ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING
project report for the examination
of
Multimedia Information Retrieval and Computer Vision

Student:
Alessandro Diana

AA. 2022/2023

Contents

List of Figures

1 Introduction

1.1 Project description

This report explains how the project was designed its various parts, decisions, and implementation steps.
The project concerns the topic of Information Retrieval and in particular the implementation of a local search engine developed entirely in Java. This program must be able to:

- read documents from the collection using UNICODE and handling possible form errors;
- create an inverted index from a collection of given documents;
- execute on the generated inverted index the queries, both conjunctive and disjunctive, entered by the user;
- read user-entered queries via simple line commands;
- display the results (i.e., pids) for each query executed.

Then other optional features are:

- reading the compressed collection and decompressing it at runtime;
- stemming and stopword removal;
- inverted index compression;
- implementation of another scoring function in addition to TFIDF (in our case BM25);
- optimization in query execution by implementing skipping and a dynamic pruning algorithm (in our case Max Score).

In the version discussed in this paper, the project has all the points listed. In addition, a performance evaluation part was also done using a list of predefined queries.

1.2 Code and Files organization

All the developed code and documentation can be found at this link:
https://github.com/Arzazrel/Project_MIRCV.
The readme file in the repository contains all the instructions and explanations concerning the code, documentation, and operations required to import and run the program.

1.2.1 Code

All codes created are contained in the 'Search_engine/src/main/java/it/unipi/dii/aide/mircv' folder.
In this folder there are the main classes and some sub-folders for organize the class of the code. Inside this folder are the following subfolders:

- 'compression' containing all the classes for the compression and decompression of the posting lists;

- 'data_structures' containing all the classes concerning the data structures of the project;
- 'utils' containing utility and support classes.

1.2.2 Files

All files created and used by the code are contained in the 'Search_engine/src/main/resources' folder.
The files that are in this folder and are not organized in any sub-folders are:

- 'collection.tar' that is the compressed document collection used for the project;
- 'stopwords' that is the list of the English stopwords;
- 'collectionStatistics' containing the collection statistics;
- 'flags' containing the values for the flags.

Inside this folder are the following sub-folders:

- 'merged' containing the final file for the inverted index;
- 'partial' containing the partial file of the inverted index construction (partial inverted index);
- 'test' containing the compressed test queries collection;
- 'upperBound' containing the precomputed logarithm for the term frequency values and the terms upper bound.

1.3 Document collection

The collection of documents used can be found at this link: https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020
.
The collection contains 8.841.823 documents and has a size, when unzipped, of about 2.85 GB.
Each document is represented by a single line with the following format: {docno}{$\widehat{\text{content}}$}

Other features of the corpus collected during the creation of the inverted index are:

- number of empty document in the collection: 124;
- len of the shortest doc in the collection: 1;
- len of the longest doc in the collection: 287 or 283(stopword removal).
- average length of the document in the collection: 58 or 33(stopword removal).

2 Indexing

In this step, all the documents in the collection are scrolled and analyzed to create the inverted index.
The creation of the inverted index is divided into two main parts:

- In the first one single-pass in-memory indexing (called SPIMI) is performed in which partial information will be collected. More details will be explained in the following section;
- In the second, merging of the collected partial information is done to obtain the complete inverted index.

In both phases, information regarding the whole collection will be collected for the design phase or for carrying out some algorithms. This information can be viewed through the user interface and will be enclosed in the class 'CollectionStatistics' and will be saved on the disk in the file 'collectionStatistics'.
At the end of these main phases, all the structures of the inverted index will be complete and saved in memory. In particular, the main files saved will be:

- 'dictionary' containing the Dictionary;
- 'docID 'containing the posting list of doc id;
- 'termFreq' contains the posting list of term frequencies;
- 'documentTable' containing the Document Table;
- 'skipInfo' containing the information needed to perform skipping.

2.1 Flags

Flags are variables that manage the behavior of the program. These flags are included in the 'Flags' class and can be viewed and modified by users through the user interface.
Each flag is a Boolean variable that is also saved on disk in the 'flags' file.
The flags in the project are as follows:

- 'sws_flag': if 'true' stop words removal and stemming are enabled. If 'false' stop words removal and stemming are disabled;
- 'compression_flag': if 'true' the compression of the posting lists is enabled. If 'false' the compression is disabled;
- 'scoring_flag': if 'true' the scoring function used is BM25. If 'false' the scoring function used is TFIDF;
- 'skip_flag': if 'true' the skipping is enabled. If 'false' the skipping is disabled;
- 'qdPruning_flag': if 'true' the queries are executed with a dynamic pruning algorithm (Max Score). If 'false' the queries are executed with the simple DAAT algorithm;
- 'wholePLInMem_flag': considered only if skipping and compression are both enabled. If 'true' the compressed posting lists will be loaded completely in memory and not in blocks. The decompression is done in blocks and the decompressed version is only present in memory for the current block for each query posting list.

