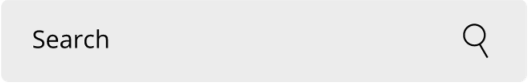
University of Pisa
Artificial Intelligence and Data Engineering

Multimedia Information Retrieval and Computer Vision

Search Engine



Supervisors
Nicola Tonellotto

Students
Martina Burgisi
Gabriele Marino
Matteo Pasqualetti

Contents

Chapter 1

Introduction

1.1 Description

This academic endeavor aims to develop a search engine capable of text retrieval across a vast collection of 8.8 million documents. The project involves two primary phases:

- **Document Indexing:** In this initial step, we focus on creating essential data structures required for efficient retrieval.
- **Query Processing:** The subsequent phase involves the actual retrieval of pertinent documents based on user queries.

Following a detailed explanation of these step, we will conduct a performance analysis, assessing memory usage and building time for the indexing phase, and evaluating efficiency and effectiveness for query processing.

The project's source code is accessible on Github via the following link:

https://github.com/gabrielemarino-gm/Search-Engine-MIRCV

1.2 Organization

Main packages and modules of our project are:

- **Driver:** Permits to interact with the all the aspects of the Search Enging via a CLI.
- **Algorithms:** Contains all the useful algorithms for the developement and usage of the index like *SPIMI*, *DAAT*, *Max Score* and *Conjunctive Ranked Retrieval*;
- **Model:** Contains all the Data Object like *InvertedIndex*, *PostingList*, *Cache*, *Document*, or *Vocabulary*;
- **Utils:** Contains supportive or optional classes like *FileManager*, *ConfigReader*, *Preprocesser*, *Compressor* or *ScoreFunction*.

Chapter 2

Indexing

2.1 Preprocessing

The class which implements this functionality is:

it.unipi.aide.utils.Preprocesser

Cleaning

In this phase any document (or query) is cleaned by using the following RegEx:

1. `URL Pattern: (https?://\S+|www\.\S+)`
2. `HTML Pattern: <[^>]+>`
3. `Non Digits: [^a-zA-Z ]`
4. `Consecutive Letters: (.)\1{2,}`
5. `Camel Case: (?<=[a-z])(?=[A-Z])`

Tokenization and Text Normalisation

In this step, the cleaned string is split into single words in the following way:

[This is A SamplE String] ⇒ [this, is, a, sample, string]

Stopword and Stemming

In this optional step, stopwords are removed by using a template file inside the *resource* folder. Furthermore, stemming is applied by using a *Porter Stemmer*.

2.2 SPIMI

The following class implements the first phase of the *Single-Pass In-Memory Indexing Algorithm*, implemented by:

it.unipi.aide.algorithm.SPIMI

During it's execution, it uses the **Corpus** class to read the corpus file. It will automatically handle any **.tsv* and **.tar.gz* file. It proceeds than by indexing every document one by one. To be memory friendly, it write a partial **Inverted Index** on the disk when the memory reaches a certain threshold. A **Document Index** is also generated, containing some information for each new document.

All the *docid* are assigned in an incremental way to each document.

Additionally, to allow the usage of MaxScore algorithms, TF and BM25 upperbounds are calculated for each term:

- **TF**: Simple as taking the max $< tf_i >$ for each term in each block. The max in all blocks will be used later in Merging phase.
- **BM25**: Calculated by memorizing the pair $< tf_i, dl_i >$ that maximize an approximation formula. The pair that maximize the following formula will be also selected in Merging phase:

$$\text{Approximated BM25}(tf_i, \text{dl}_i) = \frac{tf_i}{tf_i + k_1 \cdot (1 - b + b \cdot \text{dl}_i)} \tag{2.1}$$

Even without the *avdl* term, this formula allows us to calculate this upperbound in a faster way, without iterating the posting list again in the successive phase.

2.3 Merging

The following class implements the second phase of the *Single-Pass In-Memory Indexing Algorithm*, implemented by:

it.unipi.aide.algorithm.Merging

As *SPIMI* ends it's execution, this class begins to merge terms one by one in different blocks to create one entire *Inverted Index* on the disk. Thanks to the fact that *docid*s are ordered in an increasing order, it only consists on a concatenation. Posting lists are stored in two separate files:

$$data\backslash out\backslash docids \rightarrow \text{Contains DocIDs}$$

$$data\backslash out\backslash frequencies \rightarrow \text{Contains Frequencies}$$

Block Division

To implement the **Skipping** feature, during this phase, we also implement the block-wise division of various posting lists, crucial for improve the efficiency of retrieving algorithms. The length of each block is determined by $\sqrt{n}$, where n represents the number of postings within that posting list. Blocks are created only if $n > 512$.

