Exploring Information Retrieval Techniques Through Programming Assignment 2

João Vítor Fernandes Dias
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil
joaovitorfd2000@ufmg.br

Abstract

This report documents the implementation of an indexer and query processor for a search engine system. The solution handles large-scale document processing within strict memory constraints using parallelization and efficient data structures. The system includes stopword removal, stemming, conjunctive document-at-a-time matching, and supports both TFIDF and BM25 ranking functions. Empirical evaluation shows (not so) efficient indexing of 4.6M Wikipedia entities and effective query processing.

CCS Concepts

• **Information systems → Information retrieval**; *Information extraction*; *Retrieval effectiveness*; *Retrieval efficiency*; *Distributed retrieval*; Data structures; • **Social and professional topics →** Acceptable use policy restrictions; Student assessment.

Keywords

Information Retrieval, Indexer, Python, Query Processor, TFIDF, BM25, Stopword Removal, Stemming, Parallelization

ACM Reference Format:
João Vítor Fernandes Dias. 2025. Exploring Information Retrieval Techniques Through Programming Assignment 2. In *Proceedings of Information Retrieval (IR '25)*. ACM, New York, NY, USA, 3 pages. https://doi.org/NoDOI.ForNow

1 Introduction

The system consists of two main components implemented in Python 3.13: the `indexer.py` for indexing a large corpus of entities from Wikipedia into three index structures (inverted index, document index, and term lexicon) and the `processor.py` for processing user queries against the indexed data and scoring results using either TFIDF or BM25 ranking functions. The indexing process is designed to handle a large corpus of 4,641,784 documents while adhering to a user-defined memory budget, utilizing parallelization for efficiency.

1.1 Indexer (`indexer.py`)

Processes a JSONL corpus containing 4,641,784 Wikipedia entities. For each document, it creates a thread pool to parallelize the indexing process. Considering the user-defined memory budget, the goal was to keep the least memory usage possible while still being able to index the entire corpus. For that matter, it was decided to always load the previouly created index structures from disk, append the new document to them, and then save them back to disk. The goal was to avoid exploding the memory limit.

1.2 Query Processor (`processor.py`)

The query processor is designed to handle user queries against the indexed data. It implements a Document-at-a-Time (DAAT) retrieval model, which processes queries by retrieving postings lists for each term in the query and finding common documents that match all terms. The processor supports two ranking functions: TFIDF and BM25, allowing users to choose their preferred method for scoring results. It returns the top-10 results in JSON format, including the processed query, matched document IDs, and their corresponding scores.

No parallelization was implemented in the query processor since it was an optional feature and the query processing time was already low enough to not require it, since it was tested with a small subset of 10,000 documents. The DAAT model was built in such a way that it would only load the postings lists for the terms in the query, which reduces the memory usage and speeds up the query processing time. Also, all the documents that didn't present the terms in the query were filtered out before the scoring step, unless no documents matched the query, in which case all documents were scored. This approach was to ensure the recall of the top-10 results, even if they were not so relevant to the query.

2 Data Structures

The system uses three main index structures to store the indexed data:

- **`inverted_index.json`**: maps each term to a list of postings, where each posting is a list containing the document ID and the term frequency in that document.
- **`document_index.json`**: contains metadata for each document, including their tokenized title, text, keywords and number of unique terms.
- **`term_lexicon.json`**: maps each term to a unique ID and includes a histogram of term frequencies across the corpus.

2.1 Inverted Index

Notice that the json conversion converted tuples to lists. This approach seemed later unuseful, since a key-value mapping would be a more direct way of getting the term frequency for a document while scoring in the So, the postings list were later converted for faster scoring purposes, but they are being converted for each query, which is not the most efficient way of doing it and could be improved in the future.

Inverted index structure:

```
{
  term_1 (str): [[docID (int), freq (int)], ...],
}
```

2.2 Document Index

Although only the text content of the documents was previously indexed, the document index contains all the tokenized information of the documents, including their title, text, and keywords. Each of them is tokenized, stemmed, and had their stopwords removed. Their length were also computed and stored for later use at the scoring step in the which isn't actually useful since python stores the length of the list as an attribute what certainly is generated when the json content is loaded. Another possible optimization would be to convert all theses therms into their respective term IDs, which could reduce the memory usage.

