UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering
Multimedia Information Retrieval and Computer Vision

Search Engine

Enrico Nello, Giacomo Pacini, Matteo Pierucci

0. Contents

1. Introduction

This project aims to construct a search engine that conducts text retrieval operations on an extensive compilation of 8.8 million documents available at [1]. To achieve this objective, the project is split upon two primary stages:

1. Document Indexing: the initial phase is focused on establishing essential data structures required for efficient retrieval.
2. Query Execution: the subsequent stage focuses on the actual retrieval of pertinent documents in response to specific user queries.

Eventually, an in-depth performance assessment is carried out. This evaluation encompasses various metrics:

- Memory utilization and time needed for building index.
- Efficiency and efficacy during the query processing phase.

The complete source code for the project can be accessed on GitHub via the following link:

https://github.com/pieruccim/search-engine

1.1 Project Structure & Modules

The main modules of the Java project are:

- **Common**: contains the Java bean classes and managers used by the other modules like Document Index class, Vocabulary class, Posting class, and so on.
- **Preprocessing**: performs the preprocessing phase computing the cleaning, tokenization, stemming, stopword removal and stemming.
- **Indexing**: performs the indexing of the collection saving the main data structures on disk and accomplishes the merging phase.
- **Query Processing**: contains all the classes needed to performs the processing of the queries, thus Scoring Functions and Document Processor types.

2. Preprocessing

2.1 Document and query preprocessing

Each document and query is preprocessed by the class:

src.main.java.preprocessing.Preprocessor

The operations performed by this class are executed in the following order:

1. **Lowercase text conversion**, which converts each word to lowercase
2. **Bad characters removal**, which removes characters different from letters and digits
3. **Stopwords removal**, which removes common English stopwords
4. **Punctuation removal**, which removes punctuation
5. **Stemming**, which execute Porter stemming algorithm on each word to extract the root word

3. Indexing

3.1 File encoding and automatic decompression

The Indexer is capable of opening file encoded with different charset, including both **UTF-8** and **UTF-16**. If the collection is compressed in an archive, the Indexer detects it and automatically manages the file decompression.

3.2 Indexing

The process of indexing involves the generation of main data structures, specifically the inverted index, vocabulary, and document index. This procedure unfolds in two distinct stages: firstly, the Single-pass in-memory indexing (referred to as Spimi), followed by the subsequent phase known as the Merging algorithm.

3.2.1 Spimi

During the Spimi algorithm, the documents are sequentially retrieved from the collection file, pre-processed and finally indexed into partial posting lists. The Java class in charge of implementing the Spimi is:

src.main.java.indexing.Indexer

Every block of inverted index consists of two essential components: a file for containing docids within posting lists and a file for corresponding frequencies. Similarly, in each block we store partial vocabulary and partial document index. Along those files, we also create partial skipBlock files which hold the DocId/frequencies file offsets and other useful information for enabling faster query processing [see 4] .

3.2.2 Merging

The Merger algorithm analyzes the partial inverted indexes generated by Spimi, one term at a time. Whenever a new term is encountered, all the relative posting lists linked to that term across various partial inverted indexes are merged together. Eventually, the merged posting list is written on disk, using compression techniques for docid and frequency. Similarly, the merged vocabulary, merged document index and merged SkipBlocks file are written on disk.

Posting List Format

Posting lists are written to disk by using two distinct files, one for docid storage and the other for frequency. More specifically:

- *data/output/invertedIndexBlocks/docIds* → stores docIds
- *data/output/invertedIndexBlocks/freqs* → stores term frequencies

Posting lists are divided in skipBlocks. Each skipblock contains at most maxLen postings with maxLen being a configurable parameter.

SkipBlock Format

Each skipBlock is stored in:

- *data/output/skipBlocks*

and holds the following information:

Figure 3.1: SkipBlock Format

Vocabulary Format

The vocabulary is stored in:

- *data/output/vocabularyBlocks*

A single vocabulary entry follows the format in figure:

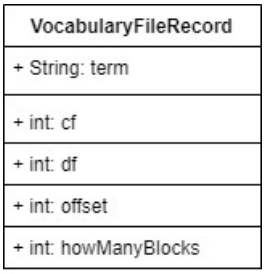

Figure 3.2: Vocabulary Format

3.3 Compression

We decided to adopt compression algorithms to reduce memory occupancy of the inverted index. In particular, we distinguish two different algorithms, depending on the integers to be compressed:

- **Unary encoding** algorithm, to compress *frequencies*
- **D-gap encoding** combined with **Variable byte encoding**, to compress *docids*

We maintain values of *frequencies* and *docids* separate, in order to adopt different compression mechanism, so in the end, we obtained two compressed files: one for *frequencies* and the other for *docids*.

