UNIVERSITÀ DI PISA

MULTIMEDIA INFORMATION RETRIEVAL AND COMPUTER VISION

A.A. 2022/23

Developed by**:**

- **Armillotta Domenico -** badge : 643020 - email: d.armillotta@studenti.unipi.it
- **Stefano Dugo -** badge : 564370 email: s.dugo@studenti.unipi.it
- **Bellizzi Leonardo** - badge : 643019 email : l.bellizzi@studenti.unipi.it

What the program can do

Our program's goal is to retrieve information from a collection of documents, given a query issued by a user; the retrieval should be fast and the information retrieved should be relevant with respect to the query, which means that the retrieved documents should hold useful information for the user. This is made possible by the implementation of two distinct programs.
The first program builds the index data structure, consisting in a vocabulary or lexicon with all the distinct terms in the collection and some useful information about them, the document table with information about the documents and the inverted index data structure, in which the posting lists with the docIDs and term frequencies for each term are stored.
The second program is responsible for the query processing: given a user query, it returns the document most similar to the query in a responsive manner.

How it works

First program - Indexer

Indexing

The first program is responsible for building the index data structure. In order to do so we decided to implement an indexing strategy based on *SPIMI (Single-Pass In-Memory Indexing)*: the program reads from the *collection.zip* file, which is the file containing the document collection, using UTF-8, and processes one document at a time (we decided to do so to speed up the merging procedure described later, instead of taking a single token at a time from the input stream), which is preprocessed like we describe in the *Preprocessing* section; each time a document is read we need to check if we have memory available to build the posting lists and the lexicon data structures, if not we write the current data to a block on the disk. We used the memory of the JVM and checked if the memory occupied didn'go over the 80% of the available memory of the JVM. For each block we create a lexicon, in which we map for each term the pointers to the posting lists, and we use an *InvertedIndex* class in which we accumulate the postings and update the statistics for the lexicon of the current block. We use a *Posting* class to keep together docIDs and term frequencies, but the two information are stored in two different files (two for each block), because we compress them with different compression algorithms. After the memory available is exhausted, the posting lists and the lexicon are written to three different files (*lexicon_nblock, docids_nblock, tfs_nblock*). We end up with three files for each block, so we need to merge them. We also make a *parameters* file in which we store total document length, the total number of documents and the average document length, which is required to compute the BM25 scoring function (that we will use later). Both the lexicon and the document index we have fixed size entries, which are made of the following fields:

- Lexicon: document frequency, collection frequency, offsets of the docIds and the term frequencies in their respective files, together with their length in bytes (along with other informations later described)

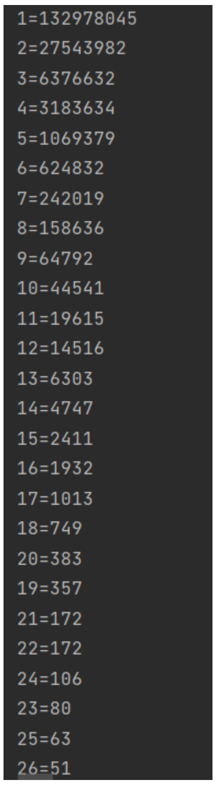
- Document index: docno (the *pid* on the collection file), the docId (generated during the indexing) and the document length; the docID starts from 1 since we don't compress 0 (nor for the docIDs neither for the term frequencies,because they will never be 0)

Merging

For the merging phase we adopted a two-way merging, merging the blocks in pairs, until all the files of the blocks are merged. We scan the lexicon for both files and check if one word comes lexically before the other or not, or if they are the same and the posting lists need to be merged. We also need to check if one file is finished while the other is not.

