UNIVERSITÀ DI PISA

Department of Information Engineering
Artificial Intelligence and Data Engineering

Multimedia Information Retrieval & Computer Vision

Document Search Engine

Michael Asante

Academic Year 2021/2022

Contents

1 Introduction

This project aims to create a document search engine with a collection of documents totaling about 9 million documents. To achieve the objective of this task, two primarily different activities were done:

1. Indexing of the documents

   This process was quite important in achieving a robust and efficient system. It involved the creation of the entire data structures strategy, needed to develop the system.

2. Query processing

   This process involved the retrieval of the relevant documents as top k, which is returned to the user, This ensured that the two metrics, efficiency and effectiveness were achieved for the user with results returned following a query.

Project Source code is also available on GitHub through this link: MIRCV Search Engine

1.1 Project structure & modules

The modules of the project are grouped into:

- *commandlineinterface* → This scope is responsible for accepting a user's query and passes them to module *QueryHandler* module to continue program execution.
- *common* → contains the main application Java classes and used by other parts of the project like the compression module, utils, preprocesser etc.
- *indexer* → performs the indexing of the collection and saves the main data structures created on disk.
- *performanceTest* → performs some query tests and writes the results for trec_eval.
- *queryHandler* → processed the query inputted from the *commandlineinterface* and returns the list of the top k relevant documents according to such query.

2 Indexing

2.1 Preprocessing of the documents

The Preprocess class is responsible for preprocessing documents and queries. The code block is defined in this path:

it/unipi/dii/aide/mircv/preprocess/Preprocesser.java

The various processes defined in this Java class include

2.1.1 Text cleaning

The documents containing text to be preprocessed will be done using these entire methods defined:

1. Removal of non-Unicode characters
2. **URL Matcher**: This regex matches URLs starting with `http`, `https`, `ftp`, or `mailto` (case-insensitive). It handles optional `www.`, domain names with alphanumeric characters and some special characters, followed by a domain extension (2-24 characters), and then optional path/query parameters.
3. **HTML Tags Matcher**: This regex matches HTML tags by finding strings that start with <, followed by any character except >, and end with >.
4. **Non Digit Matcher**: This regex matches any character that is not an alphabetic letter (uppercase or lowercase) or a whitespace character.
5. **Multiple Space Matcher**: This matcher removes one or more whitespace characters such as tabs, newlines and spaces.
6. **Consective Letters Matcher**: This regex matches any character that appears at least three times consecutively.
7. **Camel Case Matcher**: Removal of extra whitespaces at the beginning, end and in the middle of the text

Text Before Preprocessing:
"Visit our site at https://www.example.com and check out our awesome HTML page! <html><body>Welcome to Example.com! </body></html> You can also email us at info@example.com.

We love codingInCamelCase and numbers like 1234 and 5678."

Text After Preprocessing steps:
"Visit our site at and check out our awesome HTML page Welcome to Examplecom You can also email us at infoexamplecom We love coding In Camel Case and numbers like and "

2.1.2 Tokenization and Normalization of text

Tokenization is performed by first, splitting on white-spaces and then on camel case. All tokens are then lower-cased in a list of array.

”ThisIs an example Text”
↓
[this, is, an, example, text]

2.1.3 Stopword removal and stemming

This phase is optional. A list of stopword files have been placed in a config file used for removing unnecessary tokens and using the library *Porter Stemmer*.

”The runners were running swiftly towards the finishing line in the competition.”
↓
[the runner were run swift toward the finish line in the compet]

2.2 Indexing

The indexing is the process where the data structures are created inversely, known as the inverted index for the vocabulary (terms) and the document index. Two separate files handle this phase. The *Single-pass in-memory indexing class*, and the *Merger* class.

2.2.1 Spimi (*it/unipi/dii/aide/mircv/algorithms/Spimi.java*)

The Java class `Spimi` implements the SPIMI (Single-Pass In-Memory Indexing) algorithm to create an inverted index from a collection of documents. During the SPIMI algorithm execution, documents are read sequentially from the collection file, which can be either compressed (e.g., a .tar.gz archive) or uncompressed (e.g., a .tsv file). Each document undergoes preprocessing and is then indexed into partial posting lists. The `initializeBuffer` method sets up a buffer reader for the collection file, while the

`rollback` method handles the cleanup of partial data structures and the document index file in case of errors.

