University of Pisa

Multimedia Information Retrieval And Computer Vision

Information Retrieval Project

Angelo De Marco
Daniele Canzoneri
Marco Pardini

A.Y. 2023-2024

Contents

Chapter 1

Introduction

"Information retrieval is a field concerned with the structure, analysis, organisation, storage, searching, and retrieval of information" (Salton, 1968).

The objective of this project will be the creation of an information retrieval system and all the related structures (vocabulary, document index and posting lists).

The document collection indexed is the MS MARCO passage collection (8.8M documents, 2.9 GB), available at: `https://microsoft.github.io/msmarco/Datasets.html`

Link to the GitHub page of the project: `https://github.com/Sgazzirro/MIRCVProject`

1.1 Project structure & Main modules

The system is made up of 2 main components: the indexing component and the querying component.

- The indexing component will be the one in charge of indexing the document collection, producing the data structures necessary to the subsequent task of fast retrieval of information.
- The querying component receives queries from the user that are looking for a specific information, and uses the index to find the most relevant documents, that are returned in relevance order.

1.1.1 Vocabulary

In our Vocabulary implementation we have only a field, which is a NavigableMap¡String, VocabularyEntry¿. Let's take a look at the VocabularyEntry:

- private int documentFrequency;
- private double upperBound;
- private long docIdsOffset;
- private long termFreqOffset;
- private int docIdsLength;
- private int termFreqLength;

- private PostingList postingList;

In our vocabulary entry we decided to store the document frequency === posting list length, the upper bound for MaxScore implementation, docIdsOffset, docIdsLength, termFrequencyOffset and termFrequency length to fetch the compressed posting list from file.

1.1.2 Posting list

We have 2 implementations of posting lists, to handle the compressed case and the uncompressed case. The uncompressed case is implemented in the class PostingListImpl, and it has the following fields:

- private List¡Integer¿ docIdsDecompressedList
- private List¡Integer¿ termFrequenciesDecompressedList
- private int pointer = -1

Since this is only for testing purposes, we don't have any implementation of blocks nor skipping, and therefore the nextGEQ(int docId) function will iteratively pass all the list. In the compressed case we have much more complexity. The idea is that we're going to have 2 byte arrays that contain the compressed lists of doc ids and term frequencies and 2 list of integers that instead contain the decompressed version of the block in which we're in only. It is implemented in the class PostingListCompressedImpl and it has the following fields:

- private List¡Integer¿ docIdsDecompressedList
- private List¡Integer¿ termFrequenciesDecompressedList
- private int pointer;
- private byte[] compressedDocIds;
- private byte[] compressedTermFrequencies;
- private List¡Integer¿ docIdsDecompressedList;
- private List¡Integer¿ termFrequenciesDecompressedList;
- private int blockPointer; // offset in the block
- private long docIdsBlockPointer; // offset in the byte array of the next block
- private long termFrequenciesBlockPointer; // offset in the byte array of the next block

Every time we exceed the decompressed list in the block with a next() or nextGEQ(int docId) call, we fetch the next block from the byte array, we decode it and we store it in the list, setting to 0 the blockPointer and updating the docIdsBlockPointer and termFrequenciesBlockPointer to the next block offset in the list.

The nextGEQ(int docId) function is implemented in the following way: For each block we print on file the upper bound and the list size before the encoded list, that will be equal to the block size for each block apart from the last one, that will be spurious.

Since the docIdsBlockPointer points directly to the couple upperBound and listSize of the next block, we just need to read the first value to know if the block will contain the docId we're searching for.

This way, we'll decode only the block that has the possibility to contain that docId, skipping all the others.

1.1.3 Document Index

Our document index implementation is composed by 3 fields: totalLength, numDocuments and a NavigableMap¡Integer,DocumentIndexEntry¿. In the DocumentIndexEntry we have 2 fields:

- private int docNo;
- private int docLength;

Chapter 2

Index Construction

In this chapter we're going to analyze the construction of the Index.

2.1 Tokenizer

The tokenizer is the first building block during the indexing phase. Our tokenizer implementation implements these functions:

- HTML tags removal;
- Punctuation removal;
- Stopwords removal (using a list of english stopwords);
- Stemming (reducing words to their root or base form, removing suffixes and prefixes) using PorterStemmer.

This pre-processing of documents may produce empty documents "". These documents are discarded, and this is the reason why a docNo is needed, to mantain the mapping between the document and the docId.

