# PA2 - Indexer and Query Processor

Daniel Carneiro Freire
daniel.carneiro.freire@gmail.com
Universidade Federal de Minas Gerais
Belo Horizonte, Minas Gerais, Brazil

## ABSTRACT

In this programming assignment we were tasked with implementing two programs. A document indexer capable of operating on limited RAM, and a query processor. The implementation was made in python 3.10.5. The final indexes (and auxiliary files) are available at https://drive.google.com/drive/folders/142xLvQJN7vQDj4ouH1mTPxlk873SwifK?usp=sharing.

## 1 INTRODUCTION

In this assignment we were tasked with implementing two programs. A document indexer, which processes a corpus of documents and indexes each document in an inverted index. And a query processor, which reads a query and searches for relevant documents in the previously built index.

Since midway through the assignment the corpus was changed from web-pages to plaintext documents after the web-page version was already implemented, both versions were implemented in the final version. We will mention the original corpus as the HTML corpus and the new corpus as the plaintext corpus.

In the following section we'll be describing each implementation separately, first the indexer, then the query processor. Then we'll discuss the results and finally write a conclusion for the assignment. In this documentation whenever we're mentioning a function or an object in the actual code it will be written in `monospace`.

## 2 IMPLEMENTATION

We'll be discussing the implementation for the initial corpus that contained web-pages and not the one provided afterwards that only had plaintext, since the web-page version has everything the plaintext one has, and more.

### 2.1 Indexer

The basic idea of the indexer is very simple, and is described in algorithm 1. However there are several policies that must be followed, some of which posed a challenge.

The indexer had the following policies: Pre-processing, memory management and parallelization. The biggest challenge was the memory management, python is known to be very greedy when it comes to memory, taking a lot and not giving much back, and it does not provide tools to explicitly free allocated memory. What follows is the implementation details, including how each policy was implemented, and how the program as a whole works.

*2.1.1 Pre-processing Policy.* This policy was very simple to implement using the provided packages, specially NLTK, which provides the tokeninzer, the stemming function and the stopwords. The algorithm 2 shows how the preprocessing was done.

**Algorithm 1** Indexer

**Input:** Corpus
**Output:** Index
  Index ← A mapping of terms to a set of tuples *docid*, *term_count*
  docid ← 0
  **for each** document in the corpus **do**
    **for each** unique term in the document **do**
      count ← number of occurrences of the term in the document
      Index[term] ← Index[term] ∪ {(docid, count)}
    docid ← docid +1
  **return** Index

**Algorithm 2** Pre-processor

**Input:** Document
**Output:** Processed terms
  Tokens ← tokenize Document
  Filter non-word characters from Tokens (eg. punctuation)
  Filter stopwords from Tokens
  Stem Tokens
  **return** Tokens

*2.1.2 Memory Management Policy.* To make the program run in a manageable time (or at all) with the memory constraints (1024MB) a few things had to be implemented, starting at the document loading phase.

Firstly NLTK is a very big library, and python loads all of it. NLTK alone was consuming about 100MB, and each child process would have the same libraries, therefore if we had say 4 child processes we'd be consuming 500MB. This was not acceptable, so a lighter version of NLTK was forked from the original (Following the guidelines described in the library's license). This new library (`nltk_light`) consumed about 30 MB, a much more manageable size.

The loading of the documents was done lazily through a generator function, so that we only have in memory what we are currently processing. The provided corpus was not built with the specifications provided in the first assignment, nominally it did not only contain HTML documents. It also contained, among others, pdf documents, mp4 videos, images, binary executables, etc. At least one of the videos had more than 500 MB, spending half the memory "budget" instantaneously, so it was important to skip those documents. The excluded formats were: .mp4, .png, .fdm, .pdf, .doc, .dll, .exe, .jpg, .sh, .yml, .xsl, .xml, .mpq. This was done by checking if the url ended with one of those extensions. This does not guarantee that no files of those types are processed, to do that we'd have to

actually load them and that would defeat the whole purpose of filtering them.

