Multimedia Information Retrieval and Computer Vision

“Inverted Index”

Acampora Vittoria

Laporta Daniele

Zingariello Francesco

A.Y. 2023-2024

Summary

1. Introduction

In this project, we are asked to create a search engine performing text retrieval. In fact, the most common way for people to search for information is thanks to a search engine. So, we create an inverted index structure based on a specific collection of documents, and a program that processes queries over such inverted index.

The GitHub repository of this project is: F-Zinga/MIRCV-project (github.com).

We used MSMARCO Passages collection available on this page:

https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020.

This is a collection of 8.8M documents, also called passages, about 2.2GB in size.

We also measure the performance in terms of memory usage and building time for the indexing phase and in terms of efficiency and effectiveness for the query processing one.

The architecture of an information retrieval (IR) system is made up of two main parts:

- The indexing component takes the collection of documents and implements the indexing and create the necessary data structures.
- The querying component receives queries from users, who are looking for specific information, and uses the index to find the most relevant documents. These documents are then ranked and returned in relevance order.

1.1 Main modules

We divide our project in 3 main modules, shown as follow:

- ***indexing***: build the data structure of the inverted index and write them to disk.
- ***query***: process the query given by the Cli and returns the list of the most relevant documents for that query. The user is also enabled to set the query parameters (scoring function, query type).
- ***evaluation***: load queries from a file, then generate query result end evaluate each of them. The results are written in the format required by the trec_eval tool to evaluate the performance of the search engine.

1.2 Main functions and classes

The major functions of our project are:

- ***MainIndexing:*** the main class of the indexing module. All its aspects will be discussed in chapter 2.
- ***ByteReader:*** the class used for reading encoded integers, using a buffer input stream and a specific compressor. The class ***TextReader*** instead is used to read text data, using a Scanner. The class ***RandomByteReader*** is used to read bytes from a random-access file, using a specific compressor.
- ***ByteWriter:*** the class used for writing encoded integers, using a buffer output stream. The class ***TextReader*** instead is used to write text data, using a bufferedWriter.
- ***Parser:*** the class for the parsing of the documents. All the uses are showed in chapter 2, during the preprocessing.
- ***Compressor:*** the class used for the compression and decompression of a list of integers. This class uses the Variable Byte Encoding method, as shown in the chapter 2, in the compression section.
- ***Merger:*** this class is used to facilitate the merging of the block files (generated during the SPIMI algorithm, in the indexing phase), in a single file.
- ***Term:*** a class to represent a generic term in the inverted index with all the information needed.
- ***MainQueries***: the main class for processing the queries. It interacts with the user to enter a query and execute queries with the inverted index.
- ***MaxScore***: the class contains the implementation of the scoring process based on DAAT algorithm. This is used both for conjunctive and disjunctive queries.
- ***PLI***: this class represents a collection of postings, to iterate through them.
- ***ScoreFunction***: class to compute the scores, both BM25 and TFIDF.
- ***MainEvaluation***: the main class for the query evaluation. The methods of this class are used to load queries from a file, generate a query results file and then evaluate them using the trec_eval tool.
- ***Parameters***: is an interface that contains all constants and paths.

2. Indexing

2.1 Pre-processing

For the creation of the index, in the information retrieval system, we can select two formats for the index: binary format or ASCII format (bytes or text). The Indexer component will use the *TarArchiveInputStream* class to decompress the collection.tar file, which contains the document collection. To handle text written in non-ASCII character sets, we adopt the Unicode standard of representation, and we select the UTF-8 encoding.

The uncompressed file contains one document per line in the following format: *<pid>\t<text>\n* where *<pid>* is the docno and *<text>*.

The class to pre-process each line is the *Parser*, which takes a line as input and returns a pre-processed string.

2.1.1 Text Cleaning

The tokenization is performed splitting the text using the white spaces and tokenizing the documents in the format: $[doc\_id]\backslash t[token1\ token2 \ldots\ldots tokenN]\backslash n$

To make the pre-process of the text, the Parser performs the following steps:

- Convert all the characters in the document body to lower case.
- Remove punctuation replacing them with a white space, to not merge 2 subsequent words; remove all the non-ASCII characters, replacing them with regular expressions “[^a-zA-Z0-9]”.
- Remove extra whitespaces at the beginning, end and in the middle of the text (during the tokenization).

The pre-processing of documents may produce some empty documents, so these documents are discarded. For this reason, we save the docNo to keep the mapping between the document and the document Id.

2.1.2 Stemming and stopwords removal

For the *stopwords removal* we downloaded a list of English stopwords so the tokens in this list (that are useless tokens) are removed from the array of tokens obtained during the tokenization phase.

For the *stemming* we use a library implementing the Porter Stemmer.

If the respective flag is true, we implement the process of removing stopwords and applying stemming procedure.

