Search Engine

Multimedia Information Retrieval and Computer Vision

Matteo Manni
Lorenzo Mazzei
Salvatore Patisso

Summary

Abstract

Search Engines are one of the end goals in Information Retrieval and they have been a topic of research for many years. Different approaches can be carried out in order to build a Search Engine: the focus of this project is to build one which meets the most critical requirements in order to be considered valid. Each component of our Search Engine was designed and optimized according to some well known techniques to assure effectiveness and fast response. We deployed it in Java - JDK21.

1. Index

The size of the collection(s) we had to take into account was not trivial. The Search Engine was designed to be adaptable to real world data and therefore it had to be able to deal with millions of documents. Given this premise, the Inverted Index data structure needed to process the future queries of the users had to be built in a Scalable approach. This means that we couldn't assume that the whole collection of documents could fit in memory, so we processed it using the SPIMI algorithm: we indexed the documents one at a time until the memory was filled, we then stored the data structures constructed until now into the disk and we emptied the memory to process the next bunch of documents.

Before indexing a document, a pre-processing is carried out on it:

1. Tokenization
    a. HTML tag removal.
    b. Punctuation removal.
    c. Bring words to lowercase.
    d. Split on " ".
2. Optional stopword removal (by using a sorted list of stopwords, on which binary search is performed for each token).
3. Optional stemming (by using a PorterStemmer class).

Compressed Reading

We made the Search Engine able to load the collection file in memory in its compressed version (in our case tar.gz) and process it by storing the compressed data in a buffer in memory. Once decompressed, the way each document is processed is the same as the uncompressed reading version.

1.1 SPIMI

We used a threshold for the memory used during the processing of a bunch of documents by SPIMI: when the remaining free memory percentage became less than the chosen threshold, we proceeded with the next bunch of documents. We tested different values for the threshold and it turned out that the best compromise was at 8%.

The output of the SPIMI algorithm resulted in a series of blocks of data on the disk, therefore each of them containing a portion of the posting lists of the terms that occurred in that block. To get the final inverted index structure, we had to merge these blocks in order to collect the complete posting lists of each term that occurred in the collection.

1.2 Merge

Since we had to deal with multiple files, written in binary, we approached this step by opening in parallel all the files: from the SPIMI algorithm we used in the indexing step, we could leverage on an important property of all these files, the fact that all the terms in a file are ordered lexicographically. This allows us to deal with the files in the following way: we process the terms one at a time, more specifically we look for the minimum term in lexicographic order between all the ones we are currently at in all the files, in this way we are guaranteed to scan all the instances of that term in the files in the same moment: this procedure is handled with a Priority Queue data structure, which allows us to retrieve the minimum term more efficiently w.r.t using a simple array data structure.

Therefore, we then retrieve all the partial posting lists for that term and we merge them into the full posting list. By carrying out this procedure until all the files are processed, we get as output the final inverted index which is stored in a binary file on the disk.

During the merging phase we also compute additional values (such as the Term Upper Bounds) and statistics for the Document Index (such as the average document length) which will be used during the query processing.

** After the merging phase the blocks created in SPIMI could be deleted. The program actually doesn't delete them for debugging purposes, but should do it in a final version.

2. Query Processing

The main 2 paradigms we could use to process the queries of the users were Term At A Time (**TAAT**) and Document At A Time (**DAAT**). DAAT was clearly the better choice to get a faster response from the system. Both Conjunctive and Disjunctive DAAT were developed as a way to process a query. The results of each query are shown to the user in the form of DocNo associated with the returned documents.

- **DISJUNCTIVE**: we scan all the posting lists in parallel and process the document with minimum docID at each iteration. We use a MinHeap data structure to store the scores and the associated docIDs that satisfy the score threshold.

- **CONJUNCTIVE**: we first sort the posting lists with respect to their length (document frequency); at each iteration the current docID of the first (i.e. the shortest) posting list is searched in the other posting lists, we only process the document if it is found in all the posting lists. The MinHeap is used here as well.

MinHeap Implementation: the "*MinHeapScores*" class was used to implement this data-structure used in the query processing to efficiently implement the ranked retrieval and maintain only the top k result found. This class represents a priority queue of scores that, thanks to an HashMap (k:score v:array[docIDs]) that maintains associations between each score and all docIDs with that score, allows to keep sorted docIDs in this queue while also not letting docIDs with a score lower than the MinHeap threshold enter the queue.

