UNIVERSITÀ DI PISA

Department of Information Engineering
Artificial Intelligence and Data Engineering
Multimedia Information Retrieval & Computer Vision course

Search Engine

B. Tessa, F. Pezzuti, P. Tempesti

Academic Year 2022/2023

Contents

1 Introduction **1**
- 1.1 Project structure & main modules . . . 1

2 Indexing **2**
- 2.1 Preprocessing of the documents . . . 2
  - 2.1.1 Text cleaning . . . 2
  - 2.1.2 Tokenization and text normalisation . . . 2
  - 2.1.3 Stopword removal and stemming . . . 2
- 2.2 Indexing . . . 3
  - 2.2.1 Spimi . . . 3
  - 2.2.2 Merging . . . 3
- 2.3 Compression . . . 5
  - 2.3.1 Unary improvement for Java . . . 5

3 Query Processing **5**
- 3.1 Vocabulary's entry search . . . 6
- 3.2 Document Scoring . . . 6
  - 3.2.1 MaxScore Term Upper Bounds . . . 6
  - 3.2.2 Posting list interface . . . 6

4 Performance **7**
- 4.1 Indexing . . . 7
  - 4.1.1 Compression . . . 8
- 4.2 Query Handling & trec_eval . . . 9
  - 4.2.1 DAAT & MaxScore comparison . . . 9
  - 4.2.2 BM25 and TFIDF comparison . . . 9
  - 4.2.3 Stemming and stopwords removal . . . 9
- 4.3 Caching . . . 10

5 Limitations and further improvements **10**

1 Introduction

This academic project aims to create a search engine performing text retrieval on a collection of 8.8 millions of documents. To this end, two main steps were carried out:

1. Indexing of the documents
2. Query processing

The first step involves the creation of all the data structures needed for the retrieval whilst the second the actual retrieval of relevant documents according to a given query. After a detailed explanation of the aforementioned steps, a performance analysis will follow. The performance will be measured in terms of memory usage and building time for the indexing phase and in terms of efficiency and effectiveness for the query processing one.

Project's source code is available on *Github* at the following link:

`https://github.com/PieTempesti98/searchEngine`

1.1 Project structure & main modules

The main modules of the *Java* project are:

- ***Cli*** → receives queries from users, passes them to module *QueryHandler* and shows the results
- ***Common*** → contains the core Java classes and functions used by the other modules like *Posting list* class, *Vocabulary* class, *Preprocesser* class, and so on.
- ***Indexer*** → performs the indexing of the collection and saves the main data structures on disk.
- ***PerformanceTest*** → performs tests and writes the results in a format suitable for *trec_eval*
- ***QueryHandler*** → processed the query inputted from the *Cli* and returns the list of the most relevant documents according to such query

All project's major functionalities have been tested with *Junit*.

2 Indexing

2.1 Preprocessing of the documents

The class implementing documents and queries preprocessing is:

it.unipi.dii.aide.mircv.commons.preprocess.Preprocesser

2.1.1 Text cleaning

The text to be preprocessed will be transformed in the following manner:

1. Removal of non-Unicode characters
2. Removal of URLs
3. Removal of HTML tags
4. Collapse of 3 or more repeating character into 2
5. Removal of punctuation and numbers
6. Removal of extra whitespaces at the beginning, end and in the middle of the text

"The university of Pisa https://www.unipi.it has beeen founded in 1343 !!!"

↓

"The university of Pisa has been founded in"

All transformations are performed through the usage of regular expressions

2.1.2 Tokenization and text normalisation

We perform tokenization firstly by splitting on whitespaces and then on camel case. All tokens are then lower-cased.

"ThisIs an example Text"

↓

[this, is, an, example, text]

2.1.3 Stopword removal and stemming

This phase is optional. We downloaded a list of English stopwords, used for removing useless tokens and we used a library implementing the *Porter Stemmer*.

"this is an example document containing words"

↓

[exampl, document, word]

2.2 Indexing

The indexing is the process of creation of the main data structures, namely the inverted index, the vocabulary and the document index. Such process is carried out in two separate phases: the first one being *Single-pass in-memory indexing* (also known as *Spimi*), and the second one being the *Merger* algorithm.

2.2.1 Spimi

During the *Spimi algorithm* execution the documents are read subsequently one at a time from the collection file (which can be compressed in a *.tar.gz* archive or a raw *.tsv* file), preprocessed and then indexed in partial posting lists. The Java class that implements the *Spimi* algorithm is:

it.unipi.dii.aide.mircv.algorithms.Spimi

The Document Index is built and stored on disk, in particular once a document has been indexed, we compute and immediately write its document entry to disk. Each document is given a *docid* assigned in incremental order, used for internal representation.

Each partial inverted index is composed of:a file for storing posting lists' docids, one for the frequencies and a last one storing its partial vocabulary.