2.2 Text processing

The text is processed to clean up any errors and be able to go to individual terms in a document or query. This process is necessary both for building the inverted index and for query execution. The same kind of text processing must be done both on the documents in the collection to build the inverted index and on the queries that are then to be run on that inverted index.
Four different techniques (which will be explained in the next sections) can be applied during text processing: text cleanup, tokenization, stop removals, and stemming.
These operations are done by the "TextProcessor" class.

2.2.1 Text cleaning

In this step, the text is cleaned up and formatted in the most useful way for the creation of the inverted index.
It is a fundamental and mandatory step, and through regular expressions, the following operations are performed:

1. remove URL: removes any URLs present in the text;
2. reduction to lowercase: to make no distinction between the same words but written differently all uppercase characters are converted to the corresponding lowercase character;
3. removal HTML tags: removes any HTML tags present in the text;
4. removal punctuation: removes punctuation (is not useful to our task);
5. removal Unicode chars: remove unicode chars in the text;
6. removal extra whitespaces: removes any additional spaces, so the terms will be divided by a single space.

2.2.2 Tokenization

In this step, the text is divided according to whitespace such that a list of tokens is returned. In this way, the text of each document (and query) is transformed into an array of tokens.
It is a fundamental and mandatory step.

2.2.3 Stop removal

This step is optional and the user can choose to do so via the "sws_flag" flag.
This is the step that removes "stop words," which are words that appear very frequently (e.g., articles) but without making significant meaning or contribution to the analysis of the text.
Removing these words helps to greatly reduce the size of the text (documents and queries therefore also the inverted index) without removing meaning.
To do this step it needs a collection containing all the "stop words" to be removed. This collection changes depending on the language, in our project English language is used (the file is called stopwords.txt).

2.2.4 Stemming

This step is optional and the user can choose to do so via the "sws_flag" flag.
This is the step that reduces the terms into stems, removing the end parts (e.g., gender and number). This is an irreversible process and each language needs its stemmer.

In our case, we use a stemmer for English, and specifically, we used the java class "org.tartarus.snowball.ext.PorterStemmer" which implements the widely used Porter Stemming algorithm.
This step tends to reduce the number of terms in the dictionary and increase their posting list. This is because several interrelated terms will be merged into a single dictionary term (causing a reduction in the number of terms) and then the elements of the posting lists of the related terms will be merged into the posting list of the resulting term (causing an increase in the length of the posting list).
For example, in this collection, with only tokenization the number of terms in the dictionary is 1.437.891 and the average length of the posting lists is 247.
Instead in the case of stemming and stopword removal, the number of terms in the dictionary is 1.237.531 and the average length of the posting lists is 174(caused also by the stopword removal step).

TYPE	NAME	BYTES	DESCRIPTION
INT	nDocs	4	Number of documents in the collection
LONG	totDocLen	8	Sum of the all document length in the collection
DOUBLE	avgDocLen	8	Average document length (used in BM25 scoring function)
DOUBLE	k	8	Used in BM25
DOUBLE	b	8	Usend in BM25
INT	emptyDocs	4	Number of empty docs in the collection
INT	malformedDocs	4	Number of malformed docs in the collection
INT	minLenDoc	4	Len of the shortest doc in the collection
INT	maxLenDoc	4	Len of the longest doc in the collection
INT	maxTermFreq	4	Max term Frequency in the collection
INT	maxPLLength	4	Length of the longest posting list in the collection
INT	minPLLength	4	Length of the shortest posting list in the collection
DOUBLE	avgPLLength	8	Average length of the posting lists in the collection
LONG[]	mostTFOcc	8*len	Array of the most common TF occurrence in raw value
DOUBLE[]	mostTFPerc	8*len	Array of the most common TF occurrence in percentage value
INT[]	mostTF	4*len	Array of the most common TF value
DOUBLE	avgDIDGapInPL	8	The average gap between DID of the same posting list (average in the whole collection)
DOUBLE	minAvgDIDGapInPL	8	The min avg gap between DID of the same posting list (average in the whole collection)
DOUBLE	maxAvgDIDGapInPL	8	The max avg gap between DID of the same posting list (average in the whole collection)
DOUBLE	avgBlockDIDGapInPL	8	The average gap between DID of the same block (average in the whole collection)
DOUBLE	minBlockAvgDIDGapInPL	8	The min avg gap between DID of the same block (average in the whole collection)
DOUBLE	maxBlockAvgDIDGapInPL	8	The max avg gap between DID of the same block (average in the whole collection)

Figure 1: Illustrative table of the fields of collectionStatistics.