Compression

After merging the blocks, we need to compress the index to reduce the space occupied in the disk. We applied integer compression on the docIds and term frequencies files; these files were the most space consuming and so compressing them may improve retrieval performances.
For the docIds we decided to implement Variable Byte compression, since the values of the docIds could reach more than 8800000, so among the other integer compression algorithms looked like the best option.
For the term frequencies we decided to implement unary compression; since the term frequencies followed the Zipf's law (very very many infrequent terms, very very few frequent terms), given that the elements with term frequency equal to 1 were more than one hundred million, it looked like a really good option since we use very often a byte instead than four (Java is byte aligned, so the minimum that we could do was using one byte), so we expect a compression factor of 4 for term frequencies.



```
1=132978045
2=27543982
3=6376632
4=3183634
5=1069379
6=624832
7=242019
8=158636
9=64792
10=44541
11=19615
12=14516
13=6303
14=4747
15=2411
16=1932
17=1013
18=749
20=383
19=357
21=172
22=172
24=106
23=80
25=63
26=51
```

Figure on the right: term frequencies distributions (tf values on the left, number of occurrences of the values on the right)

Additional information

Other than all the informations previously described, for our retrieval task we needed to store other informations on our files:

- *Idf* (*Inverse Document Frequency)*: this field is required to compute the scoring functions we implemented (both BM25 and TFIDF, as described later); we decided to compute it during the indexing phase and store it in the lexicon to avoid to compute it during query processing for each term; moreover, the idf field we computed is the logarithm of the idf, so we don’t need to compute it later.

- *Term Upper Bounds:* this values are required to implement the *MaxScore* pruning algorithm for speeding up *document-at-a-time* query processing (as described later); since we need all the posting lists to compute all of them, it can be very time consuming to do during query processing, so we do it at the end of the indexing phase. We computed term upper bounds both with TFIDF and with BM25, to make possible to choose which function to use in query processing (implemented in the *Scorer* class)
- *Skip information:* in order to implement skipping in query processing, for each posting list we divided it in blocks, with length equal to the square root of the number of postings (if the remainder is not zero the last block will have less postings). We needed to store in the lexicon the offset and the length of the skip info to distinguish them between the information about the lists; since we compress the lists in two different ways, we decided to store them in a separate file, the *skipInfo* file.

This information is computed along with the compression of the lists, so that the information in the lexicon (offsets and lengths) are adjusted correctly.

Preprocessing

As mentioned before, we had to preprocess the documents before computing all the posting information; we implemented two indexes, one containing all the stop-words and another without stop-words and stemming all the words of the lexicon. As shown in the results section, the first index is almost twice as large as the second one.

Second program - Query Processing

Some considerations

In query processing, we are interested in retrieving the relevant documents with respect to a user's query in the fastest way possible; so we can't just blindly load into memory all the posting lists and the dictionary because it's really time and space consuming (they obviously don't fit in memory). When taking a query in input, we preprocess it in the same way we preprocessed the documents in the index and then we retrieve pointers and statistics from the lexicon through binary search in the *lexicon* file (that is why the lexicon is made of fixed entries, with the term as a key for the binary search). We still load into memory the document index using the *DocumentIndex* class, since we need to access it to compute the BM25 scoring function and accessing each time to the disk is very time consuming (we would need to access it for each new document we process, so A LOT of time!). The process of loading the document index in memory takes approximately 10 seconds.
The queries are preprocessed in the same way as the document collection

Processing strategy

The strategy we decided to adopt for query processing was *document-at-a-time* query processing; we implemented both conjunctive and disjunctive queries:

- For conjunctive queries, we process each document and for each list we check if the docId is present in all the lists; if it's not we go to the next document, otherwise we compute the score
- For disjunctive queries, we process all the documents in a disjunctive (OR) way; this can be very time consuming, so we decided to optimise this kind of query with a dynamic pruning algorithm

Pruning algorithm

We decided to implement the *MaxScore* pruning algorithm; we use the term upper bounds computed at the end of the indexing phase, sorting the terms of the query in increasing order of term upper bounds. Then, for each term in the list, we compute the document upper bounds, also sorted in increasing order, that will be used to divide the lists between essential and non-essential. We set the threshold equal to 0 and when the heap with the partial score reaches size k (the rank we are computing) we select the smallest value as the threshold and update the lists if it goes above document upper bounds; the non essential lists will be skipped if the document upper bound plus the partial score is not above the threshold; the essential lists are computed in pure disjunctive mode.