The information required for skipping is stored in a file, with entries structured as follows:

- `max DocID`: Maximum Document ID within the block.
- `number of Postings`: Count of postings in the block.
- `offset DocID`: Offset of the first DocID.
- `offset Frequencies`: Offset for the first Frequency.
- `bytes DocID`: Number of bytes occupied by all the DocIDs in that block.
- `bytes Frequencies`: Number of bytes occupied by all the Frequencies in that block.

This approach ensures an organized and efficient merging process while facilitating the implementation of skipping mechanisms for improved retrieval performance.

Optional

On the configuration file, it is possible to choice to enable **Compression** and or **Block Division**.

2.4 Compression

The Compressor class implemented by:

it.unipi.aide.algorithm.Compressor

It provides methods to apply different types of compression. A dual compression strategy has been implemented, employing Variable Byte and Unary techniques to accommodate the distinctive features of frequency and docid.

Unary for Frequencies

Unary encoding was selected for frequencies due to its effectiveness in handling relatively small numeric values. This choice enables a concise representation of frequency data.
More than 90% of Postings has a frequency value of 8 or less, which allows us to compress the frequencies with a 95% compression ratio.

Variable Byte for DocId

Variable Byte encoding was chosen for docids, considering its capacity to represent sequences of consecutive number, making it well-suited for larger values.

Chapter 3

Query Processing

Before becoming ready to handle queries, the system loads into memory a file containing the document lengths of all documents in the collection. To expedite the reading and loading of this file, the decision was made to compress it using the Variable Byte algorithm.

Once the system is prepared, and after the user selects preferences regarding which algorithm to use for query processing, which score function to employ, and how many documents to display in the results, it can commence serving all incoming queries. For each query, the system begins loading into memory the vocabulary entries related to the query terms (using binary search) and then loads only the first block of each posting list for the query terms. Afterward, it executes the algorithm chosen by the user.

3.1 Data Structures

3.1.1 Vocabulary

Vocabulary entries consist of the following fields, for a total of 74 bytes:

- `term`: term, bounded on 46 Bytes;
- `totalFrequency`: total frequency of the term, integer on 4 bytes;
- `numPosting`: number of posting lists, integer on 4 bytes;
- `numBlocks`: number of blocks, integer on 4 bytes;
- `offset`: offset of the term, long integer on 8 bytes;
- `termUpperboundTFIDF`: upperbound of the term for the TF-IDF model, float on 4 bytes;
- `TermUpperboundBM25`: upperbound of the term for the BM25 model, float on 4 bytes.

3.1.2 Posting List

To handle every term posting list two Java classes has been created:

it.unipi.aide.model.PostingList

and

it.unipi.aide.model.PostingListSkippable

The last one has been developed to efficiently support postings retrieving from disk with the usage of block-based skipping for improved performance.

The design of the class revolves around some of the following key components:

- **Block Descriptors**: The class utilizes block descriptors to efficiently locate and retrieve blocks of postings from disk. These descriptors contain information such as the maximum docID, the number of postings in the block, and the offsets for docID and frequency.
- **Compression Support**: The class supports both compressed and uncompressed posting lists. Depending on the configuration, it employs compression techniques to reduce the amount of data read from disk, enhancing overall performance.

In the PostingListSkippable class the **next** and **nextGEQ** methods has been implemented to iterate through posting lists and efficiently searching for specific posting (and consequently, documents) within those lists.

- **next()**: This function advances the iterator to the next position in the posting list, returning the next available posting.
- **nextGEQ()**: Taking a docID parameter, nextGEQ moves the iterator to the position of the first posting with a docID greater than or equal to the specified one.

3.2 Algorithms

3.2.1 Disjunctive Ranked Retrieval

Two types of algorithm has been developed to support disjunctive retrieval. Both of them supports two scoring models: TF-IDF and BM25.

