UNIVERSITÀ DI PISA

Department of Information Engineering
MSc in Artificial Intelligence and Data Engineering

Inverted Index and Query Processing

Multimedia Information Retrieval and Computer Vision Project

Irene Catini
Ludovica Cocchella
Adelmo Brunelli

Academic Year 2022/2023

Contents

1 Introduction

1.1 Project description

This paper deals with the topic of information retrieval and describes the choices and steps we made to construct a search engine in which users translates her/his information needs into queries and obtains a list of documents that the system considers most relevant based on scores computed with different metrics. The project developed is composed by three principal parts:

1. The Index construction
2. The queries processing
3. The performance evaluation

The index is a composition of data structures designed to support search. It is composed by a **Vocabulary**, to store unique terms and their statistics, a **Document Index**, to store documents and their statistics, a **Collection Index** to store collection information like the number of documents and the average length, and the **Inverted index**, which provides the map between terms and documents in which appear.

We worked with a collection composed by 8.841.823 documents (total dimension 2.2GB) where each document is described by a DocNo and the text. this collection is available on this page: https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020.

In the queries execution part, we developed a program able to read, with a simple command line interface, the queries inserted by the user and to return a list of IDs assigned at each document.

The performance evaluation part will report all the based information useful to understand the goodness of our data structures. To compute them we used a list of predefined queries located in the same page of the collection.

1.2 Code organization

We choose to use the Java framework to do this project. Java is a popular choice for building inverted indexes because it is a high-level, object-oriented programming language that is well-suited for large-scale software development. All the developed code is available on GitHub (link: https://github.com/IreneCantini/MIRCV). We divided the code into four packages:

- Cli: it contains the executable main used by the user to interview the search engine
- Common: it contains all the java files used by both Indexing construction part and Query processing part like the data structures and the file management.
- Index: it contains an executable file to start the construction of the index.

- Query processing: it contains all the java files responsible for the execution of the queries like the algorithms to compute the score for each document and the executable file to start the performance test.

The collection containing the documents to index have to be present on the resource folder together with a stopword file (used in the pre-processing phase as we will explain in section 2.1.3) and a file containing queries to use during the performance phase. All the files crating during the indexing construction will be inserted in the same folder and they will be read by the query processing phase.

2 Indexing

This phase consists in the construction of the inverted index and all the other data structures. This part is divided in two main step which are the execution of the *Single-pass in-memory indexing*, also called *SPIMI*, and the index merging algorithm. At the end of this phase we will have five files which are the following:

- File containing the Vocabulary
- File containing the posting list of doc_id
- File containig the posting list of frequencies
- File containing the Document Index
- File containing the information needed to perform skipping

2.1 Text Preprocessing

Before starting to build the inverted index and process a query, it is necessary to perform text preprocessing. In our case we use the same text preprocessing for both documents and queries and the implementation is within the class:

it/unipi/dii/aide/mircv/common/text_preprocessing/TextPreprocesser.java.

2.1.1 Text Cleaning

Text preprocessing is carried out using regular expressions. In our case the following steps are performed:

1. Removal URLs
2. Reduction to lowercase
3. Removal tag HTML
4. Removal punctuation
5. Removal Unicode chars
6. Removal extra whitespaces with a single one

2.1.2 Tokenization

Tokenization is performed by splitting the text of each document on whitespace returning in this way an array of tokens. An example is shown below:

"this is an example of tokenization" → [this, is, an, example, of, tokenization]

2.1.3 Stopword removal and stemming

This step is not mandatory. Before the index construction begins, a flag is set indicating whether or not to proceed with stop word removal and stemming. So during the execution of text preprocessing a check of the flag will be done and if this is set then stopword removal and stemming will be done otherwise the text preprocessing ends. To perform stopword removal, we downloaded a file containing a list of English stopwords while regarding stemming we used the java class *org.tartarus.snowball.ext.PorterStemmer* that implements the widely used Porter Stemming algorithm.

2.2 Data Structures

All the data structures used in our project are located in the package *common/-data_structures* and they are used in all the phases of the project. They are used to maintain information that will be written into disk and also to contain the information read from that. They are (written in lexicografic order):

- CollectionInfo
- Dictionarylem
- DocumentIndexElem
- Flags
- Posting
- PostingList
- SkippingElem

They are analyzed in the following subsection.