2.2 Collection reading, Dumper & Fetcher

2.2.1 Compressed Reading

The document collection is uncompressed during parsing using the the GZIPInputStream class, in order to avoid decompressing all the document collection, saving disk space and time.

2.2.2 Dumper & Fetcher

We've created 2 interfaces, Dumper and Fetcher, whose responsibilities are:

- Dumping and Reading the entire vocabulary
- Dumping and reading a specific vocabulary entry
- Dumping and reading the document index

```
+-------------------------------------------------------+
|   term    |   DF  |   UB   | docO | docL | termO | termL |
+-------------------------------------------------------+
Ordered by term:
DF → Document Frequency, Length of the posting list,
UB → Upper Bound, a double computed during index creation for scoring purposes
docO → DocumentIdOffset, the offset (in byte) at which the list is stored
docL → The length of the list in byte
termO → TermFreqOffset, the offset (in byte) at which the list is stored
termL → The length of the list in byte
```

Figure 2.1: Lexicon Entry Structure

We have 2 implementations Dumper and Fetcher, that are DumperTXT/FetcherTXT and DumperBinary/FetcherBinary. The first is used for debugging purposes, while the second one is used to dump intermediate blocks in the SPIMI indexing phase. DumperCompressed and FetcherCompressed extend DumperBinary and FetcherBinary respectively, and they're used to dump and read the posting list in a compressed way.

2.3 SPIMI

We implemented a SPIMI (Single Pass In Memory Indexing) algorithm composed of basically two phases:

- In the first phase, a series of blocks are written. In each block we write all data structures that would compose an index of documents belonging to that block: lexicon, document index and inverted index
- In the second phase, we merge these structures in order to create the final ones

Files Structure

From now on, when talking about dumping structures, we will always refer to generating the following files:

- **Vocabulary.bin** : the lexicon. Here we store terms and all informations required for pre-scoring (upper bound) and posting list retrieval (offsets and length)
- **DocumentIndex.bin** : a sequence of triplets (DocId , DocNo, Length). The very first two entries are N and L, the total number and length of documents inside the collection.
- **DocIds.bin** : the sequence of docids of all posting lists
- **TermFrequencies.bin** : the sequence of term frequencies of all posting lists

Since the lists of posting are written without any *"mapping"*, we store inside the lexicon both the offset and the length in byte of the list(2.1), in order to easily navigate the other two files.

Phase 1: Blocks Writing

In order to write blocks, we follow the implementation of `InMemoryIndexing.java`:

```
...
while(<memory_allowed>)
    <processNextDocument>

dumpVocabulary()
dumpDocumentIndex()
...

DEF processNextDocument(doc)
    terms = doc.Tokenize()
    stems = terms.Stemmer()
    <updateVocabulary>
    <updateDocumentIndex>
```

The class has a Tokenizer interface (see above), and the vocabulary entries are updated adding eventually a Posting to the posting list or creating a new entry in case of new term. Regarding document index, a new couple `DocId | DocLength` is created.

The indexer has a Dumping interface that is adaptive with the mode we are running the indexing session with. When we request for an explicit TXT version of the index, the dumper is the `DumperTXT.java`. **Otherwise, It is always** `DumperBinary.java`.

Infact, regardless compression is required or not, posting lists are not compressed inside each block, but only in the second phase of the indexing, when multiple pieces of the same posting list are merged together.

The creation of a new block is triggered when the used heap memory is greater than the 80% of the total memory allowed. *Note: since in general SPIMI works under memory limit assumption, we force a garbage collection (* `System.gc()` *) between each block.*

Phase 2: Merging

Regarding merging, we have decided to open K pointers to all blocks together for reading, and 1 writer for each final file to be written.

In practice, the merging phase consists of:

```
...
concatenateDocIndexes()
mergeBlocks()
...

DEF concatenateDocIndexes()
    <init N, L>
    FROM ALL BLOCKS
        <read N_B, L_B>
        N += N_B
        L += L_B

    concatenateFiles()
```

```
DEF mergeBlocks()
    WHILE(!end())
        term = lowestTerm()
        list = mergePostings(term)
        <updatePointers>
        dumpStructures()
```

Since blocks are created by parsing the original collection, It appears clear that document indexes have to be *glued together*. The exception is in the first two entry of the file, reserved for the Number of documents and the total Length of documents. We want to do that *before* merging vocabularies in order to have the total number of documents available for pre-scoring purposes (upper bound computations).