2.3 Data structures

All data structures used in our project are located in the folder 'data_structures'. They are used to hold information that will be written to or read from disk and are (written in lexicographic order):

- CollectionStatistics;
- DictionaryElem;
- DocumentElement;
- Flags;
- Posting;
- SkipInfo;

They are analyzed in the following subsection.

2.3.1 CollectionStatistics

Used to keep statistics related to the collection and used both for further analysis and algorithms during query execution.
The fields are shown in Table 1.

Considering the default value of 5 for the array, in total this structure occupies 220 Bytes in memory.

2.3.2 DictionaryElem

Used to keep information linked to a term in the collection.
The fields are shown in Table 2.

TYPE	NAME	BYTES	DESCRIPTION
STRING	term	20	The term.
INT	df	4	Document frequency, number of documents in which there is the term
INT	cf	4	Collection frequency, number of occurrences of the term in the collection
LONG	offsetTermFreq	8	Starting point of the posting list of the term in the term freq file (set in partial(SPIMI) and total index)
LONG	offsetDocId	8	Starting point of the posting list of the term in the docid file (set in partial (SPIMI) and total index)
INT	docIdSize	4	Dimension in byte of compressed DocID of the posting list
INT	termFreqSize	4	Dimension in byte of compressed termFreq of the posting list
LONG	skipOffset	8	Offset of the skip element
INT	skipArrLen	4	Len of the skip array (equal to the number of skipping block)

Figure 2: Illustrative table of the fields of a dictionary element.

With compression and skipping disabled, each term occupies 44 Bytes.
With only compression enabled each term occupies 52 Bytes.
With only skipping enabled each term occupies 64 Bytes.
For each term found in the collection, there is an instance of this class.

2.3.3 DocumentElement

Used to maintain the information on a single document across the whole collection.
The fields are shown in Table 3.

TYPE	NAME	BYTES	DESCRIPTION
INT	docid	4	DocID of the document
STRING	docno	10	DocNO of the document.
INT	doclength	4	Length (number of word) of the document
DOUBLE	denomPartBM25	8	First part of denominator, saved for optimization purpose

Figure 3: Illustrative table of the fields of a document element.

In total, a DocumentElement occupies 26 B. For each document found in the collection, there is an instance of this class.

2.3.4 Flags

Used to maintain the flags indicating the behavior required by the user for the creation of the inverted index and the parameters for query execution.
The fields are shown in Table 4.

TYPE	NAME	BYTES	DESCRIPTION
BOOLEAN	sws_flag	1	true = stop words removal and stemming enabled, false = stop words removal and stemming disabled
BOOLEAN	compression_flag	1	true = compression enabled, false = compression disabled
BOOLEAN	scoring_flag	1	true = scoring enable, false = scoring disable
BOOLEAN	skip_flag	1	true = skipping enable, false = skipping disable
BOOLEAN	qdPruning_flag	1	true = query executed with dynamic pruning algorithm (Max Score), false = query executed with classic DAAT algorithm
BOOLEAN	deletePartFile_flag	1	true = delete the partial file after complete the indexing, false = doesn't delete partial file after complete the indexing

Figure 4: Illustrative table of the fields of flags.

The fields are of type boolean but are saved on disk as integers so in total the flags occupy 28 Bytes.

2.3.5 Posting

Used to maintain the information on a single posting for a term.
The fields are shown in Table 5.

TYPE	NAME	BYTES	DESCRIPTION
INT	docId	4	The DocID of the document
INT	termFreq	4	The occurrence of the term in the document identified by docId.

Figure 5: Illustrative table of the fields of posting.

Each posting occupies 8 bytes and all postings relating to a term constitute the posting list for that term.

2.3.6 SkipInfo

Used to maintain skipping block information.
The fields are shown in Table 6.

TYPE	NAME	BYTES	DESCRIPTION
INT	maxDocId	4	The maximum DocID in the skipping block
LONG	docIdOffset	8	Offset of the first docID in the next skipping block
LONG	freqOffset	8	Offset of the first termFreq in the next skipping block

Figure 6: Illustrative table of the fields of skipInfo.

