

Universidade Federal de Minas Gerais  
Departamento de Ciência da Computação  
TCC/TSI/TECC: Information Retrieval

## Programming Assignment #2 Indexer and Query Processor

**Deadline:** Jun 14th, 2022 23:59 via Moodle

**Overview** The goal of this assignment is to implement the indexer and query processor modules of a web search engine. In addition to the source code of your implementation, your submission must include a characterization of the index built for a mid-sized corpus and the results retrieved for a set of queries.

**Implementation** You must use Python 3 for this assignment. Your code must run in a virtual environment **using only the libraries included** in the provided `requirements.txt` file. Execution errors due to missing libraries or incompatible library versions will result in a zero grade. To make sure you have the correct setup, you can test it in one of the Linux machines provided by the Department of Computer Science<sup>1</sup> using the following commands:

```
$ python3 -m venv pa2
$ source pa2/bin/activate
$ pip3 install -r /path/to/requirements.txt
```

**Indexer** Your indexer implementation must build upon the provided `indexer.py` file, which will be executed in the same virtual environment described above. The `indexer.py` module will be executed as follows:

```
$ python3 indexer.py -m <MEMORY> -c <CORPUS> -i <INDEX>
```

with the following arguments:

- `-m <MEMORY>`: the memory available to the indexer in megabytes.
- `-c <CORPUS>`: the path to a directory containing the corpus WARC files.

---

<sup>1</sup><https://www.crc.dcc.ufmg.br/infraestrutura/laboratorios/linux>

- `-i <INDEX>`: the path to the index file to be generated.

At the end of the execution, your `indexer.py` implementation must print a JSON document to standard output<sup>2</sup> with the following statistics:

- `Index Size`, the index size in megabytes;
- `Elapsed Time`, the time elapsed (in seconds) to produce the index;
- `Number of Lists`, the number of inverted lists in the index;
- `Average List Size`, the average number of postings per inverted lists.

The following example illustrates the required output format:

```
{ "Index Size": 2354,
  "Elapsed Time": 45235,
  "Number of Lists": 437,
  "Average List Size": 23.4 }
```

**Indexing Policies** For each document in the target corpus (see the `-c` argument above), your implementation must tokenize, pre-process and index it. At the end of the execution, a final representation of the inverted index should be stored as a single file at the designated path (the `-i` argument). Because of the imposed memory restriction (the `-m` argument), your implementation must not rely entirely on the available memory to produce the index. Instead, partial indexes should be produced in memory and merged on disk, either during the indexing process or at the end of it. To speed up development, you are encouraged to use the smaller corpus you crawled during Programming Assignment #1 and play with different memory restriction settings. Your final implementation must be able to index a larger corpus comprising one million documents, which will be made available shortly. In addition to this standard workflow, **your implementation must abide by the following indexing policies:**

1. *Pre-processing Policy.* To reduce the index size, your implementation **must perform stopword removal and stemming**. Additional pre-processing techniques can be implemented at your discretion.
2. *Memory Management Policy.* To ensure robustness, your implementation **must execute under limited memory availability**. In the provided `indexer.py` file, a RAM limitation mechanism is implemented, which your implementation must obey. Note that the actual memory limit (the `-m` argument) refers to the total memory available to your implementation, not only to the memory needed to store the index structures. Your implementation must be able to operate with as little memory as 1024mb.

---

<sup>2</sup>[https://en.wikipedia.org/wiki/Standard\\_streams#Standard\\_output\\_\(stdout\)](https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout))

3. *Parallelization Policy*. To ensure maximum efficiency, you **must parallelize the indexing process across multiple threads**. You may experiment to find an optimal number of threads to minimize indexing time rate while minimizing the incurred parallelization overhead.
4. *Compression Policy (extra)*. Optionally, you may choose to implement a compression scheme for index entries (e.g. gamma for docids, unary for term frequency) for maximum storage efficiency.