Since blocks'vocabularies are sorted by term, It is sufficient to check the lexicogrphically lowest term pointed by pointers to decide which term to merge. For all blocks pointed during the iteration, we load relative postings and we merge just by concatenating them (since postings in the block 2 are necessarily greater-in-docId than block 1's, when term is equal). When compression is enabled, we dump the compressed view of the merged list.

After fully dumping a term, we obviously have to advance only pointers relative to blocks that have been considered during this iteration, in order to don't miss some lists.

Note: we dump term by term, even if we suggest for future implementations to buffer entries when memory available.

2.4 Compression

Compression is crucial during indexing in information retrieval as it significantly enhances efficiency and resource utilization. By reducing the size of indexed data, compression minimizes storage requirements, accelerates data retrieval times, and optimizes overall system performance. In our trials, we implemented different algorithms (all implementing the *Encoder* interface) in order to find the best ones for our specific case.

2.4.1 Doc ids

Elias-Fano

The Elias Fano compression algorithm is a technique designed for compressing ordered lists of integers efficiently. In our implementations, in order to use skipping pointers, we defined this architecture for compressing with Elias-Fano:

```
| U | n |     BLOCK      | U | n |     BLOCK    ...
```

By knowing U and n, we know how many bytes the block will be composed of. This way, the nextGEQ(int docid) implementation will simply look at U, if it is higher than the docId than the current block will be the one to be decoded, otherwise we'll know how many bytes to jump in order to find the next U and n, relative to the next block. This means that we won't need to decode all the blocks in between, but only the ones in which we're interested.

```
int lowHalfLength = (int) Math.ceil(Math.log((float) U / n) / Math.log(2));
int highHalfLength = (int) Math.ceil(Math.log(U) / Math.log(2)) - lowHalfLength;
```

```
int nTotHighBits = (int) (n + Math.pow(2, highHalfLength));
int nTotLowBits = lowHalfLength*n;
int nTotHighBytes =(int) Math.ceil((float)nTotHighBits/8);
int nTotLowBytes = (int) Math.ceil((float) nTotLowBits/8);
int totBytes = nTotHighBytes + nTotLowBytes
```

Simple-9

The Simple9 compression algorithm is a technique specifically designed for compressing lists of integers with a fixed bit-width. To reduce the dimensions of the blocks, with Simple9 we do not encode the doc ids directly but we encode the DGaps (difference of two consecutive doc ids). In particular, the first doc id will be encoded as is and from the second onwards we encode the difference with the previous doc id.

Similar to Elias Fano, we add before each block a skipping pointer that consists of the upper bound and the length of the block (both informations used by nextGEQ to speed up posting retrieval)

```
| length | U |     BLOCK      | length | U |     BLOCK    ...
```

PForDelta

The PForDelta algorithm is an algorithm that encodes a large portion of numbers (in our implementation 90%) with a fixed number of bits k: we discards the highest numbers that would require an higher number of bits and encode them separately as outliers. We store all the important information needed to decode the block (and also the upper bound used by nextGEQ) before each block

```
| U | b | k | size | outliers size |    BLOCK    |  outliers BLOCK  |
```

where b is the base, i.e. the smallest number of the list (when we encode a number x we encode the difference $x - b$ as this requires less bits), `size` is the number of elements in the block and `outlier size` is the length of the outliers block.

As for Simple9, we do not encode the doc ids directly and instead encode the DGaps.

2.4.2 Term Frequencies

Simple9

We used Simple9 also for term frequencies compression: the algorithm is the same of the one used for doc ids compression, the only difference is that we do not store the upper bound of the block in the skipping pointer since this information is useless for term frequencies

```
| length |     BLOCK      | length |     BLOCK    ...
```

PForDelta

We used PForDelta also for term frequencies compression since we expect a lot of postings having low term frequencies and just a few ones having higher term frequencies (that will be encoded as outliers). The structure of our block is the same as the one used for doc ids with the exception of the upper bound

```
| b | k | size | outliers size |    BLOCK    |  outliers BLOCK  |
```

Unary

The Unary compression algorithm is a simple algorithm that encodes a number n using just n bits. We expect this algorithm to perform well with term frequencies since a lot of postings have a low term frequency (most words of a document appear just once or a few times at most).

Also in this case we added before each block a skipping pointer with the length of that specific block

```
| length |     BLOCK      | length |     BLOCK    ...
```

2.4.3 Skipping

The skipping mechanism is used by the `nextGEQ(int docid)` function during the querying phase. In fact we do not encode our doc ids and term frequencies lists as whole, but we split them in blocks of fixed size (`BLOCK_SIZE`) and encode each block separately.