3.3.1 Frequencies compression

As the frequency of a term in a single document turn out to be commonly a *small* integer, we exploit this characteristic to encode with **Unary encoding** this value. In particular, we create and store a *byte sequence* for each posting list, that contains *sequences of bits* representing frequencies encoded using Unary encoding. If the resulting sequence of bit is not byte-aligned, then a padding is added to allow Java to store the sequence in a Byte array.

Integers list: [2, 4, 1, 2, 3]

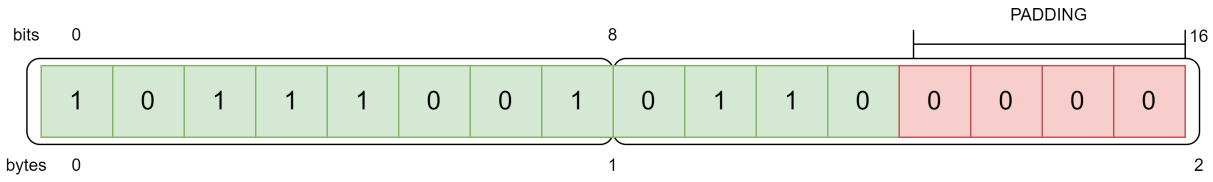

Figure 3.3: Example of *Unary* encoding

3.3.2 Docids compression

Docids are integers that uniquely identify documents. As docids are assigned subsequently during indexing, we exploit **D-gap encoding** to represent docids. In this way we store the first docid of a posting list followed by delta representations that identify next docids. In particular, the first docid and the following delta values are represented using **Variable byte encoding**, that allows us to save storage space when delta values could fit in less than 4 bytes.

Integers list: [121, 124, 175]

D-gap encoded integers: [121, 3, 51]

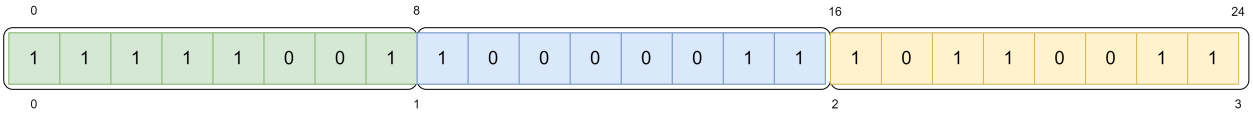

Figure 3.4: Example of *D-gap* encoding combined with *Variable byte* encoding

As shown in the example above, the combination of *D-gap* encoding and *Variable byte* encoding are particularly effective when the docids inside a posting list are near to each other.

4. Query Processing

A query is submitted by user employing textual user interface, then it is preprocessed the same way as documents to retrieve its terms. For each term it is obtained the corresponding *Vocabulary entry*, that contains information to retrieve the *Skip Blocks* data structures required to load the blocks of postings relative to the term. Once the posting list is accessible, document scoring is evaluated using TAAT, DAAT or MaxScore exploiting either TFIDF or BM25 as scoring functions.

4.1 Query workflow

Once the query terms are available, the main data structures to execute the query processing are retrieved from files stored on disk. Referring to Chapter 3, the operations executed are the followings:

1. *VocabularyFileRecord* is retrieved from the *term* of the query,
2. *VocabularyFileRecord* contains *offset* and *howManyBlocks* that allow to find the *SkipBlocks* where the posting list is located,
3. each *SkipBlock* contains *docIdFileOffset*, *docIdByteSize* and *howManyPostings* that allow to retrieve docids of the posting list, and similar fields to retrieve corresponding frequencies.

4.1.1 Posting List Iterator

The class *PostingListIterator* is in charge of loading the posting lists from disk in order to prepare the data structures to execute the query workflow. The methods available in this class are:

- ***openList***, which opens docids and frequencies *BinaryFileManagers* and loads the *SkipBlocks* using fields of the *VocabularyFileRecord*
- ***next***, which returns the next *Posting* in the iterator if any, else returns **null**, it also manages the sequence of *SkipBlocks* that compose the *Posting List*
- ***hasNext***, which returns **true** if the iterator has not reached the end or returns **false** otherwise
- ***getCurrentPosting***,which returns the current *Posting* or **null** if no posting is been read so far
- ***nextGEQ***, which returns the first *Posting* whose docId is greater or equal to the specified one, this method is used to implement skipping methods in conjunctive queries and reduce the query processing time

4.2 Document Processor and Dynamic Pruning

We implemented *Document at a time* (DAAT) and *Term at a time* (TAAT) as Document processing mechanisms. Then we propose also an approach based on *Max Score*, which uses *Dynamic pruning* and *Term upper bounds*. In particular, we pre-compute upper bounds during indexing and store them in specific binary files inside *data/output/upperBounds*. We compute two different upper bounds for each term, considering both **TFIDF** or **BM25**, in both cases we use the maximum score among postings of the same term.