DAAT

The Java class involved is:

it.unipi.aide.algorithm.DAAT

The DAAT class provides an implementation of a retrieval algorithm for processing disjunctive queries over the inverted index. It utilizes the Document-At-A-Time processing strategy, scoring and ranking documents based on the given query terms.

MaxScore

The Java class:

it.unipi.aide.algorithm.MaxScore

is designed to implement the MaxScore algorithm. This algorithm combines essential and non-essential posting lists to optimize the scoring process.

3.2.2 Conjunctive Ranked Retrieval

The ConjunctiveRetrieval Java class:

it.unipi.aide.algorithm.ConjunctiveRetrieval

is designed to implement a Conjunctive Ranked Retrieval algorithm, which retrieves the top-k scored documents satisfying the conjunctive conditions specified in a query. The algorithm combines posting lists to identify documents that match all query terms. It supports both BM25 and TF-IDF scoring functions for document ranking.

3.3 Cache

The involved class is:

it.unipi.aide.model.Cache

In our system, we have integrated an LRUCache that extends the functionality of the Java LinkedHashMap class. This LRUCache is designed to cater to three specific caching scenarios, contributing to a more efficient and responsive system.

- **L1 Cache: PostingListSkippable Object Caching**
  The L1 cache specializes in storing PostingListSkippable objects, eliminating the need to retrieve the initial block from the disk. This targeted caching strategy optimizes the system's performance by ensuring that frequently accessed PostingListSkippable objects are readily available in memory.

- **L2 Cache: TermInfo Object Caching**
  Our L2 cache is dedicated to caching TermInfo objects, minimizing the necessity of fetching them from the disk. This approach enhances system responsiveness, particularly in scenarios where repeated queries for TermInfo objects occur. By proactively caching these objects, we streamline data retrieval and reduce latency.

- **L3 Cache: Binary Search Path Optimization**
  The L3 cache optimizes binary search operations by storing half of the paths traversed by terms. This selective caching strategy focuses on the commonly traversed sections of the search path. Storing the midpoint of the path allows us to avoid frequent disk accesses during binary searches, thereby improving the efficiency of the search process for terms.

Chapter 4

Performance

4.1 Index Creation

In the following table index creation performances and statics has been noted to compare different type of building mode. Specifically, "Preprocessed" label indicate for simplicity the case in which the stemming and stopword flag has been selected.

Building mode	Time	DocIds	Frequencies	Vocabulary
Uncompressed, Raw	29.34 min	1.36 GB	1.36 GB	81.9 MB
Compressed, Raw	30.12 min	1.28 GB	63 MB	81.9 MB
Uncompressed, Preprocessed	26.32 min	714.9 MB	714.9 MB	67.5 MB
Compressed, Preprocessed	26.14 min	672.5 MB	31.5 MB	67.5 MB

Table 4.1: Index Creation performances

4.2 Query

The following scores have been computed considering the Compressed index with Stemming and Stopword removal. We used 200 queries to test the average response time and the variance for all the retrieval algorithms, and TREC DL 2020 to evaluate system performances.

4.2.1 Disjunctive Algorithms Comparison

Algorithm	Average response time	Variance
DAAT	68.25 ms	2747.69
MaxScore	29.27 ms	374.51

Table 4.2: Comparison between DAAT and MaxScore algorithms.

4.2.2 Conjunctive Ranked Retrieval

	Average response time	Variance
Conjunctive Ranked Retrieval	16.87 ms	193.41

Table 4.3: Conjunctive Ranked Retrieval statistics.

4.3 Overview with boxplots

The following plots were generated using the same set of 200 queries employed to evaluate system performance. The purpose is to illustrate the elapsed time required by the retrieval algorithms to respond to a query with an increasing number of tokens.