We implemented skipping by writing on file (`doc_ids.bin`) before each block all the information needed to skip the block without decoding it (for each algorithm the skipping pointer is implemented in a different way). By reading the skipping pointer, we're able to understand if the block contains the doc id we're searching for (without decompressing all the list), and if not, we're able to compute the number of bytes we need to jump in order to land to the next block. When we jump to the next doc ids block we concurrently jump to the next term frequencies block (they always contains the same number of elements).

This mechanism enables to avoid decompressing the entire posting list when searching for a particular posting.

2.5 Collection Data Analysis

We analyse here the size of the vocabulary and of the relative indexed posting lists (even over the relative Zyps' Law).

Note: all studies are done with Porter Stemmer applied and a stopwords file[1] used.

First of all, some collection facts:

- There are 8.8M of documents inside the collection
- The *average length* of documents is really low, with 30 words per passage
- We have collected around 1.76M of terms

We found out that Zipf' predictions are respected, as you can see in (2.2).

/AppearsInAtLeast	1	5	100	10K	100K
# Terms	1743965	324854	45907	3252	348
Avg. # Postings	114	606	4183	46613	183779

Table 2.1: Posting Lists sizes and Cardinality against Term Popularity

[1] https://raw.githubusercontent.com/stopwords-iso/stopwords-en/master/stopwords-en.txt

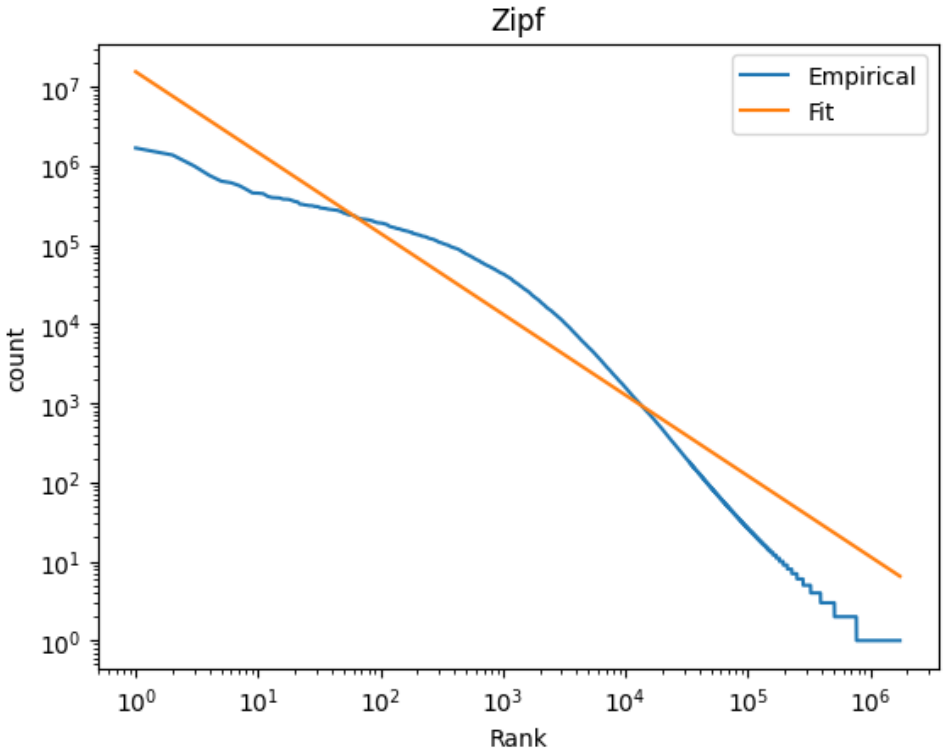

Figure 2.2: Zipf Fitting against real Postings size

Moreover, we made some experiments on the size of posting lists. You can check for results in table (2.1),

Since the posting lists sizes are quite agreeing with the Zipf's law, we have decided to vary the size of compression blocks in a common-sense-range [10K, 100K]. We believe that, having blocks greater than 100K is in practice useless, since our studies showed that only 348 terms appear in more than 100K document.

Regarding document length, in order to understand whether BM25 and BM11 could have some advantages with respect to BM15, we found an average length of 30 tokens with a spread of 13.

Chapter 3

Query processing

In this chapter we're going to analyze the Query processing task.

3.1 Scoring functions

We've decided to implement 2 of the several scoring function that we've studied: TF-IDF and BM-25.