2.2 Indexing algorithms

The indexing is the process of creating the inverted index data structure for the collection. This process is composed of 2 separate phases:

- *Single-pass in-memory indexing* (*SPIMI*)
- *Merger algorithm*

2.2.1 SPIMI

The process of building the index for the information retrieval system is based on the Single-pass In-memory indexing (SPIMI) algorithm and is performed in the *createIndex()* function. We perform a check, before processing documents, to see if the amount of memory being used is greater than or equal to a certain percentage of the available memory. If this threshold is reached, the partial data structures are written to disk, using the *writeBlock()* function, and the memory is cleared by creating new objects for the data structures and calling the garbage collection to free up memory.

Every time we write the data structures in the disk, we indicate the block identifier in the name of the file.

We use files to store our data structures: one for the posting list's frequencies, one for the posting list's docIds, one for the partial lexicon sorted in alphabetical order and another one for the document index.

2.2.2 Merger

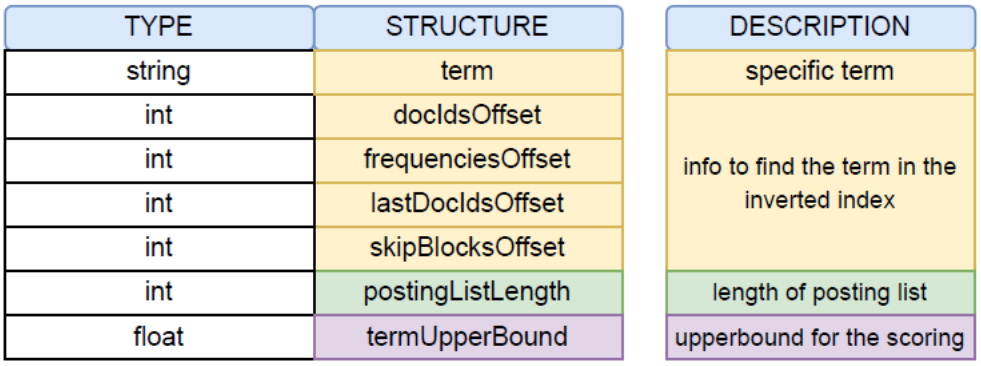
The Merger class is used to merge all the files containing the partial data structures obtained during the indexing phase (the docIDS, frequencies, lexicon, skipping and DocIndex blocks), in a single file.

During the SPIMI algorithm, a pointer to every block file is opened, using a buffer reader, and for every output file, a write buffer is opened. To perform the merging of the docIds and the frequencies files we need to select every time, the minimum term in the pointed lexicon blocks; this is possible because the docIds are ordered and separated on every block file.

To implement the merging phase, we decide the length of the blocks of the posting lists (500) and we save the skip pointers and the relative lastdocIds in two different files, to implement the skipping in the query processing. So, in the lexicon we save the offset of the docIds, the offset of the frequencies and we also save the offset of these two previous files. At the end of the merging phase, we obtain the final files: Lexicon, DocumentIndex, Skipping (with skip pointers and lastDocIds), Frequencies, DocIds and Statistics.

LEXICON

The lexicon (class *Lexicon*) is built with the HashMap constructor, between the string term and the class *Term*. Let's see the format of a generic lexicon entry:



TYPE	STRUCTURE	DESCRIPTION
string	term	specific term
int	docIdsOffset	info to find the term in the inverted index
int	frequenciesOffset	
int	lastDocIdsOffset	
int	skipBlocksOffset	
int	postingListLength	length of posting list
float	termUpperBound	upperbound for the scoring

DOCUMENT INDEX

The document index (class *DocIndex*) is built with the HashMap constructor, between the integer docId and the class *DocInfo*. Let's see the format of a generic docIndex entry:

TYPE	STRUCTURE	DESCRIPTION
integer	doc Id	document identifier
int	doc name	name of the document
int	doc len	len of the document

<u>POSTING AND INDEX BUILDER</u>

The class *Posting* is composed of two integers.

TYPE	STRUCTURE	DESCRIPTION
int	doc Id	document identifier
int	termFrequency	occurances of the term

The *IndexBuilder* class is built with the HashMap constructor, between the string term and the structure of an array list of Posting.
For each term we create a posting, if it does not already exist, and the document identifier is added to the term's posting list. After all the documents have been processed, the collection statistics are updated including the number of documents and the average length of a document. These statistics will be used by the ranking algorithm later.

2.2.3 Compression

We use a compression algorithm, in order to reduce the memory used by the inverted index data structure. This algorithm is the *<u>Variable Byte Encoding</u>* of the class *Compressor*, used to write to disk, in a compressed format, the docids and the frequencies (when the posting list of the term is merged). We also compress the docIndex, the skipPointers and the lastDocIds files. The compression algorithm will affect the size of the inverted index files (for each of the indexing configurations). We can also see speed improvement during the merge process when compression is enabled.

The indexing program has the second parameter that can be “text” or “bytes” to write files in ASCII format for debugging purposes or in binary format for a proper execution.