2.1 Score Function

We implemented 2 metrics for the scores of the documents w.r.t the queries: **TFIDF** and **BM25**. TFIDF was less complex to deal with, since it depends only on the document frequency and the term frequency: the document frequency (i.e. the length of the posting list associated with that term) is retrieved from the file just once, at the start of the query processing, so no other access to the files has to be made.

BM25 instead depends also on the document length, which requires an access to file at each iteration of the query processing. This clearly impacts negatively on the response time for every query, so we assumed that all the document lengths could fit in main memory: since we used 4 bytes to store each document length, even when dealing with a huge collection storing each document length in memory appeared to be feasible (in our case the number of documents is about 8.8million, approximately 36MB). We proceeded this way to optimize this problem.

2.2 Optimizations

To process the queries in the fastest way possible, optimizations were used both for Conjunctive and Disjunctive mode. More specifically:

- **Skipping**: additional data structures called skip descriptors were developed to allow the skipping of portions of the posting lists during their scanning, this feature was used every time we could be sure that some of the docIDs of the posting list we were working on wouldn't have had an impact on the final result of the query. Skipping was therefore used both in Conjunctive mode and in MaxScore. Skip descriptors are represented as an arraylist of maximum docIDs of the posting list blocks. They are stored in a file and loaded in main memory to improve the performance of the system when the programs perform query processing.
  The size of a skipping block is equal to the square root of the posting list length.

- **MaxScore**: algorithm that allows us to skip the processing of DocIDs which we are sure won't make it into the final results. To carry on this algorithm we first needed the Term Upper Bound Scores, i.e. for each term, the maximum score a docID belonging to that term's posting list can have. This additional information is computed during the Merge phase, i.e. when we retrieve the full posting list for each term. MaxScore is used for the Disjunctive mode and it implements dynamic pruning.

- **Cache**: we haven't implemented a caching system because executing the same query multiple times consecutively allows for an implicit caching system to significantly reduce the execution time.

2.3 Scalability

Posting List Blocks

To make the Search Engine scalable, we couldn't assume that we could store whole posting lists in memory. Therefore, we handled each of them by loading portions of the posting lists (Blocks) in memory. The dimension of a block is fixed and chosen by us beforehand in the Config class. We created a class called PostingListBlock which represents a portion of a posting list. This class acts also as an iterator of the posting list blocks by implementing methods like "next()" or "currentPosition()" "currentDocId()".

3. Compression

We made the Search Engine able to be executed both with and without compression. The compression methods we used were Variable Byte and Unary:

- **Variable Byte**: used for the DocIDs of the elements of the posting lists

- **Unary**: used for the term frequencies of the elements of the posting lists

Since we used 2 different methods, we also had to keep 2 different offset for each posting list: one for the docIDs and one for the term frequencies. Since we compress and decompress by blocks, these offsets actually point to the start of the block, whose size is set to be equal to the square root of the length of the posting list.

During the loading of a new block of a posting list in memory, if the Compression flag is set, we first need to decompress that block and then we can actually load it in memory.

We implemented compression by creating an alternative version of every class who was needed to change for allowing compressed writing and reading like:

- PostingList → PostingList2
- BlockMerger → BlockMergerCompression
- SkipDescriptor → SkipDescriptorCompression
- DisjunctiveDAAT → DisjunctiveDAATCompression
- ConjuctiveDAAT → ConjunctiveDAATCompression
- MaxScore → MaxScoreCompression

In file handlers classes were added specific methods for compressed reading and writing.

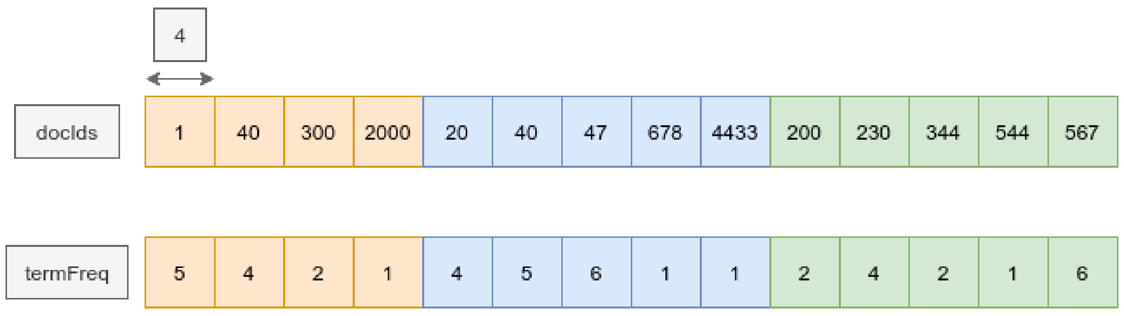
4. File Organization

The files stored on disk were handled as binary files. In the graphics below the structures of the files are reported with the corresponding number of bytes used to represent each field.

