# 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/XXXXXXX.XXXXXXX

## 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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*IR '25, May 04– June 02, 2025, Belo Horizonte, MG*
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2025/04
https://doi.org/XXXXXXX.XXXXXXX

### 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.

The indexer ends up creating three index structures: the inverted index, the document index, and the term lexicon. It's importante to notice that the index structures were created considering only the text content of the documents since it most likely contains the terms present in the title and keywords. Another possible approach would be to merge them all together.

### 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

**Algorithm 1** Indexing

```
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

**Algorithm 2** DAAT Query Processing

```
Preprocess query (tokenize, stem, remove stopwords)
Retrieve postings lists for all query terms
Find common documents (conjunctive match)
for each candidate document do
    Calculate score using selected ranker (TFIDF/BM25)
end for
Return top-10 scored documents
```

**Ranking Functions:**

- **TFIDF**: $w_{t,d} = \frac{tf_{t,d}}{|d|} \times \log \frac{N}{df_t}$
- **BM25**: $\sum_{t \in q} \log \frac{N-df_t+0.5}{df_t+0.5} \times \frac{(k_1+1)tf_{t,d}}{tf_{t,d}+k_1(1-b+b\frac{|d|}{avgdl})}$

## 4 Empirical Evaluation

### 4.1 Indexing Performance

Indexing results for 10k document subset (1GB memory limit):

**Table 1: Indexing Statistics**

| Metric | Value |
|---|---|
| Index Size | 148 MB |
| Elapsed Time | 325 seconds |
| Number of Lists | 42,817 |
| Average List Size | 8.3 |
| Throughput | 30.7 docs/second |

**Figure 1: Memory usage during indexing (1GB limit)**

### 4.2 Index Characteristics

Full corpus index statistics:

- Documents: 4,641,784
- Unique terms: 2,843,192
- Inverted lists: 2,843,192
- Postings distribution:
  - 75% of lists have $\leq$ 15 postings
  - Top 1% cover 25% of total postings
  - Longest list: "unit" (189,452 postings)

### 4.3 Query Processing

Results for sample queries (BM25 ranking):

**Table 2: Query Results**

| Query | Matches | Top Score |
|---|---|---|
| physics nobel winners | 7,412 | 24.8 |
| christopher nolan movies | 183 | 32.6 |
| 19th female authors | 9,847 | 18.2 |
| german cs universities | 672 | 27.4 |
| radiohead albums | 94 | 41.3 |

**Figure 2: Score distribution comparison (TFIDF vs BM25)**

## 5 Parallelization Analysis

Thread scaling tests (10k documents):

Optimal thread count: 8-16 (diminishing returns beyond due to I/O contention)

**Table 3: Parallelization Speedup**

| Threads | Time (s) | Speedup |
|---|---|---|
| 1 | 623 | 1.00× |
| 4 | 189 | 3.30× |
| 8 | 132 | 4.72× |
| 16 | 125 | 4.98× |
| 32 | 128 | 4.87× |

## 6 Conclusion

The implemented system efficiently indexes large corpora within strict memory constraints using:

- Thread-based parallelization with optimal 8-16 threads
- Memory monitoring with psutil for budget compliance
- Efficient DAAT query processing with BM25/TFIDF

Future work could include index compression and more advanced ranking features.

### 6.1 Empirical Efficiency

After tweaking a lot with the threads, some notable improvements were made. The Table 4 below shows the performance of the crawler in terms of time taken to crawl a certain number of URLs, as well as the expected time for crawling 100k URLs based on the mean time per URL.

**Table 4: Crawling performance**

| (1) | (2) | (3) | (4) | (5) | (6) | (7) |
|---|---|---|---|---|---|---|
| 1 | 1647 | 5767 | 1904.00 | 1,16 | 32:13:20 | |
| 8 | 100 | 1590 | 25.00 | 0,25 | 06:56:40 | |
| 10 | 1002 | 3268 | 584.00 | 0,58 | 16:06:40 | Thread Pool |
| 40 | 537 | 1204 | 47.35 | 0,09 | 02:30:00 | Thread Pool (Fixed) |
| 40 | 539 | 598 | 58.98 | 0,11 | 03:03:20 | No WARC Batch |
| **50** | 100 | 558 | 6.26 | 0,06 | **01:40:00** | |
| 50 | 100 | 848 | 12.97 | 0,13 | 03:36:40 | Domain only |
| 50 | 1048 | 3189 | 82.60 | 0,08 | 02:13:20 | Thread Pool |
| 50 | 502 | 2048 | 109.63 | 0,22 | 06:06:40 | Thread Pool: broken warc |
| 50 | 502 | 2092 | 187.36 | 0,37 | 10:16:40 | Thread Pool: batch 100 |
| 50 | 534 | 2563 | 44.40 | 0,08 | 02:13:20 | No WARC Batch.gz |
| 50 | 538 | 2079 | 37.76 | 0,07 | 01:56:40 | Thread Pool |
| **60** | 559 | 2252 | 32.63 | 0,06 | **01:40:00** | Thread Pool |
| 60 | 1200 | 2113 | 82.67 | 0,07 | 01:56:40 | Converted to py |
| 70 | 547 | 1721 | 51.55 | 0,09 | 02:30:00 | Thread Pool: Worsened |

(1) Number of threads used in the crawling process
(2) Number of URLs crawled
(3) Number of URLs in the frontier
(4) Time taken to crawl the URLs (in seconds)
(5) Time taken per URL (in seconds)
(6) Expected time to crawl 100,000 URLs (in hours:minutes:seconds)
(7) Description of the crawling process

## 7 Conclusion

Received 02 June 2025