2.2.2 Merging

The Java class that implements the merging algorithm is:

it.unipi.dii.aide.mircv.algorithms.Merger

The *Merger* reads the inverted indexes produced by *Spimi* and processes them one term at a time in lexicographic order. Each time a new term is processed, all the posting lists corresponding to the term from all the partial inverted indexes are concatenated together ensuring postings ordering according to increasing docid. The merged posting list is then written to disk (possibly using compression of docids and frequencies), same for its vocabulary entry which is written to merged vocabulary's 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 $\lceil \sqrt{n} \rceil$ 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.

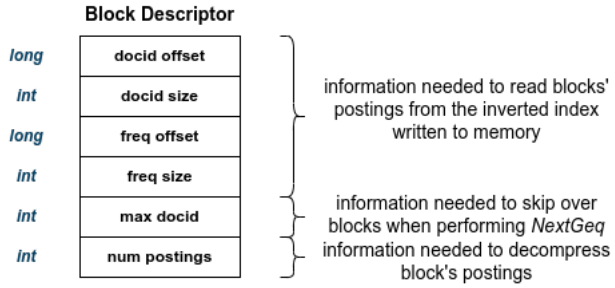

Figure 1: Block descriptor format

Vocabulary format The vocabulary is stored in:

- */data/vocabulary*

A single vocabulary entry follows the format reported in figure 2.

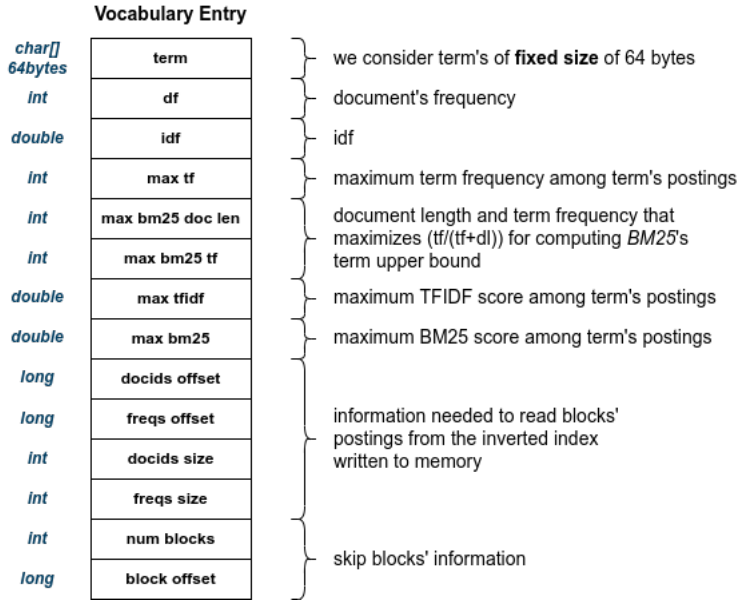

Figure 2: Vocabulary format

2.3 Compression

The compression algorithms we decided to use in order to reduce inverted index's memory occupancy are:

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

2.3.1 Unary improvement for Java

Due to our choice of developing the project using *Java* as programming language, which can read and write no less than a single byte, we implemented the following intuition: 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.

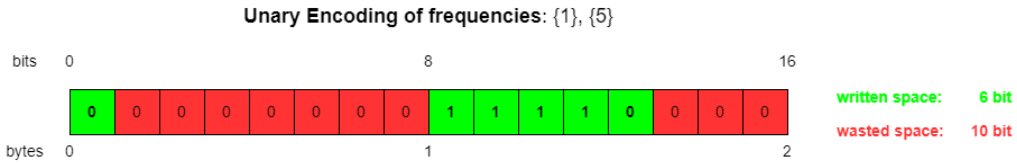

Figure 3: Example of *Unary* algorithm without improvement for Java

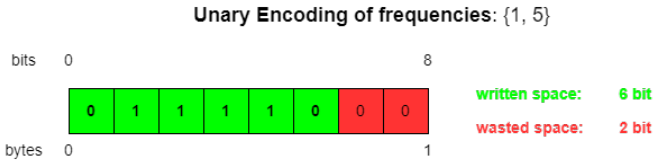

Figure 4: Example of *Unary* algorithm improvement for Java

As we can see from figures 3 and 4, with the improvement for Java we saved 8 bits (so, we saved an entire byte) when storing integers $\{1, 5\}$.

3 Query Processing

Each time a new query arrives, firstly we preprocess it to get its terms, then query 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 eventually the results are shown to the user.

3.1 Vocabulary's entry search

To retrieve a vocabulary entry, the cache is queried first: if the entry is not present in cache, it is retrieved from disk and then cached. For a more efficient retrieval on disk, since entries are for sure sorted in increasing lexicographical order, binary search has been implemented.

3.2 Document Scoring

The scoring strategy implemented is *Document at a time* (DAAT), but it can be optionally improved by using *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, we use a different value as term upper bound for determining which posting lists are essential and which are not. Both term upper bounds are **pre-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 (it would be too much time consuming), 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 interface

Open list & close list operations In order to retrieve from disk a posting list's postings, firstly one should perform an *open* operation on the list in such a way that the information about the list's skipblocks are retrieved from the block descriptor file and kept in main memory and that a *filechannel* is opened. Once the posting list has been processed, it must be closed using the *close* operation in such a way that main memory is cleaned and the *filechannel* is closed.