4.3 Multi-threading

In order to reduce the wall clock time of execution of our search engine we implemented parallelization of some tasks in the query processing phase.

4.3.1 Parallelized loading of iterators

Inside the class *PostingListIteratorFactory* we implemented a method that receives a list of *VocabularyFileRecords* and returns the associated iterators. In our implementation every iterator is loaded by a different thread, in order to reach the maximum parallelization level.

4.3.2 Parallelized loading of blocks of postings

Inside our PostingListIterator, every time a block of postings has to be loaded, a thread is made in charge of loading the successive block, so that the main thread will not have to read data from disk.

4.4 Caching

We implemented a Key-Value *LRU* Cache with parametric maximum dimension. Our caching implementation allow to specify the type of object both for the key and the value. It allows also to provide a custom Callable that will be executed on the cached object at eviction time.

These possibilities allowed us to implement caching for different parts of our search engine.

4.4.1 PostingListIterator caching

Since certain terms are more frequently searched than others, it is very effective to cache their Posting List iterator, so that every time a query will include that term, it will not be necessary to open the file managers of its iterator again. The least recently used strategy was exploited for this caching mechanism.

This caching layer is implemented inside the class `PostingListIteratorFactory` so that it is transparent to the query processors.

4.4.2 Blocks of Postings caching

We introduced another level of caching inside each iterator; in particular it is a cache which associates each SkipBlock object with the relative docIds and frequencies arrays. We decided to add this

layer in order to reduce the amount of reads from file.

5. Performance

5.1 Indexing

Indexing Type	Duration	DocID Size	Frequencies Size	Vocabulary Size
Raw	354 min	928 MB	928 MB	29 MB
Variable Byte Compression and Unary	90 min	873 MB	39 MB	29 MB
Delta + Variable and Unary	44 min	318 MB	39 MB	29 MB

The majority of time required in the case of raw is due to the fact that entire integers are written at once, while in compression scenarios, sequences of bytes are written. The reported results are obtained using 2000 as SkipBlock dimension.

5.2 Caching

The following evaluation was conducted using TFIDF as scoring function, MaxScore as document processor, disjunctive as query type and multithreading for iterators factory.

Iterators Cache	Blocks Cache	Mean response time	Std. Dev.
Enabled	Enabled	27,96 ms	± 43,9
Enabled	Disabled	32,21 ms	± 47,1
Disabled	Enabled	45,61 ms	± 62,1
Disabled	Disabled	44,22 ms	± 66,1

5.3 Single thread vs Multi thread

We maintained the same query parameters as before, with caching enabled, and inspected the average execution time in both single thread and multi threaded fashion.

Iterators Factory Threads	Blocks Loader Threads	Mean response time	Std. Dev.
Enabled	Enabled	30,59 ms	± 46,4
Enabled	Disabled	27,96 ms	± 43,9
Disabled	Enabled	31,21 ms	± 44,9
Disabled	Disabled	29,03 ms	± 45,2

5.4 Query Comparison

The following evaluations were conducted to compare different document score types, using TFIDF as scoring function, multithread approach and cache enabled in *Disjunctive* mode. We assessed the average response time for MSMARCO development queries.

5.4.1 DAAT vs TAAT vs MaxScore

Document Score Type	Mean response time	Std. Dev.
TAAT	64,08 ms	± 56,5
DAAT	43,1 ms	± 54,3
MaxScore	27,96 ms	± 43,9

5.4.2 BM25 vs TFIDF

We now report the second performance comparison, which involves assessing the two scoring functions, BM25 and TFIDF, that we have incorporated into our search engine. In the provided table, we have reported the mean response time, mean average precision, and precision@10 achieved by employing both scoring functions. These evaluations were conducted using MaxScore as the scoring algorithm and Disjunctive as query type. The various metrics were obtained as output from the trec_eval official comparator [2] between the file qrels.dev.tsv [1] and our results.tsv containing the ranked queries (queries.dev.tsv) using our search engine.