Skipping

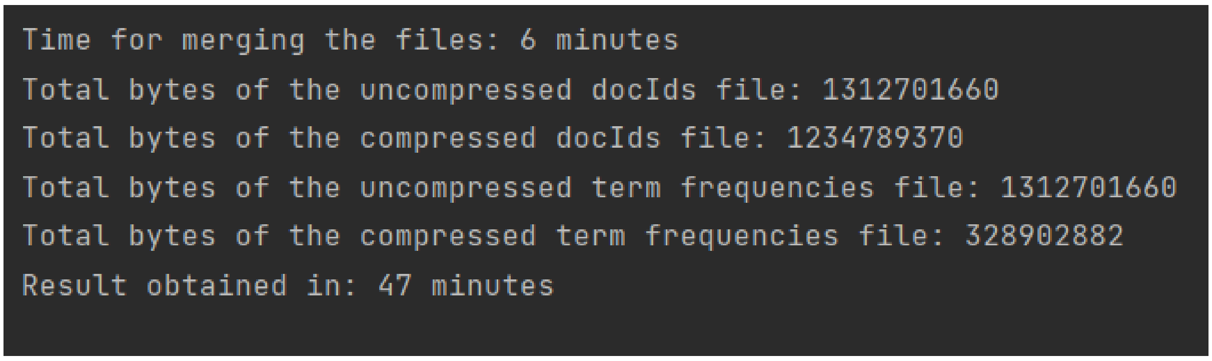
Since we do not want to retrieve the full posting lists and decompress them entirely, because it takes time, we decompress the lists in blocks, reading one block at a time using the skip information and updating the offsets while we read different blocks. We use the *openList* and *nextGEQ* methods: we take the first block of the two lists and iterate through them using nextGEQ, that takes as the next docID the first which is greater or equal than the current one; we check the end docId obtained from the skip info to check if we have finished the current block; in that case we go to the next block calling again the openList method. So, we decided to keep in memory the blocks to process them until all the docIDs of the block are processed, then substitute the block with the next one, updating the number of blocks read and the current end docID.

Caching

We implemented a cache, using the *LruCache* class, that maintains for each term of the query its pointers and statistics, so that if a new query contains at least one of those terms we don't need to retrieve them from the whole lexicon; the LRU policy of the cache is managed by the class.

Performance (Time & Space)

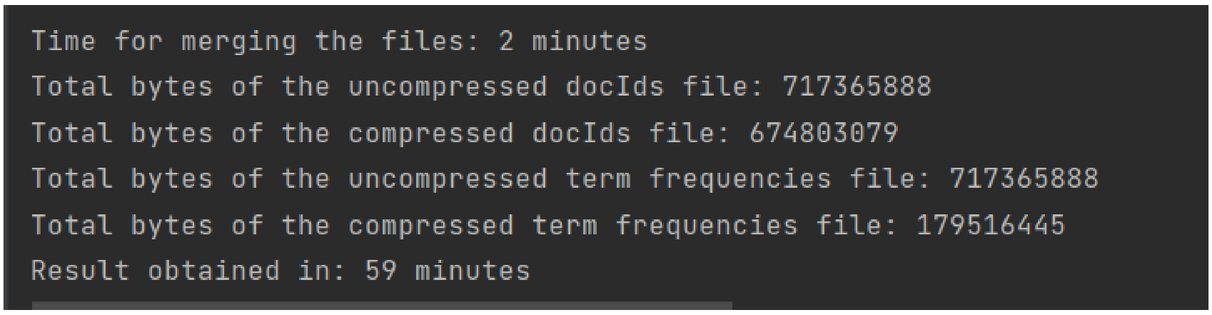
Time for building the index with the dimensions of the docIds and term frequencies files (in bytes) before and after compression, including the computation of the additional structures:



```
Time for merging the files: 6 minutes
Total bytes of the uncompressed docIds file: 1312701660
Total bytes of the compressed docIds file: 1234789370
Total bytes of the uncompressed term frequencies file: 1312701660
Total bytes of the compressed term frequencies file: 328902882
Result obtained in: 47 minutes
```

The number of blocks the program created for building this index during the indexing phase was 15.
The average document length in this case is 53.0761689077015.