**Algorithm 3** Count

**Input:** Document
**Output:** Mapping of terms to their frequency in the document
Tokens ← Pre-processor(Document)2
Count ← Reduce Tokens to a mapping of terms to their frequency
**return** Count

The actual processing of documents consisted the following steps (code written in `index.index_manager`): each document and its docid is passed as an argument to `create_count(/2)`which in turn calls `count_worker(/2)` with the same arguments. Then it extracts the visible text from each document, does the pre-processing, and creates a partial index for the provided document and writes it to a file. We call `create_count(/2)` instead of calling `count_worker(/2)` directly because the garbage collector (called with `gc.collect(/0)`) only collects objects that have no references to it, once `count_worker(/2)` returns, all its references are destroyed and we can call `gc.collect(/0)` to free some memory. This however does not guarantee that all the memory is given back to the system, since python will keep some of it for later use. The only way to guarantee that python frees memory to the system is by running tasks in subprocesses and killing them after the task is done.

This behavior of increased memory usage over-time could also be attributed to memory leaks (Supposedly the library BeautifulSoup is leaky), however the same behavior was observed when processing the plaintext corpus, which did not use BeautifulSoup. Besides, the leak would have to be very small, since all $\sim 10^6$ documents could be processed without exceeding the memory limit (and without restarting processes). In fact this increased memory-usage over time is observed even in the main process, which only runs the generator and multiprocessing libraries. It is possible of course that other libraries are leaky, causing the increasing memory use even when only processing plaintext (warcio in particullar seems to be leaky, although this was not confirmed by the author).

It is also worth noting that the BeautifulSoup parameter `parse_only` was used to skip parsing non-visible tags, such as the head or meta tags. This also decreased considerably the memory usage.

As we will describe in the next subsection, parallelization was done through the creation of subprocesses. It was determined that one process would use at most 250MB when processing documents (without restarting processes), therefore the maximum number of jobs we could run at the same time was ⌊max_memory/250⌋. However if we restart all processes after each WARC file is completely processed, we can lower that number to 150MB, which raises significantly the number of concurrent processes we can run, besides by restarting processes we do not have to call `gc.collect(/0)` after every `count_worker(/2)` returns, which increases considerably the performance of the algorithm.

Also relevant for Memory Management is the merging of the partial indexes, this was done in two steps, first we create many partial indexes consisting of the partial indexes of a 1000 documents each. This is done in order, meaning that the first partial index will have the partial indexes of the document with docid 0 to 999, etc. Then we merge these partial indexes into one final index.

In either the initial or the final merge, we can't load all those partial indexes to memory, we can't even load the first line of each partial index, without exceeding 1024MB. What was done was the creation of the class `index.file_buffer.FileBuffer`, which holds a file pointer, the id of the smallest document in the partial index and the current term corresponding to the line the file pointer is pointing to. It also has functions to retrieve the value for that term, and to skip to the next line. The class is hashable by the docid, and is partially orderable by first comparing the term, and if the term is equal, by then comparing the docid. This makes it so we can create a set containing the `FileBuffer`, and we can retrieve what to write next by getting the min of the set. This is described in algorithm 4.

**Algorithm 4** Merge

**Input:** Partial Index Files
**Output:** Merged Index
FileSet ← ∅
last ← ""
**for each** file in partial indexes **do**
    FileSet ← FileSet ∪ {FileBuffer(file)}
**while** FileSet is not empty **do**
    m ← min FileSet
    **if** m.token ≠ last **then**
        Write "\n" + m.token + ":"
    Write m.value()
    m.next()
    **if** m.token is None **then**
        FileSet ← FileSet \ {m}

*2.1.3 Parallelization Policy.* Parallelization was fairly straight forward using the external library `joblib`, but would be similar using python's standard `multiprocessing`. We have 3 tasks to accomplish, first we process the documents creating the initial partial indexes (or counts as described in algorithm 3), then we merge those partial indexes creating larger partial indexes. Then, finally, we merge those larger partial indexes creating a single, big index.

We can only parallelize the first two tasks. The document processing task was parallelized in $\lfloor max\_memory/150 \rfloor$ processes, which restarted after 10000 documents were processed. The first merge of partial indexes was parallelized in $\lfloor max\_memory/100 \rfloor$ processes, which were not restarted. The number of processes is limited by the number of CPUs in the machine.