In total, each skipInfo occupies 20 bytes and all the skipInfo's related to a term make up a skipList containing all the skipping information for the posting list (of that term).

2.4 SPIMI

For the construction of the inverted index, I use the SPIMI algorithm, which proves to be more scalable than other indexing algorithms. With this algorithm, one document at a time is read from the collection, and its text is processed, tokenizing it.
Each document is assigned a DID (Doc ID), which is an incremental integer starting from 1 and identifies the document in our system.
During execution, one token at a time is processed for which it checks whether it is already present in the vocabulary or not. If it is already present (i.e. it has already been found previously in the same or a proceeding doc) the related posting list and statistics are updated. If it is not present, it will be added to the vocabulary and the related posting list will be created.
The construction of the inverted index is divided into blocks, each of which is kept in main memory during the construction process and is transferred to disk when it has reached the maximum amount of memory to be used (in our case 80%) so that we can free it up and start the construction of a new block.
All blocks will be merged during the merge process to obtain the final data structures.
In this phase, the various files containing all the partial information necessary for the construction of the complete inverted index are created. These files are saved on disk in the 'partials' folder and are:

- 'blocks' which contains the start offset of each block;
- 'partial_dictionary' which contains the partial dictionary made at each block;
- 'partial_docID' which contains the DID values of the partial posting lists;
- 'partial_termFreq' which contains the termFreq values of the partial posting lists.

In this phase, the final DocTable is also created and saved on disk in the 'documentTable' file in the 'merged' folder. The document statistics (number, composition, length) and the maximum value for the term frequency are collected and saved.
The java file that handles this part is 'PartialIndexBuilder'.

2.5 Merging partial index

In this crucial part of the entire program, the partial blocks made by SPIMI are taken. Each of them consists of an independent inverted index, and are combined to create a single and complete inverted index that will be used in query execution.
Since each temporary dictionary is sorted alphabetically, it reads and inserts all the first terms stored in each block into a priority queue, together with the index indicating the block number to which it refers.
Then until the priority queue is empty, it extracts the first term in lexicographic order and replaces it with the next term in the same partial block, if present.
For each term, the final information will be created as a merging of the partial ones. For the current term, the partial information is read from the record, and if the term is the same as the term taken previously then the information is merged (case where you have the same term in different blocks and therefore have several partial posting lists to merge). To merge the posting list of two equal terms simply concatenate them, because the documents are processed in ascending DocID order during the SPIMI algorithm.
If the current term differs from the previous one, it means that all information relating to that term has been taken and merged and will then be saved to disk in the complete inverted index. Depending on the flags enabled before saving the posting lists, either compression or skipping or both will be performed. Finally, the complete inverted index is saved in the files seen in Chapter 2.0.
At this stage, statistics concerning the DID Gap in the posting lists and the size of the posting lists are also calculated and saved.
The java file that handles this part is 'IndexMerger'.

2.6 Compression

In this type of application, the final inverted index is very large and takes up a lot of disk space. To limit the space used on the disk and make it easier to store the posting lists, they are compressed. Compression is the operation in which a piece of data is transformed into another representation that occupies less space.
The data to be compressed, which make up the posting lists, are the DocIDs and term frequencies. Both of these fields consist of integers but have different properties, which is why two different compression algorithms (which are described in the following sections) were used in this project.

2.6.1 Term frequency compression (Unary)

The term frequency values in the posting list aren't in an ordered list, the values can be different and do not depend on the preceding or following value. For these reasons, a compression system operating on individual values must be chosen.
In theory, the frequency of terms in the docs should have low values.
To be able to choose the best possible compression algorithm, the program, when indexing and calculating the Term Upper Bound, collects and calculates statistics concerning the term frequency of the terms in the collection. The maximum value for the term frequency for both stopwords removal and stemming is 117.

In the case of tokenization only, the five most frequent values of the term frequency are:

1. TermFreqValue: 1, occurrence: 263587196 (row value), 74,0723 %
2. TermFreqValue: 2, occurrence: 61313810 (row value), 17,2302 %;

3. TermFreqValue: 3, occurrence: 15724375 (row value), 4,4188 %;
4. TermFreqValue: 4, occurrence: 8241334 (row value), 2,3160 %;
5. TermFreqValue: 5, occurrence: 3003637 (row value), 0,8441 %.