Time for building the index, without stop-words and stemming, with the dimensions of the docIds and term frequencies files (in bytes) before and after compression, including the computation of the additional structures:



```
Time for merging the files: 2 minutes
Total bytes of the uncompressed docIds file: 717365888
Total bytes of the compressed docIds file: 674803079
Total bytes of the uncompressed term frequencies file: 717365888
Total bytes of the compressed term frequencies file: 179516445
Result obtained in: 59 minutes
```

The number of blocks the program created to build this index was 7.
The average document length in this case is 27.53696980814929.

Example: comparison between different indexes and scoring methods
Results for the top 10 ranking for the query: "*The capital of the USA*":

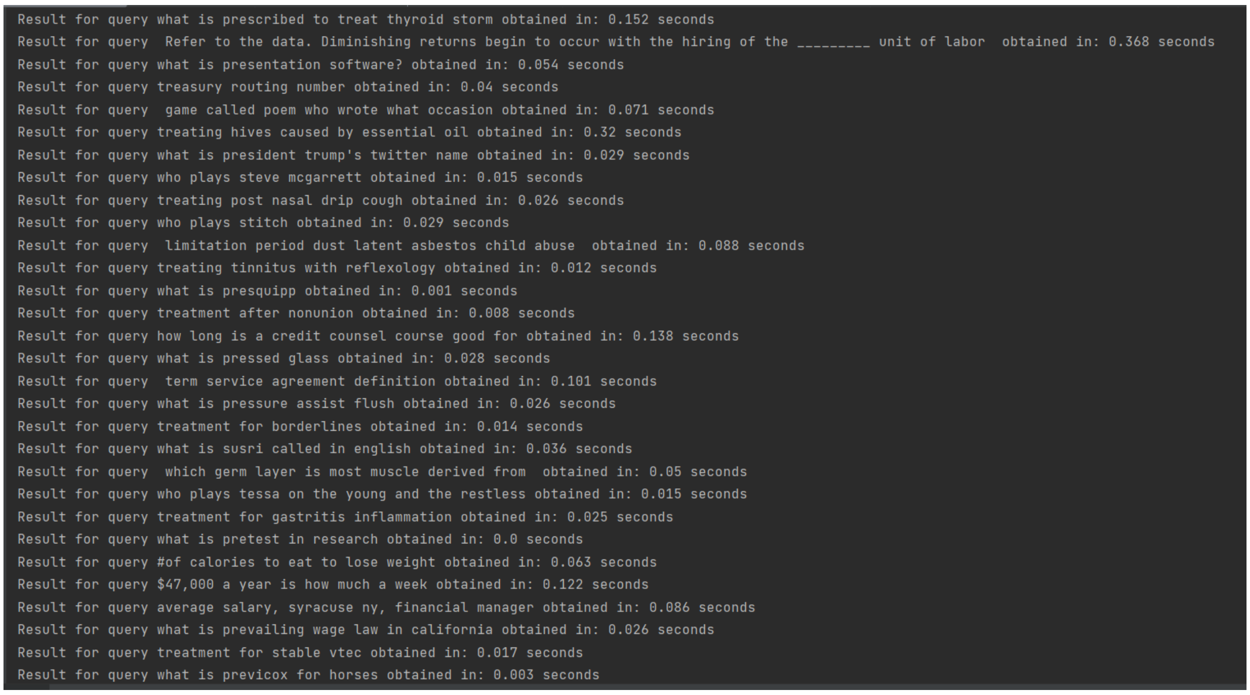
Index	**Scoring Function**	**Mode**	**Time**	**Top Document**
Unfiltered	TFIDF	Disjunctive	0.143 seconds	8829878
Unfiltered	BM25	Disjunctive	0.115 seconds	8829878
Unfiltered	TFIDF	Conjunctive	0.896 seconds	8686482
Unfiltered	BM25	Conjunctive	0.904 seconds	**8686485**
Filtered	TFIDF	Disjunctive	0.16 seconds	1000002
Filtered	BM25	Disjunctive	0.135 seconds	**8686485**
Filtered	TFIDF	Conjunctive	0.146 seconds	4179417
Filtered	BM25	Conjunctive	0.119 seconds	**8686485**

