UNIVERSITÀ DI PISA

Master's degree in Artificial Intelligence and Data Engineering

Multimedia information retrieval and computer vision course

SEARCH ENGINE

Domenico D'Orsi

Denny Meini

Matteo Razzai

Academic Year 2023/2024

Github repository: https://github.com/matteorazzai1/SearchEngine

1. Introduction

1.1 Project description

The goal of this project was to build a simple search engine based on an inverted index and capable of execute some queries on it, retrieving the best documents for each query. Starting from the MSMARCO passages collection (which is available at the following link: https://msmarco.z22.web.core.windows.net/msmarcoranking/collection.tar.gz) we developed our project executing three main steps:

1. The indexing of the documents
2. The processing of the queries
3. The evaluation of the performances

During the indexing we built the inverted index, the document index and the lexicon, that contain the information we needed in order to execute the query. We implemented two algorithms for quey processing, which are DAAT (Document-at-a-Time) and MaxScore, and we gave to the user the possibility to choose between conjunctive and disjunctive queries. Finally, we evaluate our project by performing Trec_eval, using the MSMARCO queries and qrels (2020).

1.2 Modules organization

We developed our project using Java and we organized it in 4 modules:

- Common
- Indexer
- QueryHandler
- Tester

The module **common** contains the Java classes that implement the base structures of the project (such as Lexicon, Posting List, …), the classes that deal with compression and decompression of the data, a file Constants with the paths used in the project and the values of the constants for the formulas we used (like k1 for BM25), a file FileUtils which implements some methods that deal with creation of buffers, clearing of files and folders ad retrieving of the files, and a file Preprocesser which contains methods that allows us to preprocess the collection and the queries (stemming, stop words removal, …).

The module **indexer** contains the SPIMI class, which perform the SPIMI algorithm to create the intermediate inverted indexes and the document index, and the Merger class, which merges the intermediate indexes in order to create the final inverted index and the lexicon, both of the classes save the outputs on the disk. There's an IndexerMain class which starts the execution of SPIMI and Merger.

The module **queryHandler** contains the class DAAT, which implements the homonym algorithm for disjunctive and conjunctive executions, the class MaxScore, which implements the execution of the MaxScore pruning optimization, the class Utils which implements some utility methods like

nextGEQ and the scoring functions (TFIDF and BM25) and the class LRUCache which implements some methods for caching.

The module **tester** contains a class Tester which implements the CLI for insert the queries and the setup for trec_eval.

All the modules contain the relative pom and, except tester, some testUnits which test the functionalities of the methods using **Junit**.

2. Indexing

2.1 Preprocessing

The collection from which we built our index is the **MSMARCO passages collection**, which is made of 8,841,823 passages and has a dimension of 2.9 GB. The format of a line of the collection is the following:

<pid>\t<text>\n

Where the pid is the docNo and the text is the text of the passage. As an example, here we have the first passage of the collection:

0 The presence of communication amid scientific minds was equally important to the success of the Manhattan Project as scientific intellect was. The only cloud hanging over the impressive achievement of the atomic researchers and engineers is what their success truly meant; hundreds of thousands of innocent lives obliterated.

