scores can be seen in Figure 4. For query 2, there were 250 documents matched. The distribution of scores can be seen in Figure 5.

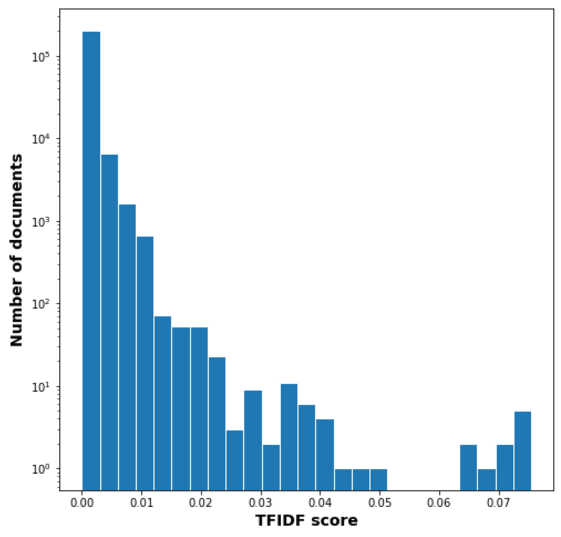

Figure 4: Distribution of TFIDF scores for query "como fazer horta em casa".

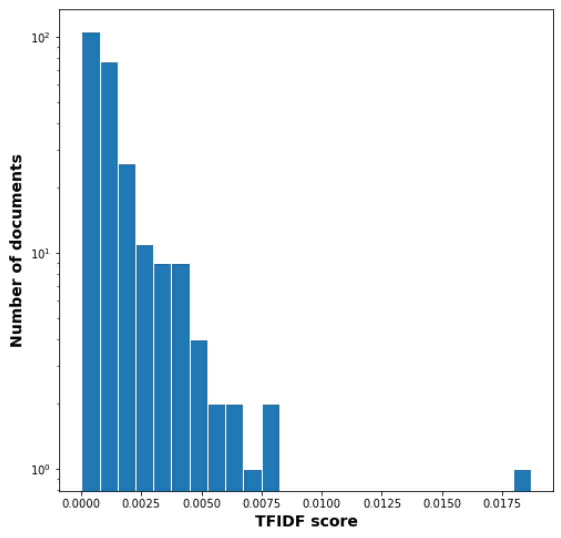

Figure 5: Distribution of TFIDF scores for query "o que é comorbidade?".

3.7.2 BM25. The number of documents matched is equal to that of TFIDF. See Figure 6 and Figure 7 for the score distributions.

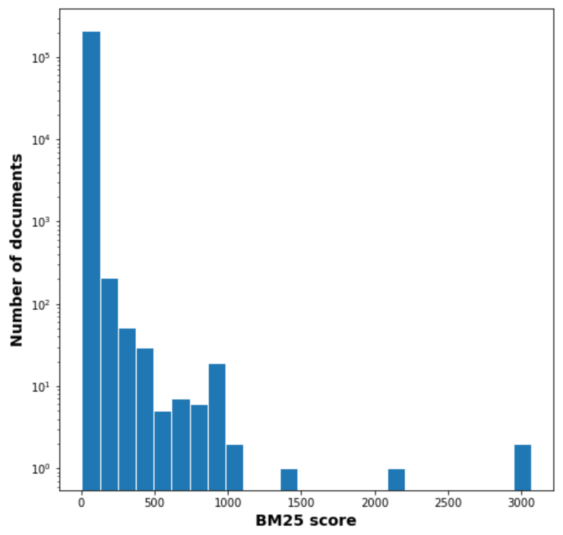

Figure 6: Distribution of BM25 scores for query "como fazer horta em casa".

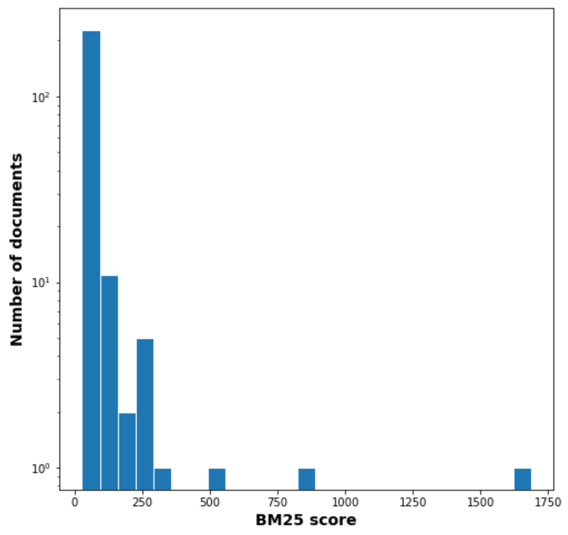

Figure 7: Distribution of BM25 scores for query "o que é comorbidade?".

A TESTING ENVIRONMENT

This project was written in Python, and runs over the Python Interpreter. It was tested with Python version 3.8, in a machine with Ubuntu 20.04.4 LTS, Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz with 4 cores, 16 GB of RAM.

B USAGE INSTRUCTIONS

The command-line interface of the indexer and query processor follows the assignment specification. Some flags were added to facilitate testing and benchmarking:

- Indexer
  - **-log-level** logging level. Logging level defaults to CRITICAL.
  - **-extra-statistics** whether to include extra statistics in the printed JSON at the end of the execution. Examples of extra statistics include the list of the number of postings for each term in the index.
  - **-track-memory** whether to log memory usage information during runtime. This was mainly used in the initial phases of development, and stopped working after the program was parallelized.
- Query Processor
  - **-log-level** logging level. Logging level defaults to CRITICAL.
  - **-parallelism** number of threads to use when ranking.
  - **-benchmarking** whether or not the program is being run for benchmarking purposes. If this flag is enabled, the program will print additional information.

REFERENCES

[1] *1000 most asked questions on Google.* https://www.mondovo.com/keywords/most-asked-questions-on-google.
[2] *BM25 – Wikipedia.* https://en.wikipedia.org/wiki/Okapi_BM25.
[3] *The nltk Python library.* https://www.nltk.org/.
[4] *The Portuguese stemmer used from nltk.* https://www.nltk.org/api/nltk.stem.snowball.html#nltk.stem.snowball.PortugueseStemmer.
[5] *The warcio Python library.* https://pypi.org/project/warcio/.