(Note: the times displayed here are an upper bound since they are computed running the program from scratch for each query; if run one after the other, the times are much less, as displayed in the next example)

In this example we can see that in three cases, which all use bm25 as a scoring function, we get the same top document; if we check the document we can see that it contains the answer of the query:

*8686485 This page displays the time difference between Lusaka (capital of Zambia) and Washington DC (**capital of USA**).To see the time difference for other cities in Zambia* *and USA use the time difference calculator to the right.his page displays the time difference between Lusaka (capital of Zambia) and Washington DC (**capital of USA**).*

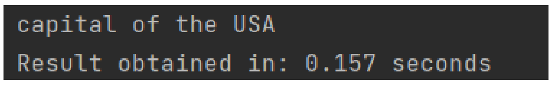
Example: times for processing the first 30 queries of the queries.eval.tsv file with disjunctive Daat with BM25



```
Result for query what is prescribed to treat thyroid storm obtained in: 0.152 seconds
Result for query  Refer to the data. Diminishing returns begin to occur with the hiring of the _________ unit of labor  obtained in: 0.368 seconds
Result for query what is presentation software? obtained in: 0.054 seconds
Result for query treasury routing number obtained in: 0.04 seconds
Result for query  game called poem who wrote what occasion obtained in: 0.071 seconds
Result for query treating hives caused by essential oil obtained in: 0.32 seconds
Result for query what is president trump's twitter name obtained in: 0.029 seconds
Result for query who plays steve mcgarrett obtained in: 0.015 seconds
Result for query treating post nasal drip cough obtained in: 0.026 seconds
Result for query who plays stitch obtained in: 0.029 seconds
Result for query  limitation period dust latent asbestos child abuse  obtained in: 0.088 seconds
Result for query treating tinnitus with reflexology obtained in: 0.012 seconds
Result for query what is presquipp obtained in: 0.001 seconds
Result for query treatment after nonunion obtained in: 0.008 seconds
Result for query how long is a credit counsel course good for obtained in: 0.138 seconds
Result for query what is pressed glass obtained in: 0.028 seconds
Result for query  term service agreement definition obtained in: 0.101 seconds
Result for query what is pressure assist flush obtained in: 0.026 seconds
Result for query treatment for borderlines obtained in: 0.014 seconds
Result for query what is susri called in english obtained in: 0.036 seconds
Result for query  which germ layer is most muscle derived from  obtained in: 0.05 seconds
Result for query who plays tessa on the young and the restless obtained in: 0.015 seconds
Result for query treatment for gastritis inflammation obtained in: 0.025 seconds
Result for query what is pretest in research obtained in: 0.0 seconds
Result for query #of calories to eat to lose weight obtained in: 0.063 seconds
Result for query $47,000 a year is how much a week obtained in: 0.122 seconds
Result for query average salary, syracuse ny, financial manager obtained in: 0.086 seconds
Result for query what is prevailing wage law in california obtained in: 0.026 seconds
Result for query treatment for stable vtec obtained in: 0.017 seconds
Result for query what is previcox for horses obtained in: 0.003 seconds
```

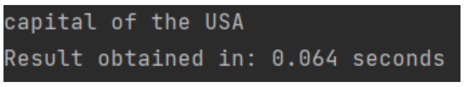
Example: times for processing a query before and after caching the terms with disjunctive Daat with BM25