Posting List Files

docIds: files containing the document IDs

termFreq: files containing the term frequencies, i.e. the occurrences of the term in the documents

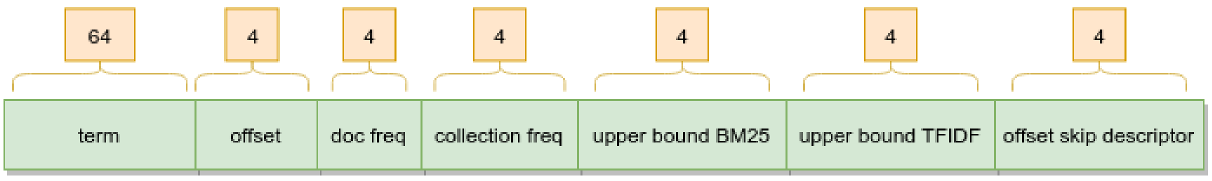


In this case to refer to these arrays we use a single offset, the second field in each lexicon entry, since a docID is 4 byte long as well as a term frequency.

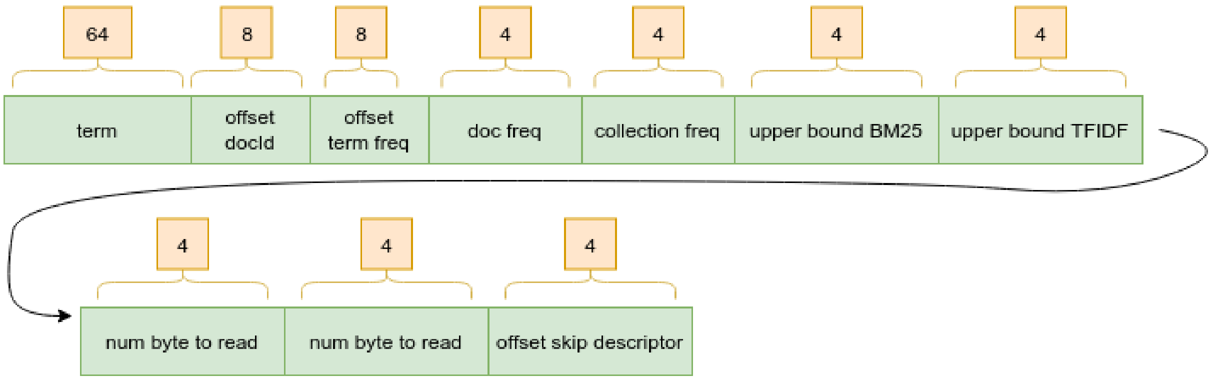
Posting List Files (Compressed version)

In this case the 2 arrays are organized as in the graphic above, the difference stands in the fact that to refer to these arrays we actually use 2 different offsets one for the docIDs and one for the term frequencies, since we use 2 different methods to compress them (respectively Variable Byte and Unary). As mentioned while discussing the compression, these offsets point to the start of the skipping block that has to be decompressed and loaded in memory.

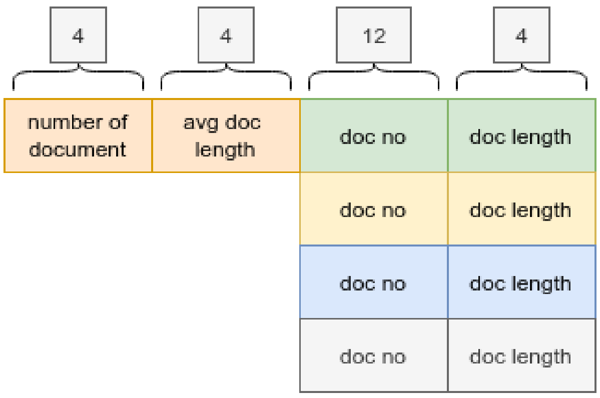
Lexicon File



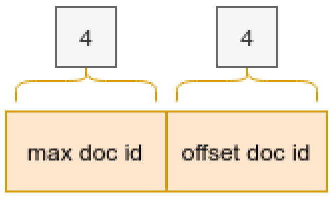
Lexicon File (Compressed version)