Scoring Function	Mean response time	map@10	p@10	nDCG@10
BM25	39,05 ms	0.1382	0.0178	0.1480
TFIDF	27,96 ms	0.1136	0.0151	0.1392

5.4.3 Queries performances

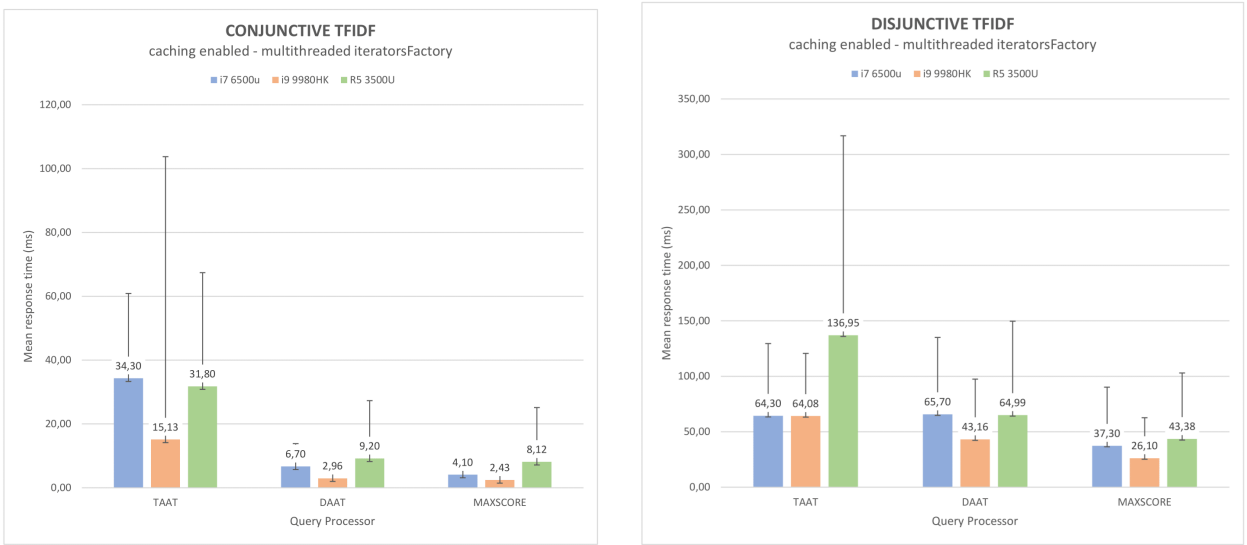

Figure 5.1: Performances on Conjunctive and Disjunctive queries

6. IR Performance Evaluation

We used Trec_eval also to assess the effectiveness of our IR system, specifically comparing its performance metrics against an other search engine.
We employed the files ”queries.dev.tsv” and ”results.tsv” which can be accessed at the following web page [1]. We used these files to assess the performance of our solution in comparison to an open source search engine, namely ”Terrier search engine”. The results of the comparison are reported below:

Metrics	Terrier	Our solution
map	0.1929	0.1382
nDCG@10	0.2290	0.1480
MRR	0.1962	0.1419

As we can see from the table we get comparable results.

7. Limitations & Possible Improvements

7.1 Vocabulary handling

Our implementation loads the whole vocabulary in memory at start up, which is a limitation since if the vocabulary size becomes too high, it would require too much memory space.

For this reason an important improvement could be to implement a system that loads from the vocabulary file only a part of the vocabulary, following a criterion based on the frequency of search of the terms, so that the terms that are more frequently used are loaded in memory at start up.
For the vocabulary, it would be effective also to cache the least recently used terms, so that they are temporarily stored in memory.

7.2 Multi-Threading in Iterators

In our implementation each posting list iterator has at most one thread in execution which is in charge to read from the inverted index the next block of postings.
Every time a block is read the thread is terminated and another thread is instantiated to load the next block.

This operation introduces overhead and, for this reason, increases the execution time instead of reducing it.
An interesting improvement would be to modify the thread workflow so that each iterators' thread is in charge of loading an amount of blocks of postings and storing each block in its iterator blocks cache.

In this scenario there would be the necessity to implement a communication mechanism between the iterator main thread and the iterator loader thread, so that the main thread could be able to tell to the loader which block is accessing (and, consequently, which will be requested in the future).

The loader thread would also be in charge of storing the loaded blocks into the cache, while the main thread could loads the blocks from that cache.

7. References

[1] TREC 2020 Deep Learning Track Guidelines. https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020.

[2] TrecEval Comparator. https://trec.nist.gov/treceval/.