3.1.1 TF-IDF

TF-IDF is a scoring function calculated by multiplying two components: Term Frequency (TF), which measures how often a term appears in a document, and Inverse Document Frequency (IDF), which assesses the rarity of the term across the entire document collection. The TF part has to be computed at runtime, while the IDF part could be computed offline, in order to avoid the computational overhead, but increasing the amount of memory used to store this value. For this reason, our choice was to also compute IDF at runtime (since it only requires to compute a logarithm and a division).

3.1.2 BM25, BM11, BM15

We've considered also the BM25 score function together with its variants: BM15 (no document length normalization) and BM11 (document length completely normalized).

3.2 Max Score

Max Score is a dynamic pruning algorithm used to fasten the queries execution. It aims to boost DAAT algorithm by skipping the evaluation of some documents that we already know won't be able to reach the threshold to enter in the top k documents. It is used for the disjunctive mode.

3.2.1 Conjunctive mode

For the conjunctive mode we've implemented a function named nextConjunctive, which is in charge to leave all the posting lists to the same docId, otherwise returns -1 (indicating that there isn't any other document which is in common to all the posting lists). Here we have the pseudo-code:

```
current = getMinimumDocId(postingLists);
while(true)
    allToCurrent = true

    for each p in postingLists:
        if ( p.docId() < current ) p.nextGEQ(current)
        if ( p.isEmpty() ) return -1
        if ( p.docId() > current )
            current = p.docId()
        allToCurrent = false
        break

    if ( allTocurrent == true ) return 0
```

By looping over this function, we're able to jump to documents that are in common to all the posting lists efficiently, since we're making use of the nextGEQ() function.

Chapter 4

Evaluation

4.1 TREC_eval

Regarding Evaluation, we have used the `trec_eval` tool (TREC_2019) to extract MAP, MRR and NDCG[1] from our scoring functions. We tested TFIDF, BM11, BM15, BM25 with a variety of parameters. All experimente, whose summary can be found at tables (4.1) (4.2), have been done on a index with stopwords removal and stemming.

/Parameters	TFIDF	BM11 (K = 0.9)	BM15
MAP	0.1049	0.0938	0.11
NDCG	0.2151	0.2036	0.2208
MRR	0.7750	0.7766	0.8047

Table 4.1: Trec Eval Results (1)

/Parameters	BM25 (K=0.9, B=0.8)	BM25 (K=1.5, B=0.5)	BM25 (K=1.2, B=0.6)
MAP	0.1002	0.0991	0.1020
NDCG	0.1997	0.2028	0.2063
MRR	0.7777	0.8118	0.8037

Table 4.2: Trec Eval Results(2)

4.2 Time statistics

Here we're going to show some timing statistics

[1] With default parameter b = 2

4.2.1 Indexing time

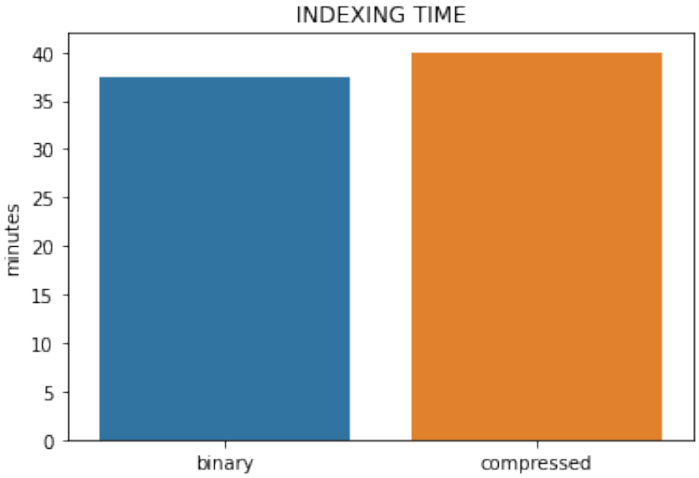

Figure 4.1: Indexing time

The indexing time for the binary version (so without any type of compression) is around 37 minutes and 32 seconds, while for the compressed version (S9+Unary, blockSize=1000) it's around 40 minutes and 11 seconds. This means that compressing the posting lists during the merging & dumping phase takes around 2 minutes and half, which is roughly 6% of the indexing time.

4.3 Index size

The uncompressed size of doc ids and term frequencies is around 721 MB.

Figure 4.2 and 4.3 shows the performances of the algorithms described in section 2.4. We notice that the higher the block size, the better the compression:

- the higher the block size, the lower the number of skipping pointers that have to be inserted in the list (and so less space is "wasted" on skipping pointers);
- the higher the block size, the longer is the list to be compressed, and this may lead to a better compression-rate of the algorithms.