The preprocessing of documents and queries is managed by the class **Preprocesser.java**, contained into the module common. The first thing we did was to put into two lists all the stopwords and the punctuation signs that will be removed from the passages during the preprocessing (image below).

```
//a list of stopwords that will be removed from the text
private static List<String> stopwords = Arrays.asList(
        "a", "about", "above", "after", "again", "against", "all", "am", "an", "and", "any", "are",
        "aren't", "as", "at", "be", "because", "been", "before", "being", "below", "between", "both",
        "but", "by", "can't", "cannot", "could", "couldn't", "did", "didn't", "do", "does", "doesn't",
        "doing", "don't", "down", "during", "each", "few", "for", "from", "further", "had", "hadn't",
        "has", "hasn't", "have", "haven't", "having", "he", "he'd", "he'll", "he's", "her", "here",
        "here's", "hers", "herself", "him", "himself", "his", "how", "how's", "i", "i'd", "i'll", "i'm",
        "i've", "if", "in", "into", "is", "isn't", "it", "it's", "its", "itself", "let's", "me", "might", "more",
        "most", "mustn't", "my", "myself", "no", "nor", "not", "of", "off", "on", "once", "only", "or",
        "other", "ought", "our", "ours", "ourselves", "out", "over", "own", "same", "shan't", "she",
        "she'd", "she'll", "she's", "should", "shouldn't", "so", "some", "such", "than", "that", "that's",
        "the", "their", "theirs", "them", "themselves", "then", "there", "there's", "these", "they",
        "they'd", "they'll", "they're", "they've", "this", "those", "through", "to", "too", "under",
        "until", "up", "very", "was", "wasn't", "we", "we'd", "we'll", "we're", "we've", "were", "weren't",
        "what", "what's", "when", "when's", "where", "where's", "which", "while", "who", "who's", "whom",
        "why", "why's", "with", "won't", "would", "wouldn't", "you", "you'd", "you'll", "you're", "you've",
        "your", "yours", "yourself", "yourselves");

// list of punctuation
private static List<String> punctuation = Arrays.asList(
        "!", "?", "(", ")", "[", "]", "{", "}", ",", ";", ".", ":", "'", "“", "”", "’",
        "+", "-", "*", "/", "|", "\"", "\\", "_", "#", "<", ">", "%", "=", "^", "$", "&");
```

The class contains 4 methods:

- **List<String> parse (String text)**: it receives as an input the string *text*, remove from it all the HTML tags, the URLs and the punctuation and then it parses the string (removing also the stopwords and all the words which contains at least one not-UTF8 character) and transforms every word to lowercase, obtaining the list of tokens of the row.
- **List<String> stem (List<String> terms)**: it receive a list of tokens and stem them using the ca.rmen PorterStemmer algorithm.
- **String processCLIQuery(String row)**: it receives as an input a query from the CLI, makes use of the methods *parse* and *stem* (in this order) and finally merge all the tokens into a space-separated string.
- **String process (String row)**: same as *processCLIQuery* but, because it receive a row of the collection or a MSMARCO query, it has to deal with the pid, so it put the pid as the first term of the output string, followed by a \t.

2.2 Data Structures

In order to keep the information about the different index and entries, we defined some data structures which are used in the methods of all the project. They are:

- Document Index
- Inverted Index
- Lexicon
- Lexicon Entry
- Posting
- Posting List
- Skipping Block

Here after we have a brief description of the structures (each class has get and set methods that will not be descripted):

2.2.1 Document Index

The class **DocumentIndex** keeps the information on the document index, the fields are the following:

Type	Name	Description
DocumentIndex	instance	The unique instance of the class
int[]	docsNo	Each element of the array indicates the name (the pid) of a document
int[]	docsLen	Each element of the array indicates the length of a document
double	AVDL	The average document length of the collection
int	collectionSize	The size of the collection

The class is a singleton class, because we will never have more than one instance of the document index. Some interesting methods are the **readFromFile()**, which opens a FileChannel with the final document index in order to retrieve it from the file, the **saveCollectionStats(double AVDL, int collectionSize)** and **loadCollectionStats()** which, respectively, saves in a Preferences object and load from it the values of AVDL and collectionSize, and **retrieveDocsNo()**, which opens a FileChannel with the final docNo index in order to get the docsNo.

2.2.2 Inverted Index

The class **InvertedIndex** keeps the inverted index, the fields are the following:

Type	Name	Description
InvertedIndex	instance	The unique instance of the class
HashMap<String, PostingList>	PostingLists	An hashmap mapping every posting list with its term

This class is a singleton class too because we will never have more than one instance of the inverted index.

2.2.3 Lexicon

The class **Lexicon** keeps the lexicon, there's only one field:

Type	Name	Description
HashMap<String, LexiconEntry>	Lexicon	An hashmap mapping every lexicon entry with its term

An important method to highlight is **retrieveEntryFromDisk(String term)**, which opens a FileChannel with the lexicon file on the disk and retrieve the lexicon entry related to the term *term* with an algorithm of **binary search**.