As documents are indexed, the `executeSpimi` method processes them one by one, assigning each a unique document ID (docid) in incremental order. This method also ensures memory constraints are respected by periodically flushing partial indexes to disk. The `saveIndexToDisk` method writes these partial indexes, sorted lexicographically, to disk, storing document IDs and term frequencies separately. The `updateOrAddPosting` method updates or adds postings in the posting list for each term encountered. Each partial inverted index comprises three files: one for posting list docids, one for frequencies, and one for the partial vocabulary. The document entries are computed and written to disk immediately after indexing, ensuring efficient storage and retrieval.

2.2.2 Merging (*it/unipi/dii/aide/mircv/algorithms/Merger.java*)

The `Merger` class is responsible for reading the intermediate inverted indexes created by the SPIMI algorithm, and merging them into a single, final index. The merger processes each term in lexicographic order, concatenating posting lists from all partial indexes while ensuring that postings are sorted by increasing document ID. The final merged posting list is then written to disk, possibly with compression applied to document IDs and frequencies. Similarly, each vocabulary entry is updated and written to the merged vocabulary file.

Posting list format Posting lists are written to disk using one file for storing *docids* and one for storing frequencies, in particular:

- */data/invertedIndexDocs* → stores *docids*
- */data/invertedIndexFreqs* → stores *frequencies*

√ Posting lists are divided in **skipblocks**. Each skipblock contains at most $[\sqrt{n}]$ with n being posting list's length in terms of number of postings, and if $n \leq 1024$ we use a single skip block not to waste further resources.

The information about each skipblock is stored inside a *block descriptor* which is written in the following file:

• */data/blockDescriptors* → stores *block descriptors*

Each *block descriptor* occupies 32*bytes* and is written according to the format shown in figure 1.

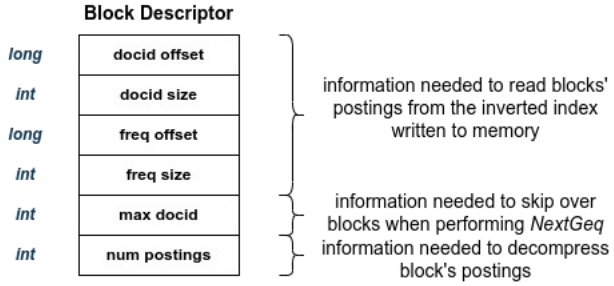


Figure 1: Block descriptor format

Vocabulary format The vocabulary is stored in:

• */data/vocabulary*

A single vocabulary entry follows the format as the diagram below

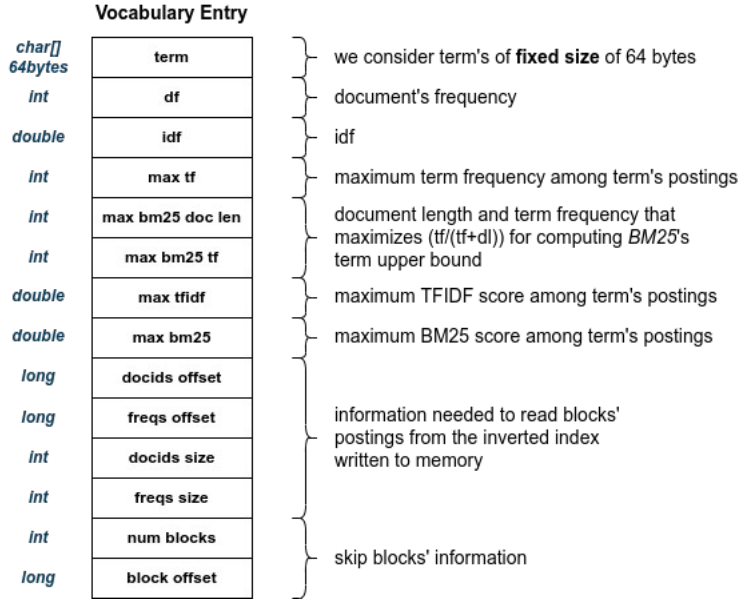


Figure 2: Vocabulary format

2.3 Compression

The compression algorithms used in order to reduce the memory occupancy of the inverted indexes were:

- *Variable byte* algorithm → to compress *docids*
- *Unary* algorithm → to compress *frequencies*

2.3.1 Unary compression

Unary encoding is a simple form of data encoding where each number `n` is represented as a sequence of `n` ones followed by a zero. This is a form of unary coding where the value of an integer directly influences the length of its representation.
The UnaryCompressor class contains two main methods:

- The integerArrayCompression
  This method basically compresses an array of integers using the unary compression. The total number of bits needed to represent the array is computed by summing up the values in the input array. Each integer `i` will contribute `i` bits to the final output. Then, compute the number of bytes required to store these bits. Since a byte contains 8 bits, the total number of bytes is the ceiling of `nBits/8. For each integer in the input array, it` convert the integer into a series of bits (each bit set to 1) followed by a terminating bit (0) and these are written into the `compressedArray`.