Document Index

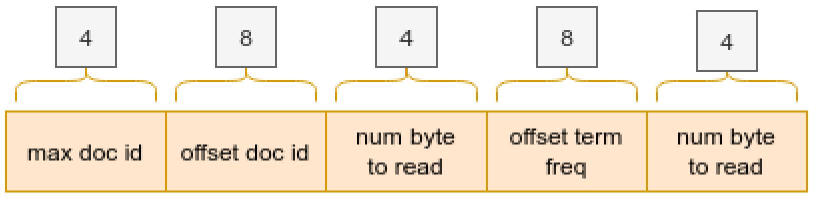


Skipping Descriptor



Each offset refers to the position of the maxDocID of each skipping block for the posting list of the associated term. These data are used for the skipping optimization during query processing (i.e. nextGEQ).

Skipping Descriptor (Compressed Version)



The length in byte of every element that is stored in these files is fixed and chosen by us beforehand. These values are stored in a Config class so that we have a parameter for everything, this allows us to test different values easily during the development of the project.

There is a class for every index data structure: "*documentIndexFileHandler*" , "*invertedIndexFileHandler*", "*skipDescriptorFileHandler*" and "*lexiconFileHandler*".

"*InvertedIndexFileHandler*" can be seen as a Posting List file handler and it manages the I/O operation from the two files (docIDs and term frequencies) on which the inverted index is split.

Each of these classes make use of the "*RandomAccessFile*" class with the "*FileChannel* " abstract class to access binary files in a Random-Access mode. These classes handle all the I/O operation by specifying and managing the "offsets" in the file using fixed data buffer dimensions written in the Config class.

5. User Interface & Tests

5.1 Command Line Interface

A simple CLI was developed to allow the user to interact with the Search Engine. The lists of command available are shown when the CLI is started, together with the explanation on how to utilize them, highlighting also the parameters to choose and the possibility to set the flags for Compression, Compressed Reading, StopWord removal, Stemming, ScoreFunction and the mode for query processing (i.e. Disjunctive or Conjunctive).

5.2 Tests

The main aspects of the project were tested by using test units:

- **Compressed Reading**: we checked that the data read in the uncompressed version is the same as the compressed reading.
- **Compression**: we checked that the encoding and decoding of both Unary and Variable Byte worked consistently, also queries with compression enabled were executed to see if the results were the same as the uncompressed version.
- **Index**: a test collection crafted by us was used to perform the SPIMI algorithm manually, we then compared the results obtained by us with the one obtained by the SPIMI method we designed to see if they were the same.
- **Queries**: we crafted a test collection on which to perform the queries, we compared the result of each method w.r.t their optimized version (e.g Disjunctive w.r.t MaxScore, Conjunctive with and without skipping etc…).
- **Scores**: we crafted a test collection (very few documents) on which to compute the scores by hand both for TFIDF and BM25 given some queries. We checked that the computed results were the same as the one collected at the end of the query processing, from the MinHeap data structure.
- **Utility methods**: some of the methods used during query processing were also tested, such as the nextGEQ, the loading of posting list blocks, sorting of the arrays for Disjunctive and Conjunctive mode.

6. Performance

The Search Engine performance was tested on the MSMARCO collection, specifically the indexing was performed on the collection.tar.gz file, and query evaluation was carried out on the msmarco-test2020-queries.tsv.gz and 2020qrels-pass.txt files. The trec_eval program was used to compute system effectiveness metrics.

6.1 Indexing

Two metrics were chosen to evaluate indexing:

- The time required to create the complete index.
- The disk space occupied by the index.

The table below presents the results for each file that composes the index. The results correspond to the index with stemming and stop word removal.

Compression	Time (min)	documentIndex (MB)	lexicon (MB)	postingListDesc (MB)	docIds (MB)	termFreq (MB)
False	12	138	328	5	898	898
True	13	138	403	18	844	40

Regarding the index with compression compared to the one without, a notable decrease in overall space usage is evident: 1.443 GB against 2.267 GB. The size has decreased by approximately 36.3%. However, there is also an increase in computational time for indexing by approximately 8.3%. What is clear from the results is that the compression impact on the Term Frequencies file is significant, unlike that of the Doc IDs.
This outcome comes from the high number of documents within the collection. Variable Byte compression is effective for compressing integers representable in binary with 21 bits. If 22 bits are required, then the algorithm will compress the integer into 4 bytes. This implies that there won't be a space advantage but rather a computational disadvantage. Unfortunately, 2^21 is equal to 2’097’152 and there are over 8.8 million documents. This is why Variable Byte, in our case, is not particularly effective.