2.2.4 Lexicon Entry

The class **LexiconEntry** keeps the information for a single entry of the lexicon, these are the fields:

Type	Name	Description
String	term	The term of the lexicon entry
int	df	Document frequency of the term (default 0)
double	idf	Inverse document frequency of the term (default 0)

int	termCollFreq	The frequency of the term inside the collection (default 0)
int	maxTf	The max term frequency of the term inside the collection (default 0)
double	maxTfidf	The value of Tfidf related to the maxTf (default 0)
double	maxBM25	The maximum value of BM25 of the term (default 0)
long	offsetIndexDocId	The entry offset in the docId file of the inverted index (default 0)
long	offsetIndexFreq	The entry offset in the frequency file of the inverted index (default 0)
int	docIdSize	The size of the posting list of the term in the docId file of the inverted index (default 0)
int	freqSize	The size of the posting list of the term in the freq file of the inverted index (default 0)
long	descriptorOffset	The starting position of the blockDescriptor into the file (default 0)
long	numBlocks	The number of blocks (≥1) to split the list into (default 1)

This class has a method **writeLexiconEntry(long positionTerm, FileChannel channelLex)** which write the lexicon entry into the lexicon file using a FileChannel and a MappedByteBuffer starting from the offset indicated by *positionTerm*. There is a method **readLexiconEntry(long positionTerm, FileChannel channelLex)** too, which perform the inverse operation. We also have a method to compute the max value of BM25 of a posting list (**computeMaxBM25(PostingList postingList)**).

2.2.5 Posting

The class **Posting** contains the information of a single posting, with the following fields:

Type	Name	Description
int	docId	The docId the posting refers to
int	frequency	The frequency of the term into the document with the docId = docId

2.2.6 Posting List

The class **PostingList** contanins the information of the posting list related to one term, these are the fields:

Type	Name	Description
String	term	The term the posting list refers to

ArrayList<Posting>	postings	A list of all the posting related to the term

There are 5 constructors for this class:

- **PostingList()**: empty constructor.
- **PostingList(Posting p)**: creates a posting list starting from a posting *p*.
- **PostingList(String line)**: receives a line from the intermediate index file containing the term at the start followed by a tab and a list of posting. It transforms them into a PostingList object.
- **PostingList(String term, ArrayList<Posting> postings)**: creates a posting list startng from the term and the ArrayList of postings.
- **PostingList(PostingList p)**: copy constructor.

We also have a method **appendList(PostingList intermediatePostingList)** which is used to merge the various posting lists of the intermediate index, and an **updatePosting(int docID)** method, that update the frequency of the posting with *docID* if it exists, or creates a new posting otherwise.

2.2.7 Skipping Block

The class **SkippingBlock** keeps the information of a skipping block, these are the fields:

Type	Name	Description
int	maxDocId	The Max docId of the block
long	offsetDocId	The offset of the block inside the docId inverted index file
int	docIdSize	The size of the block inside the docId inverted index file
long	offsetFreq	The offset of the block inside the freq inverted index file
int	freqSize	The size of the block inside the freq inverted index file
int	numPosting	The number of postings inside the block

This class has 2 constructors, one which receives the fields in input and one which is empty. An interesting method is **writeSkippingBlock(long positionBlock, FileChannel blocks)** which writes a skipping block on the block file in memory at the offset *positionBlock* using a FileChannel, then we have **readSkippingBlocks(long positionBlock, FileChannel blocks)** that does the opposite operation, reading the block. Finally we have a method **retrieveBlock()** which retrieve the posting list relative to the block directly from the inverted index saved on disk.