3. Query Processing

The query processing accepts a range of flags to specify how to operate. These flags include:

1. an Integer value K, which determines the number of relevant documents that should be returned as a result.
2. a String that specifies which scoring function to use (either "tfidf" or "bm25").
3. a String that specifies how the posting lists should be processed (either "daat" or "maxscore”).
4. a String that specifies how the queries should be processed (either "conjunctive" or "disjunctive").
5. a Boolean value (true or false) that specifies whether stopword removal and stemming are activated.
6. a String that specifies the type of the encoding (either “byte” or “text”).

Once the query processor is started, the system will accept queries from the user and return the *docNo* of the top *K* relevant documents based on the selected scoring function and their respective score.

3.1 Document Scoring

For the scoring of the documents, we can use two different functions: *BM25* and *TFIDF*.

3.1.1 MaxScore

To increase the speed of the scoring algorithm MaxScore, this is implemented as dynamic pruning algorithm. With this approach we avoid the need to score documents that will not be able to be included in the final top K results: because if a *docid* does not appear in at least one of the essential posting lists it surely cannot appear in the top k documents, so its scoring is skipped; but this is true only for the disjunctive case. In the conjunctive case, we only score documents that appear in all the term’s posting lists and so an essential list is always present.

The pruning strategy uses many components, including document upper bounds, term upper bounds, a minheap, thresholds, and essential/non-essential posting lists, to enable the system to quickly identify and eliminate irrelevant documents, allowing it to speed up the overall query processing process.

The scoring of the document is made with the DAAT algorithm: we sort in ascending order of the term upper bounds the document list. The lists are divided in essential and non-essential lists. The essential lists are the posting lists with the document such that the sum of their term upper bounds is higher than the threshold and so are the documents that can enter in the top *k priority queue*; all the other posting lists are non-essential.

3.1.2 NextGEQ

This method is defined in the class *PLI*, to ensure the iteration through the postings. If the docId that we are searching is not in the current block, we load another block into memory. So, the posting lists with *docIds* lower than the current *docId* are skipped, instead, the posting lists that have a *docId* higher than the desired one, are loaded into memory.

4. Performance

4.1 Indexing and Compression

We can index the collection:

- with or without compression (bytes or text parameter) and
- with or without stopwords and stemming removal (Boolean flag).

In addition, we specified with a string (bm25 or tfidf) the score type to use to compute the term upper bound.

In the following table we present the size of the document identifiers, the frequencies, the lexicon, and the document index all expressed in megabytes and the indexing time expressed in minutes. We use “S&S” instead of “Stemming & stopwords removal” for presentational purposes.

Indexing type	**DocIds (MB)**	**Frequencies (MB)**	**Lexicon (MB)**	**Document Index (MB)**	**Time (min)**
Text	2610	679	78.4	158	29.5
Only S&S	1402	339	66.1	158	16.6
Only compression	1240	339	80.3	71.9	19.2
Both	639	169	67.9	71.8	14.23

As we can see, the lexicon size is more or less the same in the case “Only S&S” (no compression) and in the case “Both” (Compressed and S&S). Instead for the DocIds files we see that in the “Both” cases, the size is around the half of the size of the case “Only S&S” and “Only compression”.

4.2 Query Evaluation

To evaluate the performance of the query processing, we download a file with 200 queries, called Test (2020), from the same website where we downloaded the collection.

This file is the qrel file (txt) and contains the following fields: qid, q0 (equal for all the results), docno, rating (the relevance of the document respect to a query).

When we execute the query, we append to the *query_results* file a line for each retrieved document. This line is composed as follow: $qid\ q0\ docid\ rank\ score\ runid$.

Then, with the *trec_eval* tool of the TREC community (Text REtrieval Conference (TREC) trec_eval (nist.gov)), we evaluate the results of the queries.

The computational time to process a single query goes from ~200ms, for the best case, to ~800ms for the worst one.

Inverted Index settings	**map**	**Recip_rank**	**p@5**	**p@10**	**iP@0.00**	**iP@0.10**
Compressed, S&S, disjunctive, BM25	0.1883	0.7389	0.5926	0.5426	0.8010	0.5768
Compressed, S&S, conjunctive, BM25	0.1448	0.7567	0.5696	0.4870	0.8074	0.5139
Compressed, disjunctive, BM25	0.1707	0.7985	0.6296	0.5444	0.8226	0.5616
Compressed, conjunctive, BM25	0.0610	0.8684	0.4105	02263	08684	0.2632
Compressed, S&S, disjunctive, TFIDF	0.1456	0.1636	0.5957	0.4978	0.8139	0.5058
Compressed, S&S, conjunctive, TFIDF	0.1360	0.6676	0.4963	0.4370	0.6946	0.4238

5. Conclusions

- To implement the compression, we use the variable-byte encoding algorithm, both for docIds and term frequencies. In this way we can achieve higher efficiency but to achieve a higher compression rate we should also implement another compression algorithm, for example the unary compression. The unary compression could be especially good for the compression of the term frequencies, since these frequencies are low for words in short passage.
- In general, for preprocessing and compression, it could be a good idea to test our collection with other algorithms to evaluate the differences in term of performance.
- We can also try to change the size of the blocks of the posting list (that we choose equal to 500) and study how this different parameter should cause changes in the time for the execution of the queries.