Document index structure:

```
{
  docID (str): {
    "title": {"length": int, "words":[word_1, ...]},
    "text": {"length": int, "words":[word_1, ...]},
    "keywords": {"length": int,
      "words":[[keyword_1, ...], ...]},
    "unique_terms": {"length": int}
  }, ...
}
```

2.3 Term Lexicon

The term lexicon is a mapping of each term to a unique ID, which is used to efficiently retrieve postings lists during query processing. It also includes a histogram of term frequencies across the corpus, allowing for quick access to the number of occurrences of each term. For some reason the term ID was stored as a string what was not intended at first but was later kept for consistency.

Term Lexicon structure:

```
{
"term2id": {term_1 (str): id_1 (str), ... },
"terms_histogram": {id_1 (str): count_1 (int), ... }
}
```

3 Algorithms and Complexity

Two main algorithms are implemented in the system: the indexing process and the query processing. Parallelization apart, the overall algorithms are as follows. They both share some common preprocessing steps, such as tokenization, stemming, and stopword removal using the `nltk` library that was modularized in the `auxiliar.py` file. The `psutil` library was used to monitor the memory usage of the process and ensure it stays within the user-defined budget. As for the processor, its core algorithm is based on the Document-At-A-Time (DAAT) and the BM25 and TFIDF ranking functions, those are located at the `p_rankers.py` file.

3.1 Indexing Process

The indexing process reads the corpus file line by line, tokenizes each document's title, text, and keywords, removes stopwords, stems the terms, and computes a term histogram for each document. It then updates the term lexicon, inverted index, and document index structures in a thread-safe manner while trying to keep the memory usage within the user-defined budget.

Algorithm 1 Indexer

```
Initialize index structures
for each document in corpus do
    Loads new line from corpus file as JSON
    Tokenize, remove stopwords and stem: document title, text, and keywords
    Compute term histogram for all unique terms in the document

    Compute Term Lexicon:
        Create term ID mapping
        Create term frequency histogram
        Thread-safe append to term lexicon
    Compute Inverted Index:
    for each term in the histogram do
            Get term ID from Term Lexicon
            If term not in index, create new entry
            Append posting [docID, term frequency]
            Thread-safe append to inverted index
    end for
    Compute Document Index:
        Create document entry with ID, title, text, keywords, and unique terms
end for
```

Indexing complexity:

- **Preprocessing:** $O(t)$ per document (t = terms count)
- **Index update:** $O(1)$ per term with concurrent dictionaries
- **Overall:** $O(d\bar{t})$ (d = |documents|, $\bar{t}$ = avg terms per doc)

3.2 Query Processing

The query processor retrieves postings lists for each term in the query, performs a conjunctive match to find documents that match all terms, if no documents match, it keeps all postings, and then scores the matching documents using the selected ranking function (TFIDF or BM25). The top-10 results are returned in JSON format.

Algorithm 2 Query Processor

```
Load index structures to memory
for query in queries.txt do
    Preprocess query (tokenize, stem, remove stopwords)
    Retrieve postings lists for all query terms
    From query postings select only documents that match all terms (conjunctive match)
    if no documents match then
        Keep all postings
    end if
    for each DocID in postings do
        Calculate score using selected ranker (TFIDF/BM25)
    end for
    Append top-10 scored documents for query
end for
Return queries results
```

Query processing complexity:

- **Preprocessing:** $O(q)$ per query (q = query terms count)
- **Postings retrieval:** $O(qn)$ per term (n = postings list size)
- **DAAT matching:** $O(m)$ where $m = \min(|p_1|, |p_2|, ...)$ (conjunctive match) and p_i = postings list for term i
- **Scoring:** $O(m)$ per document
- **Overall:** $O(q + qn + 2m)$

Ranking Functions:

- $TFIDF(q,d) = \sum_{t \in q} tf_{t,d} \cdot idf_t$
  - $tf_{t,d} : \frac{d_t}{|d|}$
  - $idf_t = \log \frac{|Corpus|}{d}$
- $BM25(q, d, b = 0.75, k1 = 1.2) = \sum_{t \in q} tf_{t,d} \cdot idf_t$
  - $tf_{t,d} = \frac{d_t \cdot (k1+1)}{d_t + k1 \cdot (1 - b + b \cdot \frac{|d|}{avgdl})}$
  - $idf_t = \log \frac{|Corpus| - Corpus_{tf} + 0.5}{Corpus_{tf} + 0.5}$

4 Empirical Evaluation

As part of the empirical evaluation, the indexing and query processing were tested on a subset of 10,000 documents from the full corpus. The tests were meant to evaluate the progression of threading and memory usage during the indexing process.