2.3 SPIMI

The first part of the indexing process is the Single Pass In Memory Indexing (*SPIMI*) algorithm, a state of the art method for this kind of purpose. Our version, implemented in the class *src/main/java/it/unipi/mircv/SPIMI.java* basically follows the most known and common version of the algorithm:

```
SPIMI-INVERT(token_stream)
 1  output_file = NEWFILE()
 2  dictionary = NEWHASH()
 3  while (free memory available)
 4  do token ← next(token_stream)
 5     if term(token) ∉ dictionary
 6        then postings_list = ADDTODICTIONARY(dictionary, term(token))
 7        else postings_list = GETPOSTINGSLIST(dictionary, term(token))
 8     if full(postings_list)
 9        then postings_list = DOUBLEPOSTINGSLIST(dictionary, term(token))
10     ADDTOPOSTINGSLIST(postings_list, docID(token))
11  sorted_terms ← SORTTERMS(dictionary)
12  WRITEBLOCKTODISK(sorted_terms, dictionary, output_file)
13  return output_file
```

but with respect to this pseudocode, there are some minor differences.

The first difference is that our code implements periodic flush on the disk of two heavy structures, which cannot fit entirely in memory during this process: **inverted index** and **document index.** Both, in fact, will be moved to disk by specific methods in case the main memory is 80% full.

For what regards the rest of the operations performed by the algorithm, it simply processes one line at a time from the collection, assigning a DocID to it and mapping this value to the corresponding DocNo. After the preprocessing phase, each token belonging to the document is treated by updating its own posting list increasing the frequency for the current document by 1, or by creating a new posting list if it doesn't exist.

At last, to create the *Average Document Length (AVDL)* statistic, a specific accumulator sums up the number of tokens encountered by far.

The output of this method will be the *Collection size* and *AVDL* statistics, along with a set of intermediate indexes to be merged.

2.4 Merging

The second part of the Indexing consists of the Merger algorithm. Our version of the algorithm, implemented in *indexer/src/main/java/it/unipi/mircv/Merger.java* This algorithm takes the intermediate inverted indexes produced by Spimi and merge all the posting lists related to a single term. It makes that one term at a time in lexicographically order.

After the merging phase of the posting lists of a single term, the method *saveMergedIndex* takes as input the merged posting list and produced the relative entry for the Lexicon (LexiconEntry data structure) and the entries for implementing the skipping block (SkippingBlock data structure).

Regarding the creation of the blocks, each block contains $\lceil \sqrt{(number\ of\ postings)} \rceil$ postings and we decided to split in blocks only the posting lists with length greater than 512 postings, to not waste further resources.

Eventually the merged posting list, its related lexicon entry and all the information related to the blocks in which the posting list has been split are saved on disk, using compression of docIds and frequencies for what concern the posting of the merged posting list.

2.5 Compression

We decided to use compression algorithms to reduce inverted index's memory occupancy. We decided to use two types of compression algorithms:

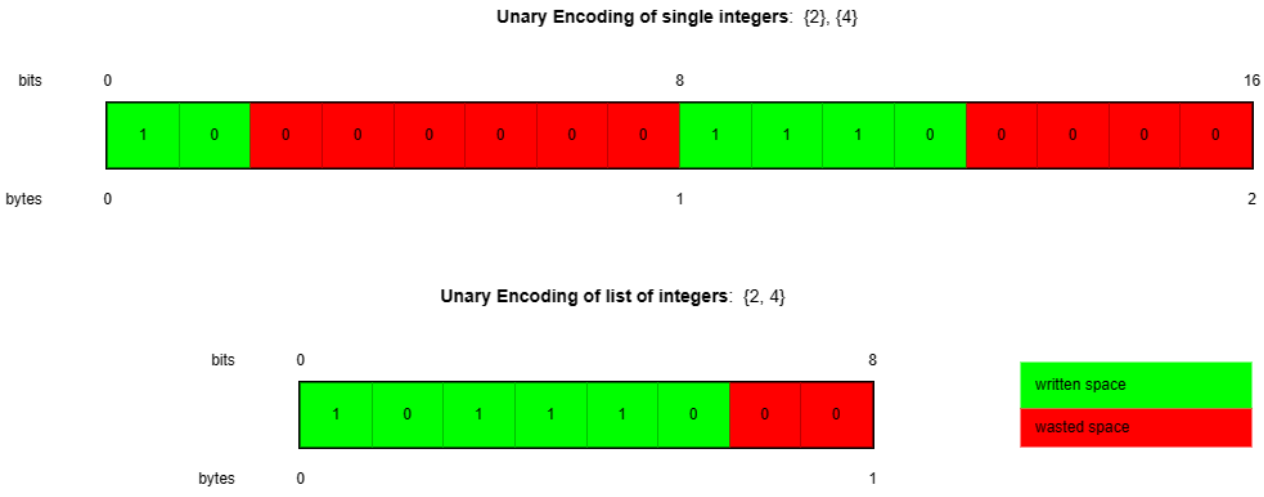
- Unary encoding algorithm, to compress frequencies.
- Variable-byte encoding algorithm, to compress docIds.