In the case of stemming and stopwords removal, the five most frequent values of the term frequency are:

1. TermFreqValue: 1, occurrence: 167281659 (row value), 77,7697 %
2. TermFreqValue: 2, occurrence: 34062446 (row value), 15,8357 %;
3. TermFreqValue: 3, occurrence: 7449871 (row value), 3,4635 %;
4. TermFreqValue: 4, occurrence: 3719373 (row value), 1,7291 %;
5. TermFreqValue: 5, occurrence: 1213390 (row value), 0,5641 %.

In both cases, the percentages are very similar and shows that the most common value is 1 with a large majority. The frequency decreases as the value of the term frequency increases. This is a perfect scenario for algorithms that save space the smaller the number to be represented.
This is why I have chosen Unary code as the compression algorithm. This algorithm represents a positive integer x with a series of (x-1) 1 followed by a 0. U(x) = $1^{x-1}0$.
In this way, a compressed number occupies many bits equal to its value.
In Java (the language chosen for the program in the project) an integer is represented on 32 bits so to have an advantage we must represent values less than 32 which is more than 99.9% of the total cases.
With this algorithm, in the case of tokenization only, the size of the term frequency values in the posting lists changes from 1.326 GB to 61.857 MB, gaining 95.44%.
With this algorithm, in the case of stop words removal and stemming, the size of the term frequency values in the posting lists changes from 820.536 MB to 35.212 MB, gaining 95.71% .
As the data show, this algorithm in this case leads to a very high gain in space.

2.6.2 DocID compression (Variable bytes)

DocID values in the posting lists are sorted in strictly ascending order, and compression techniques that exploit this property can be used. The DocIDs have positive values and in this collection range from 1 to 8841701, so you are dealing with very high numbers.
I have chosen to use "Variable-Byte codes" as a compression algorithm that operates at the byte level by representing an integer over a variable number of bytes.
For each Byte, 7 bits will be dedicated to the representation of the number while 1 (the most significant) will be for control.
The binary representation of the number is divided into parts of 7 bits which will each be put into a byte which will have the control bit at 0 if it is not the last byte of the representation for that number and 1 if it is the last.
E.g. the integer x = 67822 has a binary representation = 10000100011101110 on 17 bits. The variable byte representation will be on 3 bytes = 00000100 00010001 11101110.
In addition, I can exploit the fact that the DIDs are in an ordered list by using DGaps.
It compresses the first complete DID and then for subsequent DIDs it compresses the difference with the previous one. This saves space because the difference between two integers will take up much less space than the larger number. This technique saves additional space at the cost of having to decompress all the previous values if we only want one item in the list.

E.g. take two integers x = 67822 and y = 67922. Their binary representation is bin(x) = 10000100011101110 and bin(y) = 10000100101010010. Their difference is 100 which in binary is bin(100) = 1100100. Both numbers using only the variable-byte code would occupy 3 Byte. If they are considered to belong to a list ordered with DGap the first number (x) will be compressed and saved as seen before while the second number will be represented with the compressed form of the difference (100) occupying only one byte instead of 3.
In the case of no skipping, the posting lists will be compressed as a single list, while in the case of skipping, each block of the posting list will be considered a separate list (with the first item compressed as a whole and not as a difference from the previous one).
With this algorithm, in the case of tokenization only, the size of the DocID values in the posting lists changes from 1.326 GB to 447.717 MB, gaining 67.02%. With this algorithm, in the case of stop words removal and stemming, the size of the DocID values in the posting lists changes from 820.536 MB to 285.886 MB, gaining 65.16% .
As the data show, this algorithm in this case leads to a high gain in space.

3 Query processing

After the complete inverted index has been constructed, when the user runs the program it reads from the disk the information it needs to be able to execute queries. This information are the collection information, the flags, the document table, and the dictionary.
The loading of this information can take a few minutes, after which the user can, via a textual interface, view and edit the various information and flags and enter queries.
The queries that the user can enter are textual and after they are given to the system, they will be processed in the same way as documents when the inverted index is created. Depending on the flags enabled, tokenization, stop word removal, stemming and lemmatization will be done.
After entering the query, the user can specify several parameters such as the number of documents to be returned (10 or 20), the choice between conjunctive and disjunctive queries, the scoring strategy to be used, and which scoring function to use (BM25 or MaxScore).
The basic strategies used are DAAT (Document-at-a-time) or Max Score, in the case of dynamic pruning algorithm flags activated.
The two scoring functions used are BM25 and TF-IDF.
BM25 is a widely used scoring function, which scores each document based on term frequency and a length normalization component.
The TF-IDF is a scoring function that assigns a score to each query term based on the term frequency and the inverse frequency of the document.
In both scoring functions, one or both of the following logarithms are present: log10(tf)), log10(NDoc/postListLen(t)). In particular, in TF-IDF both are present while in BM25 only the latter is present but also has a very long denominator with multiplication and division. These logarithms and denominators are precomputed offline or at the beginning of the DAAT or MAxScore to save execution time at the cost of increased memory usage.
The result of the logarithm for all possible values in the collection for the term frequency are saved in the file 'termFreqWeight' in the folder 'upperBound'.
The values for the denominator for each document are saved in the related DocumentElement in the documentTable.