- The integerArrayDecompression
  This method decompresses an array of bytes back into integers using unary compression. First, it traverses through the bytes and bits to reconstruct the original integers. Then count consecutive ones to determine the value of each integer. As a consequent of determining where each integer ends, a zero bit helps in identifying where one integer ends and the next starts.

It is important to note that instead of compressing frequencies as single integers, we compress all the frequencies contained in the same skip block in a sequential way, in such a way to waste at most 7 bits for each block instead of wasting bits for each frequency. This way, the set of frequencies of a single skipblock is written in a byte-aligned way, while the single frequency of a block is written in a bit-aligned way.

2.3.2 Variable Byte compression

Variable Byte Encoding (VBE) is another form of data encoding used primarily in compression of integers where the length of the encoded data varies depending on the value of the integer. It uses a variable-length representation to compress integers more efficiently. The following methods were defined in the class.

- IntegerCompression
  Compresses a single integer into a variable-length byte array. First, determine the number of bytes needed to represent the integer based on its value. Then the value is encoded and converted into a base-128 representation while setting the Most Significant bit.

- IntegerArrayCompression
  Compresses an array of integers into a byte array using variable-byte encoding. For each integer, compress it into bytes and collect these bytes in a list. Convert the list of bytes into an array for the final compressed output.

- IntegerArrayDecompression
  Decompresses a byte array back into an array of integers. Traverse through the byte array. Combine bytes to reconstruct the integer values. Each byte in the encoded form has its MSB set to indicate whether it is the last byte of the integer.

3 Query Processing

Each time a new query arrives, firstly preprocessing is done to get its terms, then the query of the terms' vocabulary entries are retrieved to get information about the position of posting lists on disk and other statistics. Document scoring can be performed either with DAAT or MaxScore and the top k results are shown to the user.

3.1 Vocabulary's entry search

To retrieve a vocabulary entry, the cache is queried first which was defined using the Least Recent Used cache: which is an algorithm which organizes elements in order of use. That is the element that hasn't been used for the longest time will be deleted from the cache. If the term is not present in cache, it is retrieved from disk and then cached. For a better and efficient retrieval on disk, since entries are sorted in an increasing lexicographical order, binary search has been implemented.

3.2 Document Scoring

The default scoring strategy used is *Document at a time* (DAAT), which can be changed if the user states it as part of the query arguments to *MaxScore* as a dynamic pruning algorithm. The scoring functions that can be used are *TF-IDF* or *BM25*.

3.2.1 MaxScore Term Upper Bounds

Depending on the scoring function being used, a different value as term upper bound for determining which posting lists are essential and which are not is used. Both term upper bounds are computed **during indexing** and **stored in each term's vocabulary entry**.

- In case of *TF-IDF*, we use as term upper bound the **maximum** *TF-IDF* **among term's postings**
- In case of *BM25*, instead of using the maximum *BM25*'s value among term's postings which will not be efficient in terms of time, we use the value of *BM25* given by the pair of values *tf* and *document length* which **maximizes the ratio** $tf/(tf + dl)$

3.2.2 Posting list

Open list & close list operations

To efficiently manage and retrieve postings from disk, the `PostingList` class provides methods to open and close the posting list.

- **Opening the Posting List (`openPostingList`)**: This operation initializes the posting list for query processing. It retrieves the skip blocks' metadata from the block descriptor file, loading this information into main memory,

and initializes the necessary iterators. This ensures that the list is prepared for sequential reading and processing of postings.

- **Closing the Posting List (`closePostingList`)**: Once the posting list has been fully processed, this operation cleans up by clearing all in-memory data structures and removing the term from the vocabulary. This helps in freeing up memory resources and ensures that no unnecessary data is kept in memory, preventing potential memory leaks.

Next

The `next` method is designed to retrieve the next posting from the posting list:

- If there are postings remaining in the current block, the method reads the next posting directly from main memory.
- If the current block has no more postings to read, the method fetches the next block, loads the corresponding portion of the posting list into memory, and then reads the next posting.
- If there are no more blocks to read, the method returns `null`, indicating the end of the posting list.

This approach ensures efficient sequential access to the postings, minimizing the need for frequent disk I/O operations.

NextGEQ

The `nextGEQ` method serves an important role in dynamic pruning and efficient scoring of documents, particularly in conjunctive query processing. Given a document ID (`docid`) as input, the method operates as follows:

- **Block Search**: It searches through the blocks to find the first block where the maximum document ID is greater than or equal to the input `docid`.