Eventually we created two different compressed files, one for docIds *indexer/data/inv_index_docId.dat* and one for frequencies *indexer/data/inv_index_freq.dat*. This splitting of the inverted Index allows us to use different compression and facilitate the decompression of the two components of the posting list.

Our implementation of the compression algorithms can be found in the following folder *common/src/main/java/it.unipi.mircv/compression*.

2.5.1 Unary Encoding

Unary Encoding is used to compress the frequencies of the posting. Seen that we choose Java as programming language for this project, we must deal with the fact that this programming language write and read no less than a single byte, so the benefits of this compression encoding come from the compression of a list of integers and not from the compression of single integers, as it is possible to see in the following image.



As we can see from the above image, the wasted space decreases from 10 bits to 2 bits.

2.5.2 Variable-Byte Encoding

Variable Byte compression is used to compress the DocIds of the postings. We decided to use this type of compression because it is functional for high values, so, seen that the DocId goes from 0 to the total number of documents present in the collection, the choice of this encoding is seemed to us to most suitable to handle this situation.

3. Query processing

The main functionality of our application is of course the one related to queries. As it will be explained later, the user can interact with the system to submit queries and get as a result the most relevant documents and the relative score obtained.

As first step, we will always load in memory the collection statistics which are necessary for computing the score functions. Then, the interaction with the system starts and the user can select the number of results to retrieve, the algorithm and the scoring function to be used. To fit the requirements of the project, we developed two macro-classes of queries: *conjunctive queries and disjunctive queries.*

For what regards the first, we designer a conjunctive version of the ***DAAT (Document-at-a-time)*** algorithm, a querying algorithm designed to process the posting lists in parallel, focusing on one document at a time. We have developed a version of DAAT for disjunctive queries as well, however, for this type of query, we have also defined an optimized algorithm that leverages dynamic pruning to expedite the process: ***MaxScore.***

The two possible options for the scoring function are two of the most used and well known in information retrieval field: ***TF-IDF*** which is based only on term frequency and inverse document frequency of the term, and ***BM25*** which instead takes in account also the document length, normalizing it w.r.t. the average document length of the collection, and a couple of smoothing constants.

3.1 Skipping

Before introducing our query processing, it is necessary to define the way we implemented skipping inside of the project. This allowed us to avoid the storage of the entire posting lists to traverse during query processing in main memory, keeping only a small part of each of them in memory represented by a single *block*, accessible through a specific descriptor. This not only saves main memory space, but also allows us to decompress only a specific part of the posting list, saving processing time as well.

The two main functions used to implement skipping are the following ones:

- **NextGEQ:** nextGEQ stands for "*next greater or equal*" and this basically describes what the method does, in fact whenever we need to move the pointer for a posting list from a given docID (*NextDocId)* to another, this function gets invoked. It simply iterates through the block descriptors for that posting list, until we find one having *MaxDocId* $\geq$ *NextDocId* which means that NextDocID could be in that specific block. Then, if we moved from the starting block, the new one gets decompressed and the posting list overwritten, otherwise we simply update the index of the posting lists to be pointed at. If no block may contain NextDocId, then *null* is returned.