3.1 DAAT

The DAAT traverses the posting lists of query terms in parallel, analyzing one document at a time in ascending order of DocID and calculating the score for it.
In the project implementation, there is a priority queue as large as the required number of documents as a result. There is a threshold value, which is initially 0 and will be equal to the score of the document in the last position of the priority queue. When the score of a document is calculated if it is greater than the threshold value it will be added to the priority queue and the threshold value will be updated. When all the documents in the posting lists are processed, those in the priority queue will be returned.
In the implementation of the project, the algorithm initially discriminates multi-term queries from single-term queries, which will be executed in a customized manner that increases performance (retrieval of the DIDs in ascending order and calculation of the score will be done simultaneously).
If neither compression nor skipping is active, the program retrieves all the DIDs in the posting lists of the terms in the query and then executes the DAAT as explained before.
If compression is enabled, the entire compressed posting lists will be read from the disk, decompressed, and then the normal DAAT will be executed.
In the case of skipping enabled and dynamic pruning disabled the normal DAAT will be exe-

cuted.
If both skipping and compression are enabled, the posting lists for the query terms are read and decompressed in blocks. Only the current block for each term is kept decompressed in memory.
If both skipping, compression and wholePLInMem are enabled the term posting lists are read and kept in memory whole and compressed. Then at runtime only the current blocks are decompressed and kept in memory.

3.2 Skipping

This procedure allows parts of the posting list to be skipped to speed up reading, which is particularly effective with MaxScore.
In the case of a compressed posting list, it is useful to not have to decompress the entire list but to compress only the block that is needed at the time.
This algorithm divides the posting list into fixed blocks and keeps for each block an index pointing to the beginning of the next block and the maximum DID present in the current block.
In this way, when searching for a specific DID, one quickly finds the block in which it is contained, the first block with a maximum DID greater than the one being searched for, and one can quickly skip blocks that are not useful without having to do decompression or other operations.
In the project, skipping information is stored in the file 'skipInfo'. SkipInfo' is the class that contains the information for a skipping block. Each block has a fixed size of 1024 postings, or less in the case of an end posting list. The information saved for each skipping block is the maximum DocID in the skipping block, the offset of the first docID in the next skipping block, and the offset of the first termFreq in the next skipping block.
The 'SkipList' class is used to manage all the blocks of a posting list (a term). Using this class, in particular the 'nextGEQ' method, the skipping of blocks to the right one and the search for the searched DID is implemented.
In general, this method given a searched DID and the current position in the posting list searches for the block in which the searched DID is present and then performs a binary search on it (searching between the current position and the end position of the block). To quickly skip blocks that are not useful, it looks at the maximum DID of the block, if it is less than the searched DID it will go to the next block otherwise that will be the searched block.
This method has two different versions in case compression is also active or not and in case it also has to handle block decompression. At the start of the MaxScore execution for each query term (i.e. for each posting list to be examined) the SkipList object will be initialized, which will then be called up during the DID search.

3.3 MAxScore

The MaxScore is a strategy for dynamic pruning that is used to speed up the process by skipping non-essential documents with scores that will certainly not be among the top documents.
During the processing of a query, the best 'k' scores calculated so far, together with the corresponding DIDs, are organized in a priority queue. The smallest score value of the documents in the priority queue will be the threshold value. This threshold value can only remain constant or increase in the course of query processing. A document with a score lower than the threshold will never be able to enter the priority queue due to the increasing sorting by score
Under these assumptions a document that has an upper bound document less than the threshold value can be skipped being sure that it will never be part of the best 'k' results.
The Max Score is based on dynamically dividing the query-related posting lists during execution

into two groups.

- Essential posting lists (EPL): consisting of documents that cannot be skipped;
- Non-essential posting lists (NEPL): composed of the documents that can be skipped during execution because the documents in them alone can never be among the best (upper bound sum less than the current threshold).