- **Block Loading**: Once the appropriate block is identified, it retrieves the block from disk and loads the corresponding portion of the posting list into memory.
- **Posting Retrieval**: The method then iterates through the postings in the loaded block to find and return the first posting with a document ID greater than or equal to the input `docid`.
- **Null Return**: If no such posting exists in the remaining blocks, the method returns `null`.

This function enables efficient skipping over irrelevant postings, reducing the number of comparisons needed and improving query processing performance.

4 Performance

4.1 Indexing

As discussed in section 2 (Indexing section), we can index the collection with or without compression and with or without stopword removal and stemming. In the table below, the indexing time and the size of the inverted index files and the vocabulary in all of the possible combinations are compared.

Indexing type	**Duration**	**Docids size**	**Frequencies size**	**Vocabulary size**
Raw	28 min	1287.00 MB	1287.00 MB	158.71 MB
Compressed	26 min	1210.65 MB	59.53 MB	158.71 MB
Preprocessed	31 min	677.77 MB	677.77 MB	130.42 MB
Both	30 min	637.55 MB	29.86 MB	130.42 MB

Table 1:Indexing statistics & performances.

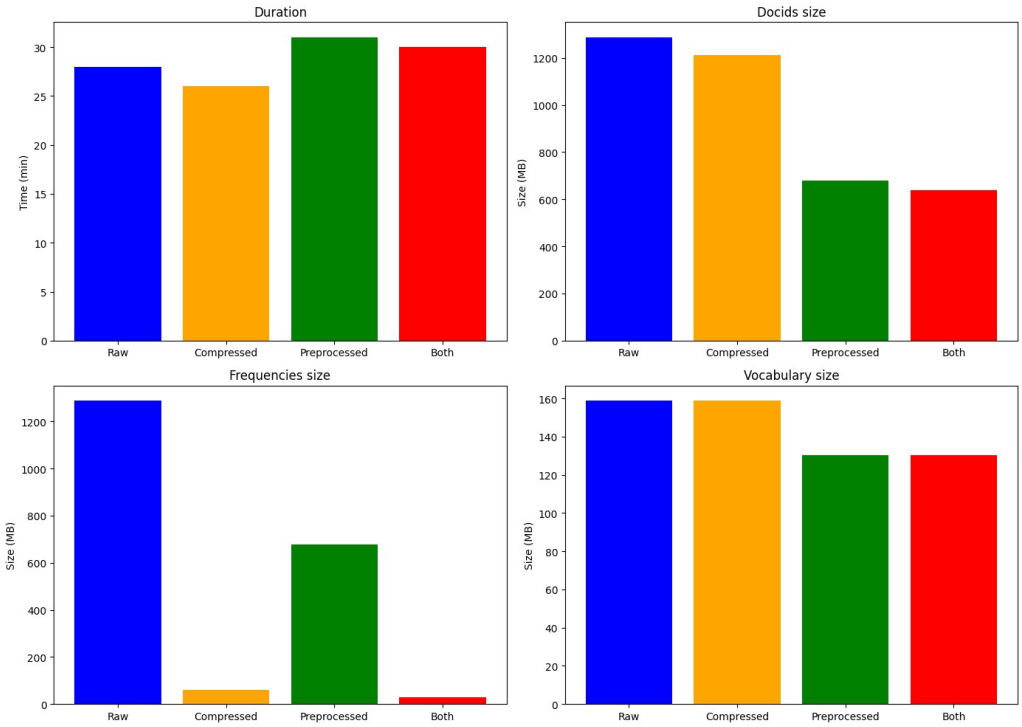
As we can see from the graph below in figure 5, there is a **great reduction in memory occupancy**, especially for frequencies' file **when using compression or preprocessing** (its size is almost halved), and by combining both of them, we obtain a total inverted index which only is a quarter of the size of the raw inverted index.

In addition, a few observations that can be made from the output are:
Compressed indexing provides the fastest processing time and significantly reduces the size of Frequencies.

Both indexing achieves the smallest sizes for both Docids and Frequencies, making it optimal for storage efficiency.

Raw indexing is the least efficient in terms of space, with the largest sizes for both Docids and Frequencies.

Preprocessed indexing balances between time and storage, with reduced Vocabulary size and moderate Docids and Frequencies sizes.



4.1.1 Compression

Going more into detail, for index compression we can see that by compressing index docids and frequencies, we **saved 689 MB of memory occupancy** (figure 7) almost without changing the time needed to index the whole document collection, the price we have

to pay is of course a growth in query's response time, in particular our search engine's **query's mean response time increased of** 2*ms* (figure 8), this is acceptable since we got a great decrease of memory waste. So, how we can see from graphs in figure 9 compression works really well.