The final indexer is shown in algorithm 5.

*2.1.4 Other details.* The indexes also produces auxiliary files to be later used by the query processor. Those are the urls_index, and the count files, which hold a mapping of docids to urls, and a mapping of docids to total term count, respectively.

The entrypoint for the indexer is `index_manager(/4)`, it's parameters are `corpus_path`, that's the path to the zip file containing the `warc.gz` files, `max_memory` which is the limit on memory to be used, `ndocs` which is an optional parameter specifying the total number of documents to be indexed and `plaintext` an optional

**Algorithm 5** Indexer - Final

**Input:** Corpus
**Output:** Index
count_jobs ← $\lfloor max\_memory/150 \rfloor$
partial_jobs ← $\lfloor max\_memory/100 \rfloor$
**for each** batch of 10000 documents in the Corpus **do**
Map Count (alg 3) onto the batch in parallel with count_jobs processes
Kill the subprocesses
Merge the counts in parallel with partial_jobs processes
Merge the partial indexes

parameter which defaults to `False` that specifies what corpus it will process, the plaintext one if `True` or the webpages one if `False`.

## 2.2 Query Processor

The query processor (algorithm 6) receives as input a sequence of queries and, using the specified ranking function, outputs the top 10 results for each query (algorithm 7). Similarly to the indexer, several policies had to be followed, namely: Pre-processing, Matching, Scoring and Parallelization. Those policies and implementation details will be described bellow.

**Algorithm 6** Single Query Processor

**Input:** Queries, Index, Ranking Function
**Output:** Top 10 ranking
$Q \leftarrow$ priority queue of tuples (rank, docid) of maximum size 10, prioritized by rank
**for each** document in the Index **do**
Put in $Q$ a tuple of the ranking (using the provided ranking function) of the query over the document and the docid of the document
**return** $Q$

**Algorithm 7** Query Processor

**Input:** Queries, Index, Ranking Function
**Output:** Top 10 ranking for each query
$R \leftarrow \{\}$
**for each** query **do**
$R \leftarrow \{\} \cup$ Single Query Processor(query, Index, Ranking Function) (alg 6)
**return** $R$

*2.2.1 Loading the Index.* The first step in the execution of the query processor is loading the index and auxiliary files. Both the urls_index and counts files are loaded to python's dictionaries at the beginning of the execution.

To hold the index, the class `query.index.PartialIndex` was created. This class is a partial index that contains only the specified terms. This partial index is a dictionary that maps terms to dictionary that maps documents to counts. This was an ideal structure for this task, since accessing values of a dictionary is $O(1)$ and so is checking if a key is present in the dictionary.

*2.2.2 Pre-processing Policy.* Each query is processed exactly as each document was processed in the Indexer (algorithm 2), so that the terms we get from each query, are the same terms that we indexed.

*2.2.3 Matching Policy.* The documents are ranked after performing a conjunctive DAAT matching, meaning that we only rank documents that contain all the terms of the query. When performing this matching, we start by the rarest terms in the query, so that we can exclude the most documents as soon as possible.

*2.2.4 Scoring Policy.* For scoring two functions were implemented, BM25 (equation 1) and TF-IDF (equation 2).

**BM25:**

$$\begin{aligned} \text{BM25}(D,Q) &= \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f_{q_i,D} \cdot (k_1+1)}{f_{q_i,D} + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)} \\ \text{IDF}(q_i) &= \ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right) \end{aligned} \tag{1}$$

**TF-IDF:**

$$\begin{aligned} \text{TF}(t,D) &= \frac{f_{t,D}}{\sum_{t' \in D} f_{t',D}} \\ \text{IDF}(t,D) &= \log\left(\frac{N}{|\{d \in D : t \in d\}|}\right) \\ \text{TF-IDF}(D,Q) &= \sum_{i=1}^{n} \text{TF}(q_i,D) * \text{IDF}(q_i,D) \end{aligned} \tag{2}$$

Where $f_{t,D}$ is the frequency of the term $t$ in the document $D$, $|D|$ is the length of the document $D$ in words, $avgdl$ is the average document length in the corpus, $k_1$ and $b$ are free parameters, chosen in this instance to be 1.5 and 0.75 respectively, $q_i$ is the i'th term of the query $Q$, $N$ is the total number of document in the corpus, and $n(q_i)$ is the number of documents containing $q_i$.