- **MoveToNext:** this method is simpler than NextGEQ but equally important, since it gets used whenever we find NextDocId in a posting list, we process it (we compute the score for the term relatively to that document) then we need to move to the next item of the list. The problem is that we may need to decompress the next block and move to it, and that's what MoveToNext perform. It checks if we're at the end of the block or not, possibly decompressing the next block, overwriting the posting list and returning the updated pointer to the list. In case we're at the end of the list, then *null* is returned.

3.2 Conjunctive queries

Conjunctive queries are those one which consider and return only the documents containing all the query terms, so we need to perform an operation somehow similar to the intersection between all the posting lists. Since we have the NextGEQ method and query optimization given by blocks splitting, we adapted DAAT algorithm to perform this kind of query, instead of simply looking for the intersection as said before.

3.2.1 DAAT

Our version of DAAT for conjunctive queries primarily iterates along the shortest posting list, ensuring that the process is executed more quickly. In practice, we traverse the posting list with fewer elements using the *moveToNext* method.

We use as data structures a list of lexicon entries (one per each query term) and a list of posting lists (index), both are sorted in increasing order by termDF, which translates into posting list length, so the first element of the lexicon entries corresponds to the first element of the index and so on. The results, instead, are saved into a priority queue in order to always being able to check the document with the current lowest score inside of this structure, and possibly removing and updating it.

After identifying the new docID candidate to be returned, we proceed to verify its presence in every other posting list, using the *NextGEQ* method, until we have processed the entire shortest posting list (most frequent stop condition) or until we have processed entirely another posting list, which means that there are no more elements in the intersection.

If it turns out that one of the terms is not present in the current document, then a specific flag will be set to false, thus interrupting the loop and preventing the calculation of the score for that document. The result of this operation will be a sorted list, containing the top K (where K is a user-set parameter) documents for that query.

3.3 Disjunctive queries

When performing a disjunctive query, we must consider all the documents for which at least one query term is present, thus we cannot iterate through a specific posting list, but we must compute the given scoring function for the minimum docID not processed by far: this implies a different approach with respect to the conjunctive queries.

Once again, we implemented this as a DAAT algorithm but this time we also implemented an alternative version which includes a form of dynamic pruning to fasten the query processing: *MaxScore*.

3.3.1 DAAT

As first step, we have once again the data structures initialization: retrieval of lexicon entries, posting list and their sorting by term to overlap them also in this case, initialization of the pointers and File Channel used to read blocks.

We initialize also the nextDocId variable by calling the *minDocId* method, which specifies the lowest document Id not processed at that time, according to the lists' pointers.

Then, we iterate through the index elements by using *NextGEQ*: each time we check if the newly pointed docID is equal to nextDocId, computing and accumulating the scoring function value if so and then invoking the *moveToNext* method on that posting list, otherwise the nextDocId computation will return that same value again and again. In case we get *null* as a result from NextGEQ, it means that we have processed the entire posting list, but this time we do not stop the entire function: we remove the entry both from index and lexiconEntries since it won't give any contribution to any further document, thus is useless to keep it.

Again, the score for the document is tested against the lowest score inside the priority queue keeping the current results only if it already has K items, otherwise the newly computed score will be simply added to the priority queue.

The last step will be to update the nextDocId value, invoking *minDocId* once again, and repeating this until we get *Integer.MAX_VALUE* as result, meaning that we processed all the posting lists, and the function can then return.

3.3.2 MaxScore

Our implementation of the MaxScore algorithm can be found at the following path: *queryHandler/src/main/java/it.unipi.mircv/MaxScore.java.*

The function *maxScoreQuery* takes 3 inputs:

- query (String): The query we must process.
- k (integer): the number of couple [document,score] to return.
- isBM25 (boolean): if true it uses BM25 as scoring function, otherwise it uses TFIDF.

and it returns the list of k documents with the highest score and the related score of each document.