4.3.1 DAAT

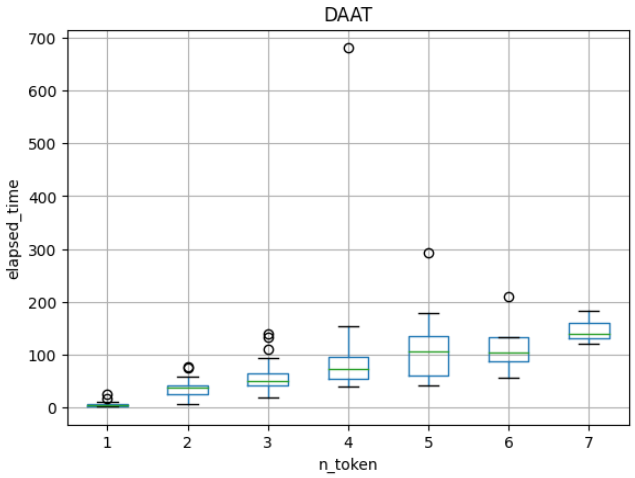


Figure 4.1: DAAT blox plots grouped by number of tokens.

4.3.2 MaxScore

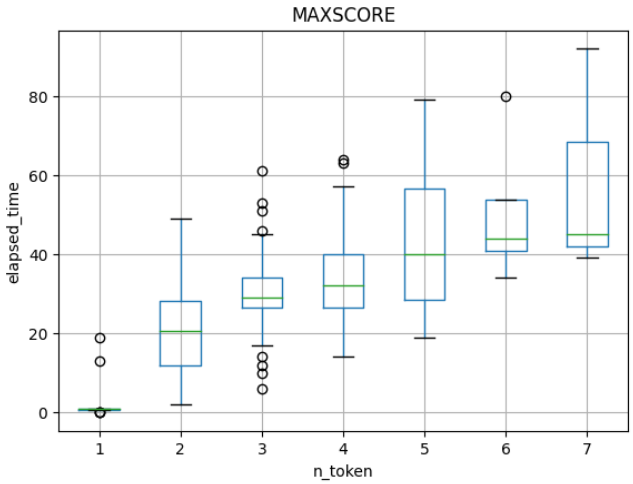


Figure 4.2: MaxScore blox plots grouped by number of tokens.

4.3.3 Conjunctive

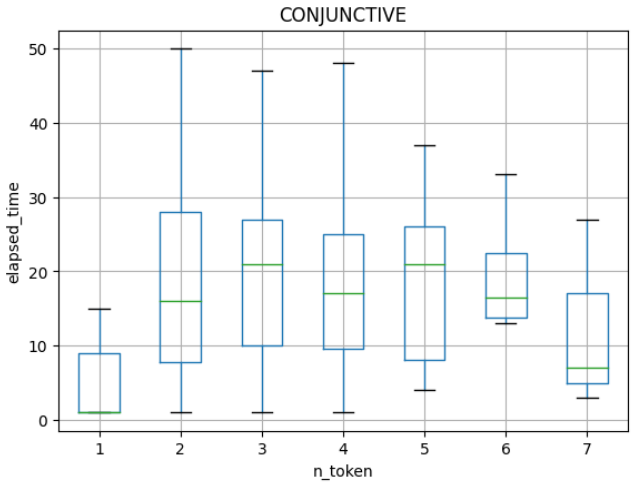


Figure 4.3: Conjunctive blox plots grouped by number of tokens.

4.4 Quality measures

	MAP_cut_10	**nDGC_cut_10**	**P@10**	**P@20**	**R@1000**
Disjunctive - TFIDF	0.1199 ms	0.4299	0.5296	0.4620	0.7259
Disjunctive - BM25	0.1357 ms	0.4678	0.5648	0.4759	0.7385
Conjunctive - TFIDF	0.1171 ms	0.4125	0.4674	0.3293	0.3326
Conjunctive - BM25	0.1144 ms	0.4095	0.4717	0.3315	0.3332

Table 4.4: Quality measures

Chapter 5

Limitations

- *docid*s compression may be improved as Variable-Byte is almost useless if raw used: it allows us to save only 2.1MB on almost 700MB; as Block division is enabled, a Listing method like GAPS should help a lot;
- To reduce BM25 execution, *doclens* are pre-compressed during index creation and loaded before makying any query. This is duable only if the collection size is contained. It may be enhanched by implementing a block division;
- BM25 and Blocking parameters where not evaluated, so they are not optimal;
- Caching of *TermInfo* or *PostingListSkippable* only reduces query executions, but not considerably high; query results or entire *Postings* could be cached to further improve performance;
- Most of the code is not optimized, which extends for further improvements;
- Compression, Stemming and Upperbounds requires to create the Index again from scratch