*2.2.5 Parallelization Policy.* Parallelization was implemented on a query-by-query basis, meaning that each query runs in a separate process.

Also running in parallel is the `query.logger.Logger` and a `multiprocessing.Manager` managing a queue for communicating with it. The logger takes any string added to the queue and prints it to stdout. This is done because printing in multiple processes at the same time can get messy.

## 2.3 Complexity Analysis

This complexity analysis was made assuming all documents and queries have the same number of terms, that the size of the document or query (in bytes) is upper bounded by some multiple of how many terms it has. The complexity for each function is listed in the tables bellow.

| Index | |
|---|---|
| `index_manager.index_manager` | $O(d \cdot n \log n)$ |
| `index_manager.create_count` | $O(n \log n)$ |
| `index_manager.count_worker_plain` | $O(n \log n)$ |
| `partial_index.merge_counts` | $O(n \cdot d)$ |
| `partial_index.merge_indexes` | $O(n \cdot d)$ |
| `partial_index.partial_index_cb` | $O(n \cdot d)$ |
| `partial_index.create_partial_index` | $O(n \cdot d)$ |
| `util.get_visible` | $O(n)$ |
| `util.count` | $O(1)$ |
| `util.warc_loader` | $O(d)$ |
| **Query** | |
| `QueryProcessor.load_count` | $O(d)$ |
| `QueryProcessor.load_urls` | $O(d)$ |
| `QueryProcessor.tf` | $O(1)$ |
| `QueryProcessor.idf` | $O(1)$ |
| `QueryProcessor.tf_idf` | $O(1)$ |
| `QueryProcessor.has_term` | $O(1)$ |
| `QueryProcessor.tf_idf_query` | $O(q)$ |
| `QueryProcessor.get_relevants` | $O(n \cdot q)$ |
| `QueryProcessor.bm_idf` | $O(1)$ |
| `QueryProcessor.bm25` | $O(q)$ |
| `QueryProcessor.bm25_query` | $O(q \cdot d)$ |
| `QueryProcessor.process_query` | $O(d(q+n))$ |
| `QueryProcessor.preprocess_query` | $O(q)$ |
| `QueryProcessor.process_queries` | $O(Q(d(q+n)))$ |
| `PartialIndex.set_index` | $O(n)$ |

| Where | |
|---|---|
| $n$ | number of terms of the document |
| $d$ | number of documents |
| $q$ | number of terms in the query |
| $Q$ | number of queries |

Therefore the overall complexity of the indexer is $O(d \cdot n \log n)$ and the overall complexity of the query processor is $O(Q(d(q+n)))$. In the following section we'll describe the results.

## 2.4 Results

In this section we'll describe the results for both provided corpora (html and plaintext). The numbers were rounded to two decimal places (With a few exceptions). The tests were run in a desktop computer with an AMD 5800X CPU, 32GB 2666MHZ DDR4 RAM, 1TB PCIe 4.0 NVMe M.2, running Manjaro Linux 21.3.0.

Table 1 shows a small summary of the produced index for both corpora. Note that the number of tokens for the html corpus was way smaller than for the plaintext corpus. This seems to have happened because when building the plaintext corpus, the visible text in different tags were joined without a separator, for instance by setting the parameter `separator` on `BeautifulSoup.get_text`. This caused many words to be concatenated.

**Table 1: Summary of the produced index for both corpora.**

| | html | plaintext |
|---|---|---|
| Number of Documents | 948096 | 948092 |
| Number of Tokens | 4807963 | 11623655 |

Table 2 shows the execution time of the indexer for both corpora. The plaintext version is faster for two reasons, we can skip the parsing step, and it consumes less memory, meaning we can use more processes. The memory usage when no fixed limit was set was 3.5GB for processing the html corpus and 2.41GB for processing the plaintext corpus.