4.1 Indexing Performance

Even after processing 10k documents distributed into queries, it seems that some kind of overwritting ocurred, since the maximum doc id is 10k but only 1843 entries were found at the document index. The postings were distributed as expected: the terms that are remarkably frequent in the corpus, such as punctuation marks, results in a large number of postings.

- All documents: 4,641,784
- Number of documents processed: 10,000
- Actually stored indexed documents: 1,843
- Inverted index terms: 15448
- Lexicon terms: 31102
- Most Frequent Terms: ",": 16969; ".": 15319, "(": 7556, ")": 7548

4.2 Query Processing

Results for sample queries against the 10k document subset:

- **physics nobel winners:** *Conj. Matches:* 0, *Matches*: 154, *BM25*: [16.59, 14.41, 13.41, 13.41, 11.11, 11.11, 9.29, 9.08, 8.91, 8.91]; *TFIDF:* [.58, .17, .41, .68, .31, .40, .38, .33, .32, .27]
- **christopher nolan movies:** *Conj. Matches:* 0, *Matches*: 23, *BM25*: [11.62, 11.12, 9.84, 9.84, 9.18, 8.78, 8.78, 8.78, 8.78, 7.77]; *TFIDF*: [.17, .15, .12, .00, .00, .00, .00, .00, .00, .00]
- **19th female authors:** *Conj. Matches:* 0, *Matches*: 67, *BM25*: [16.64, 16.64, 10.60, 10.60, 10.18, 10.18, 8.32, 8.32, 8.32, 8.32]; *TFIDF*: [.41, .33, .24, .18, .16, .16, .15, .15, .13, .00]
- **german cs universities:** *Conj. Matches:* 0, *Matches*: 86, *BM25*: [12.04, 8.88, 8.49, 7.51, 7.51, 7.51, 7.51, 7.51, 7.51, 7.51]; *TFIDF:* [.68, .60, .27, .26, .21, .21, .20, .16, .13, .00]
- **radiohead albums:** *Conj. Matches:* 0, *Matches*: 324, *BM25*: [3.33, 3.31, 3.31, 3.31, 3.31, 3.30, 3.30, 3.30, 3.30, 3.30]; *TFIDF:* [3.10, 2.89, 2.41, 2.21, 2.01, 1.98, 1.96, 1.61, 1.58, 1.56]

4.3 Parallelization Analysis

Thread scaling tests on 1000 documents showed no significant performance gains with parallelization (up to 32 threads). I/O contention from disk writes created a bottleneck, as each thread locked index structures during writes. **Note:** the Table 1's first row is for 10k documents; others for 1000.

Table 1: Thread Indexing scaling tests

Threads	Total Memory	Time	Lists	Avg Size
32	335.31 MB	3936 s	15448	4.8
32	190.87 MB	48 s	3311	2.7
16	184.58 MB	49 s	4571	2.8
16	X	54 s	4982	2.9
8	X	65 s	6200	3.2
1	184.80 MB	42 s	3801	2.8

5 Conclusion

The implemented system could potentially index large corpora but isn't ideally parallelized due to the I/O contention caused by disk writes. The memory usage was monitored and as an attempt to keep it lower as possible, the index structures were always loaded from disk, updated with the new document, and then saved back to disk. Which indeed kept the memory usage low, but also caused a bottleneck in the indexing process, which proved highly inefficient.

Due to the small number of documents indexed, the conjunctive matching in the query processor was compromised, since no documents matched all the stemmed terms in the queries. However, the processor was able to retrieve documents that matched at least one term in the query in order to be able to score them and return the top-10 results.

Received 02 June 2025