For term frequencies the best compression algorithm is Unary, for doc ids it is Simple9 for smaller block sizes and Elias Fano for very large block size. We are careful in choosing a block size that is too big since this may result in a degradation of querying performances (see section 4.3.1)

Our preferred combination of compressions and block size will be Simple9 for doc ids, Unary for term frequencies with a block size of 1000: we got 287MB for doc ids (compression rate of about 60%) and 38MB for term frequencies (compression rate of about 95%).

4.3.1 Query time

We know that different compression algorithms and different block sizes can lead to a variation both in the space occupied by the posting lists, but it will also lead to a variation in time to execute a query. Figure 4.4 shows how the time of executing a query is affected by these parameters.

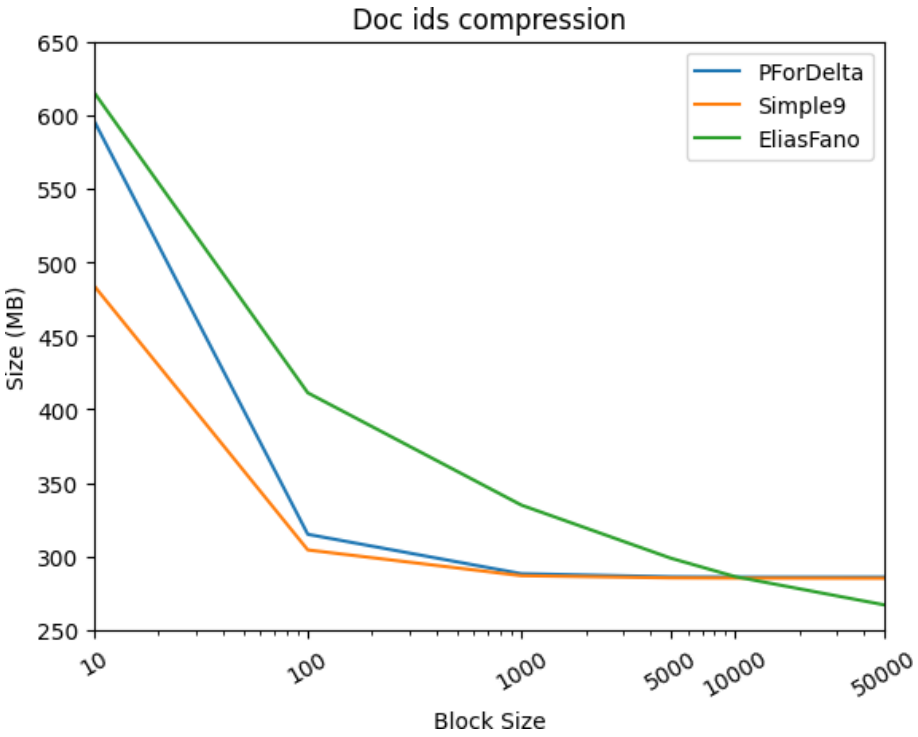

Figure 4.2: Analysis of the performances of different algorithms in compressing doc ids.