2.2.1 CollectionInfo

It is used to maintain the collection statistics that will be used in the query processing phase. Fields of this class are:

Type	Name	Bytes	Description
Long	`docid_counter`	8	Number of documents in the collection
Long	`total_doc_len`	8	Sum of document length

2.2.2 DictionaryElem

It is used to maintain the term statistics across the whole collection. Fields of this class are:

Type	Name	Bytes	Description
String	term	20	Word to which the statistics below refer
Int	df	4	Document frequency: number of documents in which the term appears at least one
Int	cf	4	Coleection frequency: Number of times in which the term appears in the whole collection
Long	offset_docid	8	Pointer to the position of the docid inverted list in the inverted file
Int	docid_len	4	Number of byte occupied by the docid list
Long	offset_tf	8	Pointer to the position of the term frequency inverted list in the inverted file
Int	tf_len	4	Number of byte occupied by the term frequency list
Int	maxTf	4	Maximum term frequency of the term
Long	offset_skipInfo	8	Pointer to the position of the skipping information in the relative file
Int	skipInfo_len	4	Number of byte occupied by the skipping information
Double	idf	8	Inverse document frequencies
Double	maxTFIDF	8	Term upper bound for tfidf usefull for the dynamic pruning algorithm
Double	MaxBm25	8	Term upper bound for BM25 usefull for the dynamic pruning algorithm

The total amount of bytes that each elem occupy is 92 Byte.

2.2.3 DocumentIndexElem

It is used to maintain the information on the single document across the whole collection. Fields of this class are:

Type	Name	Bytes	Description
String	docNo	20	String which identifies the document
Long	docId	8	ID to identify each document in the collection
Long	length	8	Number of tokens contained in the document

2.2.4 Flags

It is used to retain the information with which the inverted index was constructed. Fields of this class are:

Type	Name	Bytes	Description	Saved
boolean	compression_flag	1	Flag for checking if the inverted index was built with compression enabled	Yes
boolean	filter_flag	1	Flag for checking if the inverted index was builtwith stopwords removal and stemming	Yes
boolean	maxScore_flag	1	Flag for checking if the max score algorithm is enabled	Yes
boolean	debug_flag	1	Flag for checking if the debug mode is enabled	Yes
boolean	scoreMode	1	Flag for checking which is the score algorithm to use (TFIDF or BM25)	No
boolean	queryMode	1	Flag for checking which is the type of query to execute (conjunctive or disjunctive)	No

Its important highlight that not all these flags are saved into the final file because some of them are used only on a single phase of the project. The flags that will be saved by the indexing phase are the one with "yes" in the last column.

2.2.5 Posting

It is used to maintain the relationship between the document Id and the term frequency. It will be used by the data structure *PostingList* (session 2.2.6) to create an ArrayList of posting to represent the real Posting list for each term. Fields of this class are:

Type	Name	Bytes	Description
String	term	20	Word at which the below posting list is associated
ArrayList¡Posting¿	pl	variable	Arrays of postings containing the docId of the document in which the word appears with its term frequency

2.2.6 PostingList

It is a class used to maintain in memory the posting list for each term. The most important fields of this class are:

Type	Name	Bytes	Description
Long	docId	8	Document identification assigned by the system
Int	tf	4	Term frequency: how many times the word is present in the document with the above ID

2.2.7 SkipppingElem

It is used to maintain the skipping block information. Fields of this class are:

Type	Name	Bytes	Description
Long	docId	8	Maximum docID of the block
Long	offset_docId	8	Offset where postings list of document Id starts in the inverted file
Int	block_docId_len	4	Number of bytes containing docIDs list
Long	offset_freq	8	Offset where postings list of term frequency starts in the inverted file
Int	block_freq_len	4	Number of bytes containing term frequency list