At the beginning of the function, we initialize the necessary data structures:

- lexiconEntries (ArrayList): An arrayList of LexiconEntry sorted in increasing order of the term upper bound.
- index (ArrayList): An arrayList of PostingList sorted in the same way of the lexicon entries, in this way the i-th element of the lexiconEntries array correspond to the i-th element of the array containing the posting lists.
- incMaxScoreQueue (PriorityQueue): A priority queue to keep the best k document with the related score in increasing order of score, in this way we can update fastly the queue removing the first element to insert an element with an highest score.
- ub (array): An array of integer to keep the upper bounds, one per posting list.

We also initialize the threshold and the pivot to zero, and we initialize the currentDocId to the minimum docId present in all the posting lists related to the terms of the query.

Then we implement a while cycle basically following the pseudocode of the MaxScore algorithm visible in the next image.

```
while pivot < n and current ≠ ⊥ do
    score ← 0
    next ← +∞
    for i ← pivot to n − 1 do                                  // Essential lists
        if p[i].docid() = current then
            score ← score + p[i].score()
            p[i].next()
        if p[i].docid() < next then
            next ← p[i].docid()
    for i ← pivot − 1 to 0 do                                  // Non-essential lists
        if score + ub[i] ≤ θ then
            break
        p[i].next(current)
        if p[i].docid() = current then
            score ← score + p[i].score()
    if q.push(⟨current, score⟩) then                           // List pivot update
        θ ← q.min()
        while pivot < n and ub[pivot] ≤ θ do
            pivot ← pivot + 1
    current ← next
```

At the end of this cycle, we invert the order of the priority queue with the k best document, score pairs, to obtain the list of the k best documents in decreasing order of score.

With this algorithm we obtain a faster response to the query because of the dynamic pruning condition visible in the pseudocode at the beginning of the cycle related to the non-essential lists. In this way, if the partial score obtained during the processing of the essential list summed to the term upper bound is less than or equal to the current threshold, the current document processing can be early terminated.

4. Performance

In the following paragraphs, we will assess, in terms of execution timing and metrics, the various alternatives implemented within the project (scoring function, algorithms, etc.) to verify the final quality of the result and how impactful some aspects are for this.

4.1 Indexing

In the following images we see the difference between the indexing performed by us in the project, that is the first one in the table (compression and preprocessing enabled), and other type of indexing without compression and/or preprocessing. In particular, the image shows the difference in the size of the created files and the indexing execution time. For the uncompressed indexing, used only for debug purposes by us, we didn't split the inverted index in 2 files, for simplicity of debugging, so, in this case, we reported only the size of the inverted index with both the fields of each posting (InvIndexDebug).

Indexing type	Duration	InvIndexDocIds size	InvIndexFreqs size	Lexicon size	InvIndexDebug
Compressed / Preprocessed	22 min	853,8 MB	39,2 MB	159 MB	/
Compressed / No preprocessed	18 min	1295 MB	63,4 MB	185 MB	/
No compressed / No preprocessed	45 min	/	/	142,2 MB	3411,5 MB
No compressed / Preprocessed	49 min	/	/	120 MB	2251 MB

4.2 Query handling

In this section, we are going to compare, in terms of response time and *trec eval* metrics, the performances of the various possible options a user can select during the interaction with the system. The benchmark queries used for this purpose are the *MSMARCO-test2020-queries* and the relative *qrels* for the evaluation.

4.2.1 TFIDF – BM25 comparison

The first comparison we want to show is between the two available scoring functions implemented in our system: TFIDF and BM25. We used *MaxScore* algorithm returning top 100 documents to produce the following results:

Scoring function	Mean response time	map	ndcg@10	recall@100
TFIDF	92ms	0.2863	0.4639	0.4745
BM25	89ms	0.3073	0.4890	0.4934

As we can see, BM25 returns slightly better performances for each metric we're taken in account, suggesting that it is a more precise and complete scoring function.

4.2.2 DAAT – MaxScore comparison

The next performance comparison we want to carry out is between the two algorithms implemented for the disjunctive queries, which are DAAT and MaxScore. For this evaluation, we're going to use BM25 as scoring function and again returning the top 100 results, but we will only show the mean response time, since the two algorithms return the same results (thus also the same metrics) and their metric performances can be observed in the table immediately above.