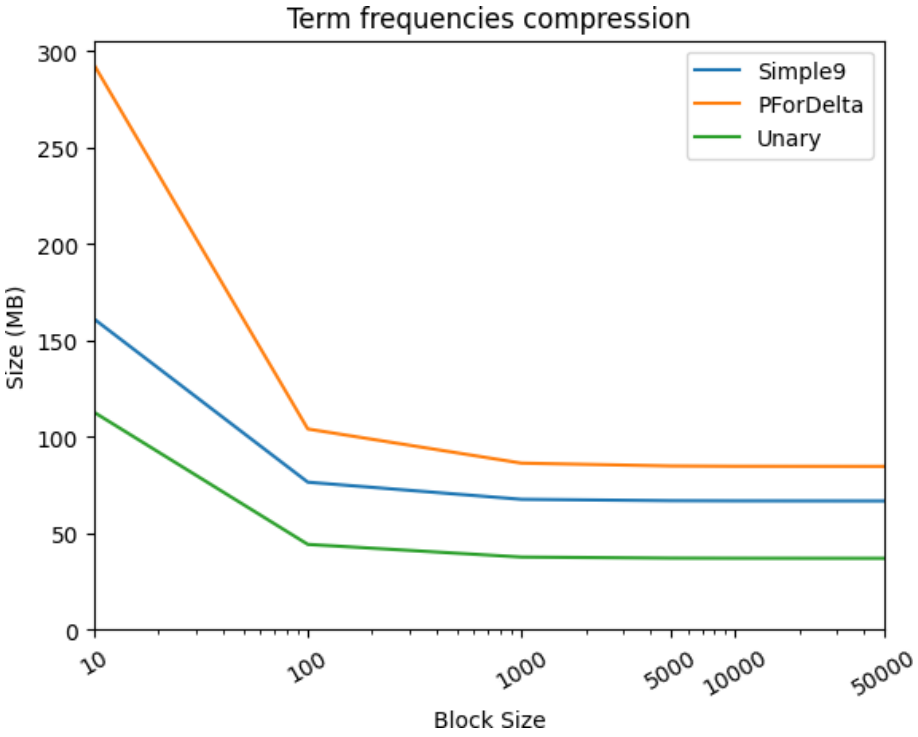

Figure 4.3: Analysis of the performances of different algorithms in compressing term frequencies.

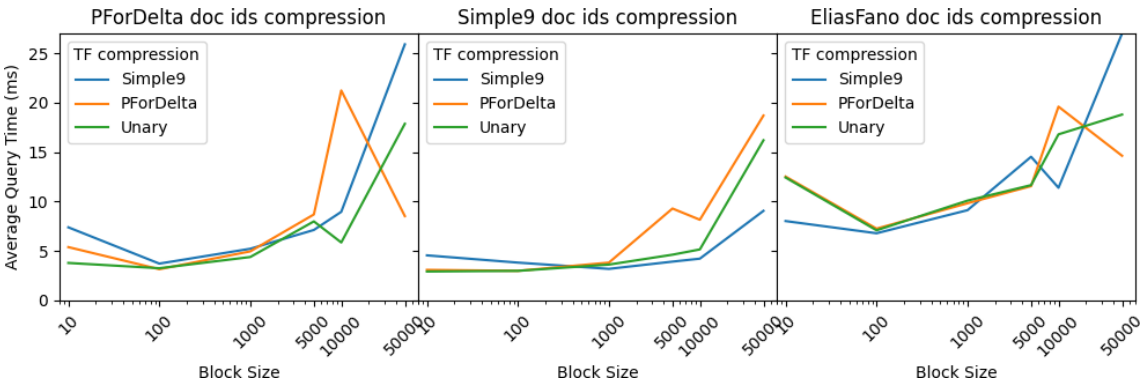

Figure 4.4: Average time for a query (in ms) with respect to various combination of compression algorithms and block sizes

We see that for very small block size (block size = 10), the time of executing a query is higher than a little bigger block size (block size = 100): this is expected since the `nextGEQ` function will need to perform a lot of jumps in order to reach a distant document, leading to an increase of time used. On the other hand, for bigger block sizes, we will perform fewer jumps but we will need to decompress bigger and bigger blocks and so this will inevitably lead to an increase of time. As always, the best values lie in the middle between the 2 extremes: it turns out that block sizes between 100 and 1000 performs very well, as they guarantee lower query times.

As mentioned before, we will adopt the Simple9 algorithm for doc ids compression and Unary algorithm for term frequencies compression with a block size of 1000: this choice gives us a good trade-off between query times and index compression.

4.4 Caching Strategy

When using the system, the caching strategy adopted has significant impact on the query throughput.

```
lake 5
heat 6
cost 6
sugar 6
locat 7
blood 8
counti 22
mean 22
definit 38
```

Figure 4.5: Example of Influencers.txt file with structure `term | #searches`

Influencer Terms

With *influencers*, we refer to the top-1000 most frequent terms refered into queries. To mantain this structure updated across sessions, we mantain a counter in each lexicon entry, and we update it every time we access it.

When the session is closed, the counter for that term is dumped on a file(4.5).

At the beginning of the session, **until we use more than 50% of the total memory allowed**, we put in RAM those entries with relative postings.

We thought about this strategy in order to make our system more responsive expecially at the beginning of the use case.

Throughput Comparison

To better show that, we have considered two case:

- *CASE 1:* No caching adopted. After each query all results computed are flushed, and nothing remains in RAM, neither lexicon entry neither relative posting lists
- *CASE 2:* Java "natural" caching adopted, in the sense that, if there is enough space in memory, results are mantained in RAM. Moreover, influencers posting list are preliminary loaded in RAM, if possible.

