Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

TCC/TSI/TECC: Information Retrieval

Programming Assignment #2
Indexer and Query Processor

Deadline: June 2nd, 2025 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 3.13 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 using the following commands:

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

Indexer Your implementation must include an `indexer.py` file, which will be executed in the same virtual environment described above, 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 the corpus file to be indexed.
- `-i <INDEX>`: the path to the directory where indexes should be written.

At the end of the execution, your `indexer.py` implementation must print a JSON document to standard output[1] 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 }
```

Document Corpus The corpus to be indexed comprises structured representations (with id, title, descriptive text, and keywords) for a total of 4,641,784 named entities present in Wikipedia. These structured representations are encoded as JSON documents in a single JSONL file available for download.[2] To speed up development, you are encouraged to use a smaller portion of the corpus to test your implementation before you try to index the complete version.

Indexing Policies For each document in the corpus (the `-c` argument above), your implementation must parse, tokenize, and index it. Your implementation must operate within the designated memory budget (the `-m` argument) during its entire execution.[3][4] This emulates the most typical scenario where the target corpus far exceeds the amount of physical memory available to the indexer. At the end of the execution, a final representation of all produced index structures (inverted index, document index, term lexicon) must be stored as three separate files, one for each structure, at the designated directory (the `-i` argument).

In addition to this workflow, **your implementation must abide by the following policies**, which will determine your final grade in this assignment:

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.

[1] `https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)`

[2] `https://www.kaggle.com/datasets/rodrygo/entities`

[3] The memory limit will be strictly enforced during grading. If your program exceeds it, it may be automatically terminated with an out-of-memory (OOM) error. To prevent this, use `psutil.Process(os.getpid()).memory_info().rss` to monitor your current memory usage (in bytes), and offload partial indexes to disk before allocating more memory as needed.

[4] Note that the memory budget refers to the total memory available to your implementation, not only to the memory needed to store the actual index structures. As a reference lower bound, assume your implementation will be tested with `-m 1024`.

2. *Memory Management Policy.* To ensure robustness, your implementation **must execute under limited memory availability**. To this end, it must be able to produce partial indexes in memory (respecting the imposed memory budget) and merge them on disk.[5]

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 implementation must include a `processor.py` file, which will be executed in the previously described environment, 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** (the `-q` argument above), your `processor.py` implementation must print a JSON document to standard output[6] 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 must be represented with the fields:

- `ID`, the respective result ID;
- `Score`, the final document score.

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

```
{ "Query": "information retrieval",
  "Results": [
    { "ID": "0512698",
      "Score": 24.2 },
    { "ID": "0249777",
      "Score": 12.4 }, ... ] }
```

[5] https://en.wikipedia.org/wiki/External_sorting#External_merge_sort
[6] https://en.wikipedia.org/wiki/Standard_streams#Standard_output_(stdout)

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 stopword 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 (extra).* To ensure maximum efficiency, you **may parallelize the query processing across multiple threads**. You may experiment to find an optimal number of threads to maximize your throughput while minimizing the incurred parallelization overhead.

Deliverables Before the deadline (June 2nd, 2025 23:59), you must submit a package file (`zip`) via Moodle containing the following:

1. Source code of your implementation;

2. Documentation file (`pdf`, max 3 pages);

3. Link to the produced index structures (stored on Google Drive).

Your `indexer.py` and `processor.py` files must 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[7] 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.

[7] Your documentation should be no longer than 3 pages and use the ACM LaTeX template (sample-sigconf.tex): `https://portalparts.acm.org/hippo/latex_templates/acmart-primary.zip`