Before:



```
capital of the USA
Result obtained in: 0.157 seconds
```

After:



```
capital of the USA
Result obtained in: 0.064 seconds
```

Limits

- Compression: with the unary compression of term frequencies, the term frequencies file is compressed by a factor of approximately 4; instead, the docIDs file has a much much lower compression rate due to the fact that variable byte for many values of the docIDs doesn't spare that much bytes, even though the dimension of the file is less than the uncompressed one; to a further reduction of the space we could have tried different algorithms and also using d-gaps instead of the full docIDs

- We didn't perform a full effectiveness analysis on the performance of our system, for instance we tried to use *trec_eval* but for some issues we didn't manage to obtain the results, so we can't say much about retrieval performances from that point of view

- We stored in our files some information that could be redundant, for instance we stored the collection frequency but we never used because at the end we didn't decide to implement a language model to score the documents with respect to the queries; also we used separate files for some additional parameters and for the skipping information

- If the query doesn't contain terms present in the vocabulary, the result is meaningless (the returned docID is a docID which is not present in our index)

- We use too much space during indexing; for instance, since the JVM frees memory when it wants to, sometimes we end up with blocks with few documents, since memory is still not completely free

- We don't have optional query processing for the two types of index; we decided to leave in the final version only the query processing on the filtered index, if we want to test the queries on the non-filtered index, we need to change the files and call the right preprocessing algorithm in the *Daat* class.

Major functions

Preprocessing Phase

1. **Normalisation :**
   First step where the following characters were removed with regex: non ascii , non printable , punctuation, digits, single characters. Then the remaining tokens are converted to lowercase.

2. **Tokenization :**
   Split the input document into tokens.

3. **Stopword Removal (in the filtered index) :**
   English language stops words removed from tokens.

4. **Stemming (in the filtered index):**
   The PorterStemmer was used for stemming document tokens.

Indexing

The main class for the indexing phase was the *SPIMI* class, in which the indexing algorithm, the merging algorithm and the algorithm which creates the additional structures are implemented (along with the *InvertedIndex* class in which the postings and the lexicon for each block are created during indexing). The main method is in the *Indexer* class.

Query processing

The main class for query processing is the *Daat* class, in which the query processing methods (conjunctive and disjunctive) and the interfaces for query processing are implemented. The main method is in the *QueryProcessor* class.

Compression

Two types of compression were used, since the docIDs and the term frequencies have a different distribution, and they were saved on different files.
For the term frequency, **Unary** compression was used.
For the doc id, **Variable Byte** compression was used.
For both compression algorithms, we implemented a method to compress a single integer converting the code in bytes and writing it to the file, and a decompression algorithm that decompresses the whole input (we decompressed only blocks on the list so we used it to decompress a block at a time). The algorithms are implemented in the *Compresso*r class.

Dynamic Pruning Algorithm

As said before we implemented the *MaxScore* pruning algorithm: the division of essential and non-essential lists is done using a pivot variable that defines the start of essential lists (terms are stored in an array in increasing order of term upper bound); then this pivot variable is updated each time we increase the threshold, if needed (if the threshold is above an upper bound we increase the pivot variable until we find an upper bound greater than the threshold).

Interfaces for the query processing

- **NextGEQ**: the next document to process is the one with a value which is greater or equal than the current one; we keep iterators on the current blocks of the lists and update them when we processed the nextGEQ document
- **OpenList:** we use the skip info to open the current block of the lists and update the offsets and number of blocks read, and we assign the iterators on the newly opened blocks
- **GetFreq**: we used the *LexiconStats* class to store the current term frequency for the current docID to process for each term and we get the term frequency from there

Other classes

- **LexiconEntry:** to store a single entry of the lexicon, with the term as a key and a LexiconStats object as a value
- **ScoreEntry:** to build the priority queue to store the documents ordered in decreasing order of score
- **TermUB:** to map terms to their term upper bounds in increasing order of term upper bounds
- **Utils:** contains some methods to create some data structures and doing binary search on the lexicon