**Query Processor** Your query processor implementation must build upon the provided `processor.py` file, which will be executed in the same virtual environment described above, as follows:

```
$ python3 processor.py -i <INDEX> -q <QUERIES> -r <RANKER>
```

with the following arguments:

- `-i <INDEX>`: the path to an index file.
- `-q <QUERIES>`: the path to a file with the list of queries to process.
- `-r <RANKER>`: a string informing the ranking function (either “TFIDF” or “BM25”) to be used to score documents for each query.

After processing **each query**, your `processor.py` implementation must print a JSON document to standard output with the top results retrieved for that query according to the following format:

- **Query**, the query text;
- **Results**, a list of results.

Each result in the **Results** list should be represented with the fields:

- **URL**, the respective page URL;
- **Score**, the final document score.

The following example illustrates the required output format for a query:

```
{ "Query": "Sistemas UFMG",  
  "Results": [  
    { "URL": "https://virtual.ufmg.br/",  
      "Score": 24.2 },  
    { "URL": "https://sistemas.ufmg.br/",  
      "Score": 12.4 } ] }
```

The results list for a query must be sorted in reverse document score order and include up to the top 10 results retrieved for that query.

**Query Processing Policies** For each query in the list provided via the `-q` argument, your implementation must pre-process the query, retrieve candidate documents from the given index (the `-i` argument), score these documents according to the chosen ranking model (the `-r` argument), and print the top 10 results using the aforementioned format. In addition to this standard workflow, **your implementation must abide by the following policies:**

1. *Pre-processing Policy.* Your implementation **must pre-process queries with stopwords removal and stemming**. This policy should be aligned with the implemented document pre-processing policy for indexing to correctly match queries with documents.
2. *Matching Policy.* For improved efficiency, your implementation **must perform a conjunctive document-at-a-time (DAAT) matching** when retrieving candidate documents.
3. *Scoring Policy.* Your implementation **must provide two scoring functions: TFIDF and BM25**. You are free to experiment with different variants of these functions from the literature.
4. *Parallelization Policy.* To ensure maximum efficiency, you **must parallelize the query processing across multiple threads**. You may experiment to find an optimal number of threads to maximize your query processing throughput while minimizing the incurred parallelization overhead.
5. *Serving Policy (extra).* Optionally, you may choose to implement a small client-server application to serve the results of your `processor.py` implementation via a web-based interface.

**Deliverables** Before the deadline (Jun 14th, 2022 23:59), you must submit a package file (`zip`) via Moodle containing the following:

1. Source code of your implementation;
2. Link to the inverted index (stored on Google Drive) produced for the provided one-million-page document corpus;
3. Documentation file (`pdf`, max 5 pages).

Your `indexer.py` and `processor.py` files should be located at the root of your submitted zip file. You must guarantee that the index generated by your `indexer.py` can be correctly processed by your `processor.py`.

**Grading** This assignment is worth a total of 15 points distributed as:

- 10 points for your *implementation*, assessed based on the quality of your source code, including its overall organization (modularity, readability, indentation, use of comments) and appropriate use of data structures, as well as on how well it abides by the aforementioned indexing and query processing policies.

- 5 points for your *documentation*, assessed based on a short (**pdf**) report<sup>3</sup> describing your implemented data structures and algorithms, their computational complexity, as well as a discussion of their empirical efficiency (e.g. the time elapsed during each step of indexing and query processing, the speedup achieved via parallelization). Your documentation should also include a characterization of your produced inverted index, including (but not limited to) the following statistics: number of documents, number of tokens, number of inverted lists, and a distribution of the number of postings per inverted list. Likewise, you should include a characterization of the results produced for the provided test queries, such as the number of matched documents per query as well as statistics of the score distributions for the two implemented ranking functions (TFIDF and BM25).

**Late Submissions** Late submissions will be penalized in  $2^{(d-1)} - 0.5$  points, where  $d > 0$  is the number of days late. In practice, a submission 5 or more days late will result in a zero grade.

**Teams** This assignment must be performed **individually**. Any sign of plagiarism will be investigated and reported to the appropriate authorities.

---

<sup>3</sup>Your documentation should be no longer than 5 pages and use the ACM L<sup>A</sup>T<sub>E</sub>X template (sample-sigconf.tex): <https://www.acm.org/binaries/content/assets/publications/consolidated-tex-template/acmart-primary.zip>