2.3 SPIMI

For the construction of the inverted index we use the SPIMI algorithm which turns out to be more scalable than other indexing algorithms.
During execution, one document at a time is read from the collection and then text preprocessed obtaining a list of tokens. Every time we read a document we immediately save its document info to disk. We assign to each document an incremental docid that is an integer which identify a document in our system.
We divided the construction of posting list and vocabulary in blocks, each one maintained in main memory during the construction process and saved them on disk when we reached the maximum amount of memory to use in order to free it and start the construction of a new block. All the blocks will be merged during the merging steps to get the final data structures. For the construction of the vocabulary and partial posting list we proceed as follows: we process one term at a time from the list of documents tokens and check whether it is already in the dictionary or not. If the term is not present it is added to the dictionary and a posting list is created with a posting containing the information of the current document, otherwise, the already existing posting list is retrieved. In this last case it could be possible that the current document has already been added to the posting list meaning that the current token was present also in the document part already analyzed so all the necessary information has already been added to the data structures. If it is the first time that we meet that term for the current document, it is necessary

to update the document and collection frequency values.
The procedure previously described is done for each document until the maximum allowed memory usage is reached. As soon as the maximum memory usage threshold is exceeded, the partial structures are written to disk and the memory is cleaned up. As soon as the memory usage values return below the pre-established threshold we start with the new partial structures and so on until we have read the entire collection.

2.4 Merging

This is one of the most important part of the indexing process. As explained in the previous section, we decided to use SPIMI as indexing algorithm and for this reason we built different blocks and saved them into disk in order to avoid to maintain all the information in main memory, each one composed by a complete independent inverted index. After the construction of these blocks we need to merge them into only one big index and save it into disk which will be used in the query execution part. The java file containing the source code is present at the path *Index\Merger* and it is based on the idea of the *multi-way merge*.
Since each temporary dictionary is sorted alphabetically, we insert all the first terms, saved in each block, within a priority queue along with the index indicating the block number to which it refers.
When all temporary dictionary files have been processed and the priority queue has been completed, a cycle is started until all queue items are processed. We first extract the first element of the priority queue and then we add the next term presents in the same file of the one just extracted.
We extract all the information of that term from the disk files (the dictionary information and the posting list information) then, if we are in the first iteration, we pass to the next one, otherwise we check whether the term extracted in this iteration is equal to the one extracted in the previous iteration. If they are equal we sum the information of the current term with the information of the previous term. It is important to notice that, since the documents are processed in increasing order of docId, in order to merge the posting list of two equal terms, we need only to concatenate them. If the two terms are different this means that the information, of the one extracted in the previous iteration, is completed and we can write them in the final files. Before going ahead in writing the file, we compute the skipping information if the final posting list is longer that a certain threshold.

2.5 Compression

In this type of project, compression, allows us to deal with the amount of data which composed the final inverted index. We decided to implement two algorithms one for the compression of Document Ids and one for the Term frequencies, both information are used to compose the posting list of each term. We decided to compress this information because otherwise the final files containing them would be of very large size causing the need of a lot of storage space and computation time to work with them.

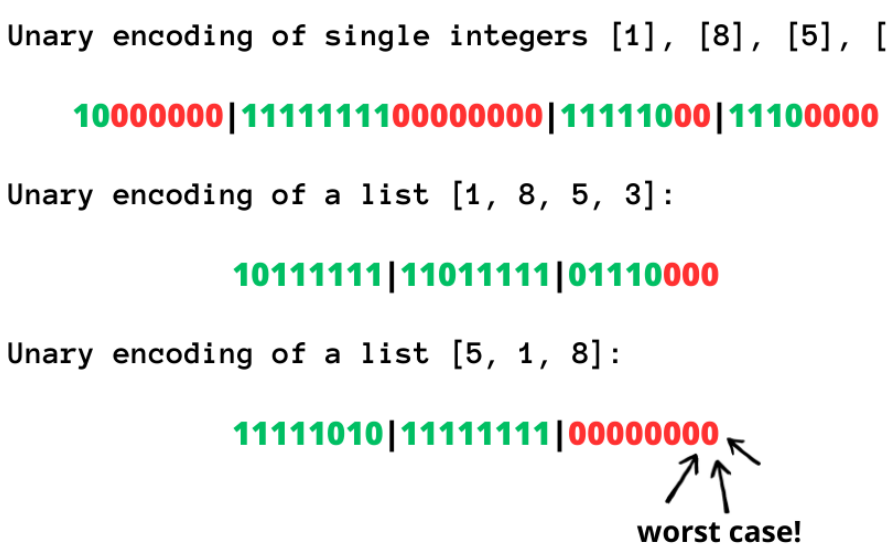
Indeed, compression is the process for which data is transformed into another representation that takes less storage space, saving space when storing data and time when we pass data from disk to main memory. The two used algorithms are:

- Unary
- Variable Byte

They are explained in the following sections and the source code can be found on path: *Common\Compression*.
In *Chapter 3* we show the performance obtained using or not compression.