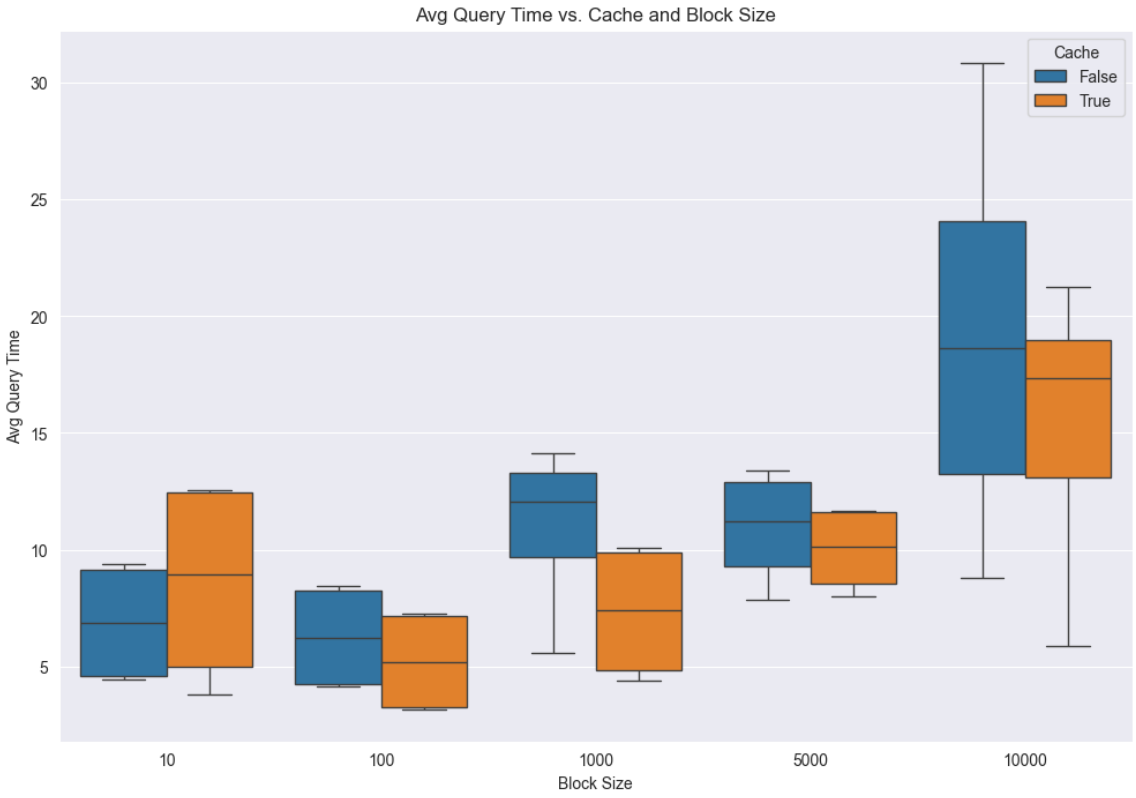

Figure 4.6: Caching Effects on all compressions varying on blocks dimension

Results are summarized in figure (4.6), where results are aggregated for the first three queries of the session (since we think that at the start the advantage is more evident). When the session lasts for a large number of queries, the time seems to align with other cases.

Moreover, It is interesting to notice that the benifts of caching seems to be not correlated with the choice of compression strategy or blocks dimension.

Chapter 5

Conclusion

Results we obtained are summarized here:

- Index Construction is possibile even under memory constraints thanks to the Single Pass In Memory Indexing strategy
- Index Size highly varies depending on the format we decide to use to dump data structures and if we use to compress or not (we in general have a quite high compression rate, higher than the 40% with respect to the ASCII version).
- Choosing the right compression is a trade off of indexing size and query throughput. With our result, we would say that the Unary compression is quite the best option for Term Frequencies compression, because of the really low frequency of a word in a passage of few words as this collection. Regarding document ids compression, we choose Simple9 with gap strategy.
- Regarding block size, we opted for a trade off between average query throughput and rate of compression, choosing a rate of 1000
- Caching the top searched terms has a significant impact on query thrughput on the very first few queries of the session, expecially when using TFIDF
- BM15 seems to have better results in all metrics involved, suggesting that the document length of documents in this collection is not so relevant (not so strange since these are little passages with a low spread in number of words).

Future Implementations

- Loading the Document Index in memory before starting asking for queries, in order to prevent BM25 latency in the online phase
- Buffer the merging phase during index construction, in order to have less indexing time