The posting lists of the terms in the query are sorted in ascending order by term upper bound and then the essential posting lists are processed via DAAT while the non-essential posting lists are only processed if there are documents that also appear in the essential posting lists.
In the project implementation:

1. The term upper bound is retrieved for each posting list (calculated offline);
2. The posting lists are sorted in ascending order by Term Upper Bound;
3. An object of type 'SkipList' is initialized for each posting list, which will have the task of managing skipping. These objects are organized in an array;
4. Initialised the ordered priority queue (heap) which will contain the best 'k' documents to be returned as results;
5. Set to 0 the threshold variable and an index which indicates the index of the first essential posting list (at the beginning it is 0 because all posting lists are EPL);
6. The normal DAAT will be executed and each time a document has a score such that it is added to the heap, the threshold value will be updated and a check will be made to see if any changes need to be made between the EPLs and the NEPLs (updating the index of the first EPL).

Documents in NEPLs are only processed if they are also present in EPLs and only as long as the Document Upper Bound is greater than the current threshold value, otherwise, they will be skipped.
In the project, depending on the enabled options (flags), the behavior of the algorithm may change.
In the case of skipping not being enabled, a Max Score without skipping will be made on the entire posting list, not split.
If skipping is enabled the algorithm described above is done and if compression is also enabled the decompression part of only the current block is added.
If both skipping, compression and wholePLInMem are enabled the term posting lists are read and kept in memory whole and compressed. Then at runtime only the current blocks are decompressed and kept in memory.

4 Result

4.1 Build inverted index

This section shows a comparison of the size and construction time of the inverted index depending on the options (flags) enabled.

FILE		NO STOP WORD REMOVAL		STOP WORD REMOVAL	
		No Compression	Compression	No Compression	Compression
DICTIONARY	Skipping	76,792 MB	87,762 MB	66,091 MB	75,533 MB
	No Skipping	60,336 MB	71,307 MB	51,929 MB	61,37 MB
DOCID		1,326 GB	447,717 MB	820,536 MB	286,331 MB
TERMFREQ		1,326 GB	61,857 MB	820,536 MB	35,297 MB
TERMSUPPERBOUND		10,97 MB	10,97 MB	9.442 MB	9,442 MB

Figure 7: Table for comparison of inverted index file sizes.

Table 7 shows a comparison of the size of the inverted index's data structures (files saved in memory once the construction is complete).
As can be seen from Table 7, the files that occupy the most space are those for saving the DocIDs and the term frequencies of the inverted index posting lists, which in the base case occupy 1.326 GB each. The removal of the stopwords reduces the space occupied by both the DocIDs and the term frequencies by the same amount, about one above third, and also reduces the size of the dictionary by about 15% (same reduction as the number of terms).
Compression leads to an increase in the size of the dictionary because it adds two fields to each term in the dictionary. These fields are both integer type and therefore lead to an increase of 8 B for each saved term.
Compression leads to a large gain in space for DocID and term frequency. This gain is different for the two types of data and can change depending on the collection, more precisely on the values that will be saved in the posting lists of the inverted index for a document collection.
In our case, there is a gain of 65-67% for DocIDs and 95% for term frequencies.
So with these techniques, at the expense of an increase in the dictionary of 15.197 MB, we manage to save 2393.3 MB (88%) in saving DocIDs and term frequencies.

PART	NO STOP WORD REMOVAL		STOP WORD REMOVAL	
	No Compression	Compression	No Compression	Compression
SPIMI	10 m 35 s	10 m 40 s	29 m 45 s	30 m 39 s
MERGING	5 m 02 s	5 m 27 s	3 m 45 s	3 m 56 s
TUB	2 m 15 s	2 m 17 s	1 m 47 s	1 m 53 s
TOT	18 m 03 s	18 m 45 s	35 m 29 s	36 m 39 s

Figure 8: Table for comparing the construction time of the inverted index. The table shows the main components of the construction with their times and the total construction time of the inverted index.

Table 8 shows a comparison of the construction time of the inverted index with or without compression and with or without removal of stopwords and stemming.
The build times were taken by running the algorithm on a laptop with these characteristics:

1. Windows 10 as its operating system;

2. RTX 3060 mobile (3840 CUDA cores, 6GB GDDR6, 120 Tensor Cores, 30 RT Cores) as its GPU;

3. i7-10870H (8 cores and 16 threads) as its CPU;

4. 16GB of DDR4 RAM.