2.5.1 Unary

Unary compression is used to compress the frequency posting lists. The programming language used for this project is Java, which reads and writes no less than a byte. For this reason the implementation has been changed in order to avoid encoding one integer at a time which would have led to a waste of memory. In fact, in case the encoding is smaller than a byte, it would be added as many 0s as those missing to reach the size of a byte. So what we are doing is encoding a list of integers by going to add an extra byte of zeros only in case the encoding from int to unary was perfectly a multiple of 8 bits. The addition of the zeros byte is necessary, for the purpose of decoding, since the latter does nothing more than go looking at each iteration for the next bit at 0. So, it allows to distinguish one integer from the other in the encoded list otherwise if we would have a list containing only bits set to 1 we would not be able to tell where one integer ends and the other begins. Below we show a series of examples to better understand what mentioned before:



From the first example we can see how many bits we waste (red bits) if one integer is encoded at a time compared to the second example where a list of integers is encoded. The last example instead shows the only case in which we have a waste of bits.

2.5.2 Variable Byte

Variable byte compression is used to compress the Document IDs written in the term posting lists. We decided to use this compression algorithm because the DocId value ranges between 1 to the number of documents present in the collection which, in the majority of the case, is very high. For this reason, DocIds may be a large integer and Variable byte is effective with these type of numbers. It's important highlight that each docId is represented by a long variable in the java code but, instead of occupy 8 bytes for its representation, each docId will occupy only the minimum number of bytes useful for its binary representation filled with 0 in case the most significant byte is not completely filled.

3 Query Processing

When an user executes the Main file of our program the first thing that the code do is to load in main memory some of the files constructed by the indexing part (section above). This files are: *CollectionInfo*, *Flag*, *DocuemntIndex* and *Dictionary*.
After that (it takes some minutes) the user can submit a query, the first step is to pre-process the query text using the same text pre-processing techniques that were used to construct the inverted index. This typically involves tasks such as *tokenization*, *stop word removal*, *stemming* or *lemmatization*, and possibly other techniques to prepare the query text for searching.
After the query has been pre-processed, the user can specify various parameters for the search which are: the number of documents to be returned, the choice between conjunctive and disjunctive query, which scoring strategy to use (**DAAT** or **MaxScore**), and which scoring function to use (**BM25** or **TF-IDF**).

The two most common scoring strategies are **DAAT** (*Document-at-a-time*) and **MaxScore**. DAAT traverses the posting lists in parallel and analyzes documents in increasing order of docID computing for each one the score. MaxScore is used to speed up the scoring procedure avoiding the score computation for some documents that will definitely not join the top final documents.

The two scoring functions used are **BM25** and **TF-IDF**. BM25 is a widely used ranking function based on probabilistic relevance computation that assigns a score to each document based on the term frequency and a length normalization component. TF-IDF is a weighting scheme that assigns a score to each term in the query based on the weighting term frequency and the Inverse document frequency.

3.1 Conjunctive Query

We implemented it in a function which takes an integer **K** as input, which represents the maximum number of documents to be returned. The function returns a priority queue that contains the Top K documents based on their scores.

The function starts by declaring two priority queues. The first one (`decPQueue`) is used to maintain the *docIDs* with the highest score in decreasing order, and the second one (`incPQueue`) is used to maintain the *docIDs* with the highest score in increasing order (it is used to easily retrieve the docid with the smallest score). The

function then orders the posting list in increasing order of length and loops until all the docIDs of the first posting list have been processed.

The loop fetches the first docid to analyze (it must be present in all the posting list of query terms) among those not yet processed. Then, the function calculates the score using BM25 or TFIDF formula, depending on the score mode specified by the user at the beginning of the execution. The docid and the score are then added to both priority queues only if it is above the current threshold.

3.2 DAAT

The implementation of DAAT algorithm is done in the **executeDAAT** method which takes an integer **K** as parameter. The latter indicates how many documents will be returned by saving them in a `PriorityQueue` containing the documents with highest score.
The algorithm starts by initializing two priority queues, `decPQueue` and `incPQueue`, both with a capacity of K. `decPQueue` stores the documents with the highest scores in decreasing order, and `incPQueue` stores the documents with the highest scores in increasing order. The algorithm then proceeds to iterate through the documents in the posting lists of the query terms following the DAAT algorithm.