Ranking algorithm	TFIDF	BM25
DAAT	177ms	172ms
MaxScore	89ms	89ms

As we could expect, MaxScore is significantly faster than DAAT (almost half of the mean response time) thanks to the dynamic pruning.

4.2.3 DAAT Conjunctive

Finally, in the table below we show the mean response time obtained by DAAT for conjunctive queries over the same query collection returning the top 100 results both using TFIDF and BM25 as scoring functions.

TFIDF	BM25
31ms	30ms

4.3 Caching

We implemented a Cache of 64 KB to decrease the time of execution of the query in case of term repeated in consecutive queries. To do that we use two LruCache with maximum dimension 32 KB:

- The first cache is the one for lexicon entries, the key is the term, and the value is the lexicon entry. The size of the cache is 32KB, so we make it contains 32KB/ENTRY_SIZE_LEXICON entries to fully exploit the cache.
- The second cache is the one for the posting lists, the key is the term, and the value is the posting list. To be precise we insert only the first block of the posting lists related to the term, we work in this way because we save in memory only a block at a time for a single posting list, as explained in the previous chapters. The size of the cache is 32KB, so we decide to put 4 posting lists in the cache, knowing that the maximum size of a block of a posting list is 10KB, but often the dimension of a block is less and 4 is a medium number of terms in a query and an important part of queries are repeated ones, so it seemed like a good threshold.

The following table shows the difference between system evaluation and a free query run with and without caching: we will use MaxScore algorithm with 100 results and BM25 as scoring function to perform the system evaluation and obtain the relative query average time. To evaluate the impact on a free query, we submitted two similar queries ("*cat mouse dog animal*" and *"cat horse animal duck"*) to simulate an almost repeated query scenario so we can get the difference between using and not using the cache at its finest: in the table, we are showing the time required for the second query.

	Free query	**Evaluation**
With cache	83 ms	89 ms
Without cache	125 ms	123 ms

5. Test unit

Our project has also a few tests unit to evaluate the correctness of the implementation of the methods, they are located, for each module (except tester) into src/test/java/it/unipi/mircv; obviously all the tests methods succeeded. We will describe them following the modules.

5.1 Common

In the module **common** we have 2 test units, one for the compression and one for the preprocessing.

In the first one (**CompressionTests**) we tested both unary and variable byte compression and decompression, giving to the methods a few arrays of int (or byte in case of decompression tests) and verifying that the output of the compression (or decompression) methods was equal to the ones we calculated by hand.

In the second one (**PreprocesserTests**) we tested the parsing (including filtering and lowercase), the stemming and the global processing of a query and of a collection row-like string. Similarly to the compression, we compare the output with the hand-written results we had.

5.2 Indexer

In the module **indexer** we have 2 test units, one for SPIMI and one for the merger. To obtain the results into java objects and not into files, we used two slightly different classes, **UnitTestSPIMI** and **UnitTestMerger**, that does the same operations of the original classes, changing only the way to return the outputs.

In **SPIMITests** we performed the SPIMI algorithm on a very brief collection and then we compare the resulting index and document index to the ones we created by hand (they are respectively an HashMap<String, PostingList> and an ArrayList<Integer>).

In **MergerTests** we merged two intermediate indexes into one unique final index and we create the lexicon. Like in SPIMITest, we compare the outputs (an ArrayList<ArrayList<int[]>> and an HashMap<String, LexiconEntry>) to the objects we created by hand.

5.3 QueryHandler

In the module **queryHandler** we have 3 test units, one for DAAT, one for MaxScore and one for the equality.

The first two units, **DAATTests** and **MaxScoreTest**, test the execution of disjunctive (and, for DAAT, also conjunctive) queries, executing SPIMI and merging (the real ones) on a small collection, retrieving the document index and its stats and processing the query with both TFIDF and BM25 as scoring functions, then comparing the results with the ones we computed by hand.

The unit **EqualityTests** computes the same preliminary operations as above, then execute both DAAT and MaxScore (with both the metrics) and finally checks that the result are the same indipendently from the algorithm.