6.2 Query

The system was evaluated for Efficiency and Effectiveness using 200 test queries.

6.2.1 Efficiency

To represent efficiency, we utilized the average response time (ms) of the system for each of the queries. All the different methods of query processing that are implemented were tested (Warm Cache).

Compression	Disjunctive DAAT	Disjunctive MaxScore	Conjunctive DAAT
False	21.49	12.73	5.98
True	27.54	17.91	9.015

As expected, we noticed that using Disjunctive-MaxScore instead of Disjunctive-DAAT resulted in a percentage decrease of average response time by 40.7% with compression and 35% without compression. Conjunctive-DAAT processing is significantly faster compared to Disjunctive-DAAT. This is evidently due to the fact that the documents to be examined are a subset of those considered in the Disjunctive mode.

6.2.2 Effectiveness

The system's effectiveness was evaluated using the following metrics: nDCG, MAP, MRR, and Recall.

BM25:

	nDCG@10	nDCG@100	nDCG@1000	MAP@10	MAP@100
Disjunctive	0.4612	0.3695	0.3255	0.1390	0.1800
Conjunctive	0.3921	0.2575	0.2558	0.1138	0.1446

TFIDF:

	nDCG@10	nDCG@100	nDCG@1000	MAP@10	MAP@100
Disjunctive	0.4563	0.3612	0.3258	0.1297	0.1751
Conjunctive	0.3903	0.2575	0.2558	0.1155	0.1446

BM25:

	Recall@100	Recall@1000	MRR@10	MRR@100	MRR@1000
Disjunctive	0.4866	0.7316	0.7236	0.3402	0.0988
Conjunctive	0.2039	0.2130	0.7453	0.6674	0.6669

TFIDF:

	Recall@100	Recall@1000	MRR@10	MRR@100	MRR@1000
Disjunctive	0.4671	0.7202	0.7501	0.3539	0.0956
Conjunctive	0.2039	0.2130	0.7812	0.6677	0.6672

The metrics nDCG@X and MAP@X provide an indication of the 'quality' of the top X returned documents. When compared to some results and baselines founded online, we consider ourselves satisfied with these two metrics.

BM25 seems to perform better than TF-IDF as it excels in all metrics. To confirm this, we conducted the Wilcoxon statistical test on the 'nDCG@10-Disjunctive' metric. Unfortunately, with a p-value of 0.68, we're unable to reject the null hypothesis. It is likely that more test queries are needed to achieve a statistical difference between the two distributions.

Regarding Recall, it's evident that in Disjunctive mode, a significantly higher number of relevant documents is captured compared to Conjunctive mode.

The metric MRR@X is compromised. It can happen that there are fewer than X documents returned by the search engine for a query, especially in Conjunctive mode. If no document returned by the Search Engine is relevant, then RR = 1/0. trec_eval handles this exception by excluding the result from the average. That's why Conjunctive stands out compared to Disjunctive in the case of @1000. What we can assert with certainty is that if there exists a relevant document among the top 10, on average, we find it between the first and second positions (1/0.72 = 1.38). It's not a strong conclusion, but we decided to report it.

7. Limitations & Further Improvements

The Search Engine we designed has some limitations in some areas:

- **Document Length in Main Memory**. As mentioned before, we assume that we can store all the document lengths in memory, which, if we take into account collections far greater than ours, could not be feasible, therefore becoming a problem in Scalability. In this case, to prevent performance from deteriorating too much, we could implement a cache with document lengths instead of reading them from the file every time. The rest of the implementations, however, are perfectly scalable.

- **Code Organization**. The code can also be organized differently to implement a better readability, by using abstract classes since the methods for query processing share most of the constructors' code; this also applies to methods for the compressed version of the Search Engine.

- **Compression**. For compressing the docIDs, we've observed that Variable Byte isn't optimal. Exploring other compression types could be beneficial. Given that we compress and decompress the docIDs never individually but rather as lists, exploring integer list compression algorithms (such as Elias-Fano or PForDelta) might be effective.

- **Optimize Query with Compression Enabled**. Another area that could be improved regards skipping with compression enabled: in this case we set the size of the block of a posting list equal to the size of the skipping block, i.e. the square root of the posting list length. This makes the processing of a posting list slower than it could be if we would handle these two sizes independently.

- **Re-Rank**. Machine Learning could become a feature of our Search Engine to align it to the state-of-the-art approaches, therefore making the ranking of the documents more efficient.