For each document, the algorithm calculates its score based on the query terms using the BM25 or TF-IDF ranking functions (depending on the Flags configuration), and adds the document and its score to both priority queues. The process of inserting docs on queues follows this idea: if the size of `decPQueue` is smaller than k, the document is added to the queue without any other type of check on the score, if the size of `decPQueue` is equal to k, the algorithm checks if the document's score is greater than the smallest score extracted by `incPQueue`. If it is, the document and its score are added to both priority queues, and the document with the smallest score is removed from `decPQueue` and `incPQueue`.

The algorithm then moves on to the next document, and the process repeats until all the documents have been analyzed. Finally, the algorithm returns `decPQueue`, which contains the Top K documents with the highest scores.

3.3 MaxScore

The `executeMaxScore` function takes an integer **K** as input, which represents the number of top documents to return. The function initializes two priority queues, `decPQueue` and `incPQueue`, to maintain the Top K documents in decreasing and increasing order of scores, respectively. It also initializes an array list `ub` to store the sum of query terms upper bound in order to make a division between essential and not-essential posting lists.

After that division the algorithm continues analyzing only the essential posting list with the DAAT procedure. It takes the first docid to analyze, computes the partial score as the sum of essential posting list actual scores, sums it with the non-essential term upper bounds and checks if it is under or above the current threshold. In the first case it skips the computation and passes on the next docId, otherwise it goes on on the computation of the final score.

If the DocId has been processed completely it is added to the queue that will be returned at the end of the function.

3.4 Skipping

The skipping procedure is used in order to avoid decompressing and maintaining all the clear posting list in main memory. Thanks to this strategy we maintain only a subset of query term posting list and, every time we need to access to a new posting, we check if we need to download and decompress a new part of posting list or not. All the information to implement this strategy is saved during the indexing procedure in the *Sipping file*. To implement this functionality we created two main functions called:

- nextPosting
- nextGEQ

The `nextPosting` method is used to traverse a posting list and retrieve the next posting. The function begins by checking if the posting iterator has reached the end of posting list in the current block. If so, it proceeds to check if there are any more blocks left in the skipping element iterator. If there are no more blocks, the function sets the `actualPosting` to null and returns. Otherwise, the function reads a new block and updates the posting list iterator. Once the function has updated the posting list iterator, it retrieves the next posting in the posting list and sets the `actualPosting` variable to this value.

The `nextGEQ` method allows skipping to the next posting with a docID greater than or equal to the given parameter docID. This function uses the skipping information because if the current block's maximum docID is less than the given docID, the method switches directly to the next block without check each docIds containing in that block. Then it can skips also other blocks until the condition is not satisfied. Once the right block is identified, the method loads the posting list of that block and iterates through it until it reaches the first posting with docID greater than or equal to the given docID.

4 Performance

In the following section we are going to show the results obtained executing the code developed.

4.1 Indexing

In this section we show a comparison between the size of the index data structures and the indexing time construction with or without compression of the inverted index and with or without stopwords removal and stemming.

Table 1: File compression size

File	Compression		Non Compression	
	Filtered	Non Filtered	Filtered	Non Filtered
DocIds	731 MB	1.307 MB	1.550 MB	2.780 MB
Term Frequencies	58 MB	107 MB	777 MB	1.390 MB
Vocabulary	111 MB	129 MB	111 MB	129 MB

Table 2: Index construction duration

Compression		Non Compression	
Filtered	Non Filtered	Filtered	Non Filtered
34 min	23 min	39 min	33 min

4.2 Query Handling

4.2.1 Disjunctive Query

In this section we put a focus on the saved time using MaxScore instead of DAAT by comparing the average response time of the queries taken from msmarco-test2020-queries.tsv file by using TFIDF and BM25 scoring function in disjuctive query mode. In the table we show the result obtained:

Table 3: DAAT vs MaxScore

DAAT		Max Score	
TFIDF	BM25	TFIDF	BM25
19 ms	20 ms	16 ms	17 ms

4.2.2 Conjunctive Query

In this section we show the average response time of the queries taken from msmarco-test2020-queries.tsv file by using TFIDF and BM25 scoring function in conjunctive query mode. In the table we show the result obtained:

Table 4: Conjunctive results

TFIDF	BM25
15 ms	14 ms