Next If there are postings left to be read in the current block, we read the next posting from main memory, else we have to read a new block and load the portion of posting list corresponding to such block. If there's nothing left to read we return a *null* value.

NextGEQ Given as input a docid d, we search among the list of blocks to be still processed, the first block having maximum docid $\geq d$, if found, we retrieve from disk the portion of posting list corresponding to such block and we return the first posting having docid $\geq d$; if it doesn't exist, we return a *null* value. This function is used to implement skipping in dynamic pruning and score documents in conjunctive mode.

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 table 1 we can see the indexing time and the size of the inverted index files and the vocabulary in all of the possible combinations.

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 shown 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.

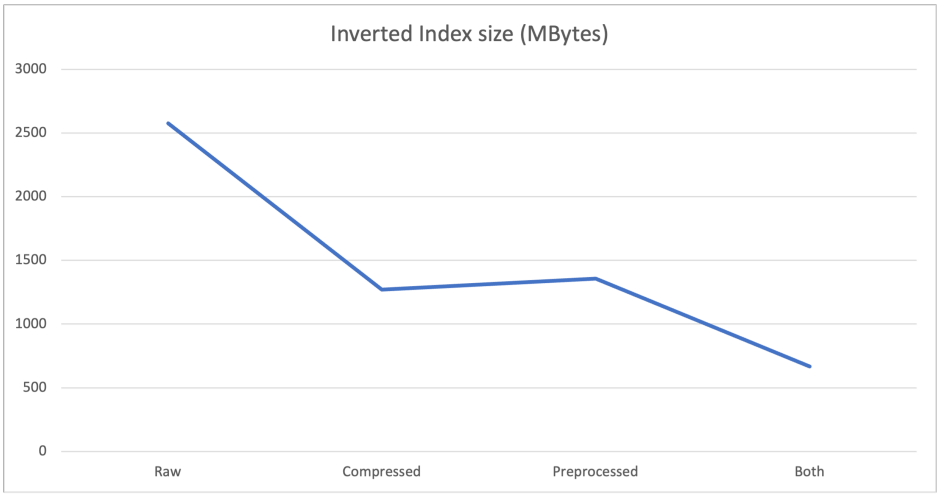

Figure 5: Size of the inverted indexes

Speaking about indexing building time, from table 1 we can see how compression also has a positive impact, decreasing the processing of the whole collection by 1-2 minutes; whilst indexing time with advanced text preprocessing increases by 3-4 minutes, because of stemming and stopwords removal.

4.1.1 Compression

Going more into detail, for what concerns 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 (figure 6), 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** $2ms$ (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.

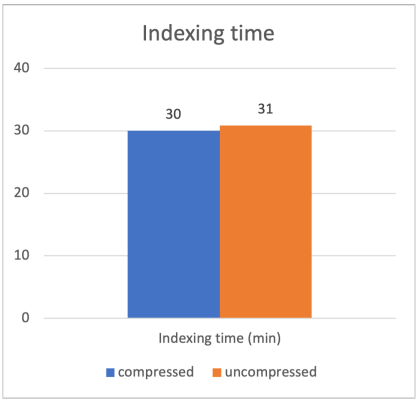

Figure 6: Indexing time comparison

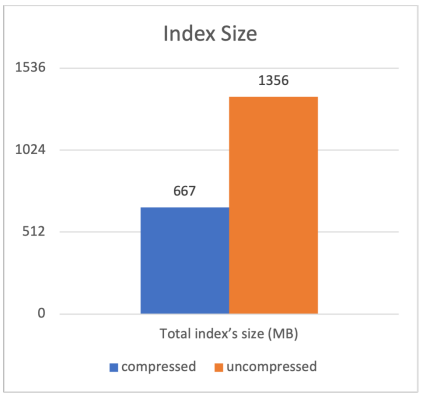

Figure 7: Indexing size comparison

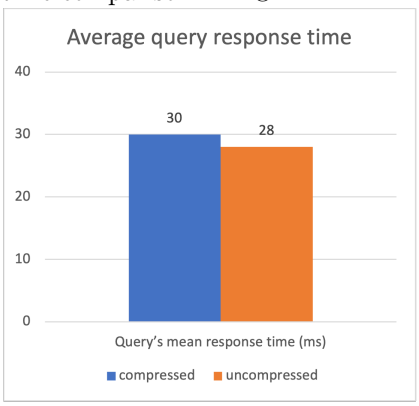

Figure 8: Query's mean response time comparison

Figure 9: 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 we compared 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.

5 Limitations and further improvements

A first improvement could be decreasing the size of each *document index entry* by bringing down the pid size (currently fixed at 64 *bytes*) to a smaller one. This not only will result in a lower memory usage, but also in an increase in speed of the set up time of the query handler, which at the start of the program reads the whole document index from disk. This could also decrease the building time of the index.

Another improvement could be implementing a caching strategy of query results or posting lists in addition to caching vocabulary entries, for a lower query response time.

Both for preprocessing and compression, it could be beneficial testing our collection against other techniques and algorithms to evaluate whether the performances increase or not.