**Table 2: Execution times for building the indexes**

| | html | plaintext |
|---|---|---|
| **1024MB** | | |
| Create Counts | 3.41h | 1.28h |
| Merge Counts | 1.15h | 1.00h |
| Merge Partial Indexes | 1.58h | 1.97h |
| Total | 6.14h | 4.25h |
| **Unlimited RAM** | | |
| Create Counts | 2.1h | 1.18h |
| Merge Counts | 0.97h | 0.87h |
| Merge Partial Indexes | 1.68h | 1.95h |
| Total | 4.75h | 4.00h |

Table 3 shows descriptive statistics of the lengths of the inverted lists for both indexes. In both most of the terms are unique to a single document. This was expected for the plaintext corpus (because of the concatenation issue) but was a unexpected for the html corpus.

**Table 3: Descriptive statistics for the inverted lists lengths of both indexes**

| corpus | mean | std | min | 50% | 75% | 95% | max |
|---|---|---|---|---|---|---|---|
| html | 64.70 | 2001.30 | 1 | 1 | 3 | 36 | 523135 |
| plaintext | 23.46 | 942.50 | 1 | 1 | 2 | 20 | 455928 |

Table 4 shows descriptive statistics for the number of matches for the provided 100 queries for both corpora. The html corpus had almost double the number of matches of the plaintext corpus, not only that but the plaintext corpus had many queries which did not match any document, probably because so many of its terms were concatenated as described previously.

**Table 4: Descriptive statistics for the number for the number of matched documents for each corpus**

| corpus | mean | std | min | 5% | 10% | 40% | 50% |
|---|---|---|---|---|---|---|---|
| html | 9.55 | 1.65 | 2 | 5.95 | 10 | 10 | 10 |
| plaintext | 5.53 | 4.82 | 0 | 0 | 0 | 1.2 | 10 |

Table 5 shows descriptive statistics for the score of each ranking function for the provided 100 queries for both corpora. Again the html corpus had a better performance, for both ranking functions the index produced from the html corpus had better scores on average, although not by a large margin. Having more matches probably makes the average score go down, also the penalizing mechanisms for common terms will have greater effect on the score of the html corpus, since it has less terms and larger inverted lists.

**Table 5: Descriptive statistics for the scores of each function for each corpus**

| **HTML** | | | | | | | |
|---|---|---|---|---|---|---|---|
| function | mean | std | min | 25% | 50% | 75% | max |
| bm25 | 0.37 | 0.54 | 0.000004 | 0.09 | 0.20 | 0.42 | 5.47 |
| tf-idf | 0.12 | 0.13 | 0.000342 | 0.04 | 0.08 | 0.16 | 1.30 |
| **Plaintext** | | | | | | | |
| bm25 | 0.36 | 0.64 | 0.000004 | 0.0336 | 0.14 | 0.47 | 7.27 |
| tf-idf | 0.11 | 0.16 | 0.00038 | 0.03 | 0.06 | 0.16 | 1.74 |

Table 6 shows the execution time and memory usage of the query processor for the provided 100 queries. It was decided to use 8 processes since it runs fast, and without consuming too much memory (arguably).

**Table 6: Execution time and maximum memory usage of the query processor for different number of concurrent processes**

| # processes | memory | time |
|---|---|---|
| 1 | 2.43GB | 7.47m |
| 2 | 4.37GB | 4.22m |
| 4 | 6.02GB | 2.42m |
| 6 | 7.11GB | 1.91m |
| 8 | 9.03GB | 1.72m |
| 16 | 14.80GB | 1.73m |

Table 7 shows descriptive statistics of the execution time (in seconds) of single queries. Most of the time processing queries is spent loading the partial index, therefore creating more compact indexes (for example by compression) could speed it up.

**Table 7: Descriptive statistics on the execution times of a single query**

| count | mean | std | min | 25% | 50% | 75% | max |
|---|---|---|---|---|---|---|---|
| 100.0 | 4.57 | 1.41 | 2.80 | 3.38 | 4.09 | 5.68 | 8.24 |

## 2.5 Conclusion

Both the indexer and the query processor were implemented following all of the specified policies and obtained reasonable results, both in execution time, and quality of output. However, many optimizations could be made, such as using ranking functions that use more document features, compressing the index, using a faster programming language and etc.