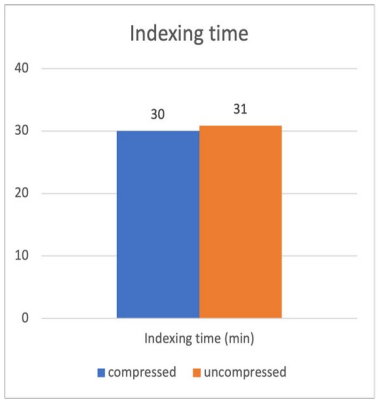


Figure 6: Indexing time comparison

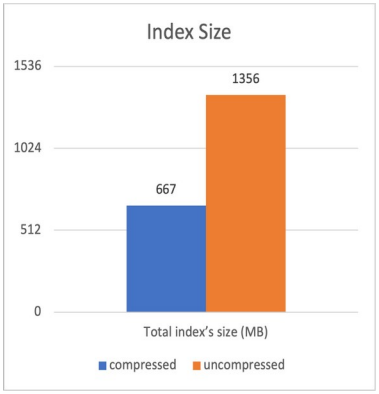


Figure 7: Indexing size comparison

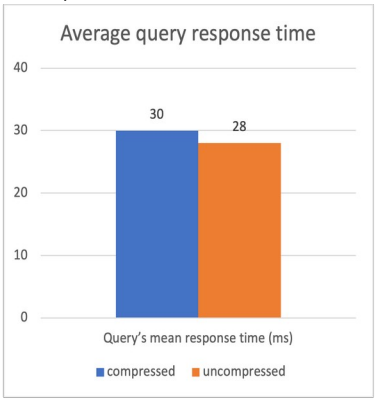


Figure 8: Query's mean response time comparison

Figure: Indexing performances with/without index compression

4.2 Query Handling & trec eval

4.2.1 DAAT & MaxScore comparison

In order to quantize the time saved using by *MaxScore* instead of simply *DAAT*, we compared the mean response time on *MSMARCO dev queries* using as scoring function *TFIDF*, and we have shown the results in table 2.

	Query's mean response time	
Scoring Algorithm	**Mean response time**	**Std.dev**
DAAT	48 ms	40.2
Max Score	25 ms	21.2

Table 2:Comparison between DAAT and MaxScore

As we can see from table 2, *DAAT* almost doubles the mean response time, so it is strongly convenient to use *MaxScore* algorithm to score queries.

4.2.2 BM25 and TFIDF comparison

The second performance comparison we want to show is the one between the two scoring functions implemented in our search engine: *BM25* and *TFIDF*. In table 3, there is the comparison of mean response time, mean average precision and precision@10 obtained using both the two scoring functions, with *MaxScore* as scoring algorithm.

	Statistics		
Scoring Algorithm	**Mean response time**	**map**	**p@10**
TFIDF	25 ms ± 21.2	0.0689	0.0084
BM25	32 ms ± 25.8	0.0656	0.0085

Table 3:Scoring functions' performances

As we can see, performances are similar both in terms of speed and effectiveness.

4.2.3 Stemming and stopwords removal

As last comparison we show the performance reached by our search engine when stemming and stopword removal are performed; the results in terms of mean response time and

evaluation metrics using *MaxScore* as scoring algorithm and *BM25* are shown in table 4.

	Statistics			
Full preprocessing	**max score**	**DAAT**	**map**	**p@10**
Disabled	261 ms ± 261.16	1119 ms ± 625.02	0.0839	0.0107
Enabled	32 ms ± 25.8	56 ms ± 39.74	0.0656	0.0085

Table 4:Performances with or without text preprocessing

Although we have a slightly better mean average precision *(1,27x)* and precision@10 *(1,25x)* the mean average response time increases massively when we don't remove stopwords *(8,16x, that becomes 19,98x when using DAAT)*, so we prefer to use the processed index to leverage the retrieval speed.

4.3 Caching

As aforementioned in subsection 3.1, a caching mechanism of the vocabulary entries has been exploited. To evaluate the performances, we computed the average time required to process queries with *TF-IDF* and *MaxScore* enabled with and without leveraging such cache.

	Query's mean response time	
Full Preprocessing	**Without cache**	**With cache**
Enabled	39 ms	31 ms
Disabled	244 ms	227 ms

Table 5:Average query's response time with/without caching.

As we can see from table 5, the usage of a cache allowed to save up to almost 26% in the first case and around 7% in the latter of time required to return documents to an user. This shows that the real increase in speed is given by *MaxScore* and the bottleneck is the length of the posting lists processed.