With more powerful computers the times could be shorter, while with less powerful computers the times could be much longer. As seen from Table 8, the longest phase is the execution of the SPIMI algorithm. The execution of lemmatization and stopwords removal leads to an increase in the execution time of the SPIMI algorithm by a factor of three and causes a decrease in the merging time (fewer terms and therefore fewer blocks to be merged).
Compression affects the merging time by increasing it slightly.

The skipping option is considered in the tables only for the size of the dictionary and not for the size of the other files or the construction time because it causes a negligible overhead in terms of time and space compared to the other options of stopwords removal and compression.
For the dictionary size is taken into account because when skipping is enabled the dictionary has two extra fields for each term saved in it. These fields consist of a long type and an int type and this causes an increase in the space occupied by 12 Bytes for each term saved in the dictionary.
The skipping file when the option is enabled occupies 27.208 MB.

The table does not include other files that are always there and whose size does not change depending on the options (flags) enabled.
These files are:

- 'documentTable' with a size of 438,469 MB;
- 'flags' with a size of 28 B;
- 'collectionStatistics' with a size of 220 B (the size can vary depending on the number of term frequency values most present that you want to track);
- 'termFreqWeight' with a size of 936 B.

4.2 Query

In this section, the performance is shown as average query execution times.
To perform these tests, two collections of 200 queries each were used, called "msmarco-test2019-queries.tsv" and "msmarco-test2020-queries.tsv.gz", which can be found at the following link.
https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020.

All times shown in the tables relate to the average query execution times for one collection. Therefore, to obtain them, a test was made running all the queries in a given test collection by adding up all the individual execution times of each query and then dividing the result by the number of queries executed in the test.
This test for more statistical robustness was repeated 100 times and the average value of the 100 tests was calculated, this value is the one shown in the tables.
Each option's time is shown in this format: 'conjunctive_time' , 'disjunctive_time' (the times are in milliseconds).

	BASE CASE	SKIP (MAX SCORE)	SR	SR/SKIPP (MAX SCORE)	SR/COMP/ SKIPP	SR/COMP/ SKIPP/WPL	SR/COMP/ SKIPP (MAX SCORE)	SR/COMP/ SKIPP/WPL (MAX SCORE)
TFIDF	655 , 664	584 , 544	22 , 23	21 , 11	56 , 60	23 , 24	47 , 37	29 , 33
BM25	840 , 848	789 , 195	56 , 58	62 , 17	108 , 115	77 , 77	87 , 45	66 , 41

Figure 9: Table with results of tests performed on the 2019 collection. Legend for acronyms: SR indicates stopword removal, Skip indicates Skipping, Comp indicates Compression, WPL indicates all compressed posting lists held in memory.

Table 9 shows the tests, with different flags enabled, performed on the 2019 collection.

	BASE CASE	SKIP (MAX SCORE)	SR	SR/SKIPP (MAX SCORE)	SR/COMP/ SKIPP	SR/COMP/ SKIPP/WPL	SR/COMP/ SKIPP (MAX SCORE)	SR/COMP/ SKIPP/WPL (MAX SCORE)
TFIDF	558 , 561	524 , 486	25 , 25	24 , 12	63 , 63	26 , 26	56 , 42	31 , 38
BM25	757 , 748	724 , 176	62 , 65	70 , 19	118 , 123	83 , 84	98 , 52	73 , 43

Figure 10: Table with results of tests performed on the 2020 collection. Legend for acronyms: SR indicates stopword removal, Skip indicates Skipping, Comp indicates Compression, WPL indicates all compressed posting lists held in memory.

Table 10 shows the tests, with different flags enabled, performed on the 2020 collection.

It can be seen from the results that in general, TFIDF is faster than BM25, which is more computationally expensive.
It can also be seen that the operation that reduces execution time the most is stopword removal.
The best performance is achieved with stopwords removal and skipping when MaxScore (dynamic pruning technique) is activated, especially in the disjunctive case.
With both skipping and compression activated, performance deteriorates due to the overhead introduced by the complexity of the algorithm, especially for repeated input-output operations.
As assumed above in small collections, such as the one used, which can fit completely in memory, it is better to activate the option to keep all the posting lists related to the saved query compressed in memory.
With this measure, the times become very similar, only slightly higher, than in the case with only stopword removal.

In conclusion, considering the speed of query execution the best setting is stopword removal and skipping enabled with the use of dynamic pruning.
If you want to occupy less space in memory, especially on disk, and maintain very good performance the best setting is stopword removal, compression, skipping, and the whole posting list in memory all enabled.