Multimedia Information Retrieval Systems

Search Engine Project

Cognata Fabio
Di Giacomo Luca
Paolini Giovanni

Summary

PROJECT DESCRIPTION

The project has been developed by:

- Fabio Cognata
- Giovanni Paolini
- Luca Di Giacomo

For the University of Pisa during the attendance of the course Multimedia Information Retrieval Systems during the academic year 2022/2023

The project aims to develop a search engine application capable of indexing a corpus of 8.841.823 documents and allow users to search through the collection.

The project has been developed using Java Programming Language along with Maven Project Manager.

The repository is mainly composed of a “components” folder, including all the parts to make both the index manager and the search engine work, and of a “utilities” folder where text normalization functions and compression functions are stored.

Other folders contain helper classes that implement some basic graphics and data classes to help with data conversion processes.

The document collection can be found at the following link:
https://msmarco.blob.core.windows.net/msmarcoranking/collection.tar.gz
The queries used for the tests are contained in the text file downloadable at the following link:
https://msmarco.blob.core.windows.net/msmarcoranking/queries.tar.gz

USAGE AND MAIN CLASSES

To use the search engine, download the jar files and place them in the same folder so that the IndexManager will create the file system that the Search Engine will exploit for queries. Input files must be placed in the data/input folder and query input files in the data/input/queries folder.

- **IndexManager**: Runs a console-based GUI that generates the necessary file system relative to the .jar file and initializes the index by saving it into .dat files inside the data/output folder by reading a formatted file in .tar; .tar.gz; .gz and .tsv formats.
  The input file must be composed of lines formatted as docno\tdocbody. IndexManager also allows the creation of a compressed version of the index and to enable/disable stopword filtering and stemming.
  To use the index manager it is only necessary to run the java -jar IndexManager.jar command and follow the steps in the menu options.

- **SearchEngine**: Allows to perform different types of queries in an interactive way by using /command/ to set-up the query modes, such as the scoring function to be used (TFIDF; BM25), whether or not to use the compressed version of the posting lists to reduce the quantity of data to be stored on memory and disabling/enabling the filtering of stopwords and stemming.
  To run the search engine, run the java -jar SearchEngine.jar command, then a guide will be printed with all the accepted commands, notice that every other string outside the commands will be used as query string and will produce a top 20 of the most fitting document identifiers with the relative assigned score.
  To print again the command list use the /help/ command. The conjunctive/disjunctive mode is recognizable by the respective initial lowercase letter in the squared parentheses immediately after Search.

CORE CLASSES

- **IndexBuilder**: It is a class that keeps track of all the information needed to handle chunks of the index, an IndexBuilder instance is able to load the chunk's information into memory and save them into a file automatically increasing the chunk size and resetting the structures for when the chunk's limit is reached.
  The core functionality is the addDocument function which takes information and terms from a document and updates the relative data structures in memory.
  The chunks produced are then merged by using a merge*sort-like algorithm over chunk pairs. Chunks are merged from the least recent to the most recent produced so that the posting list chunks containing the same terms can be concatenated easily. The average building time is 14 minutes for an unfiltered index and 10 minutes for the filtered one.
  In this branch of the project, the index is generated by separating the document IDs and the relative frequencies.

- **PostingList (Compressed)**: These classes hold the logic to implement the necessary search operations like next() which passes to the next document's information for the term and nextGEQ(d) for implementing skips.
  Uncompressed Posting List implements skippings by using binary search between the ending position and the current position, while the Compressed version uses skips stored as VB encoded byte offsets from the last skip, one every sqrt(#postings) for that list.
  Both classes allow the implementation of pruning by storing the upper bound for the term in the inverted index, greatly reducing the computation times. To initialize the data structures two methods are mainly used: *“***.from**” which generates the posting list from an Integer Buffer and *“***.openList***”* which reads the core information of a term from the lexicon and reads the file containing the document IDs taking them into a buffer, repeating the same logic for the frequencies file. In the search phase the compressed buffers are processed in parallel to not lose track of the docid-frequency pair.

- **VariableByteEncoder**: Holds the logic of the integers compression using Variable Byte. For implementing the stream of an integer the bit 0 indicates that the stream of the integer will continue on the next byte, 1 instead indicates the stream is over. In this class there are 3 core methods, the encode method is for class usage and encodes a single integer, while the encodeList and decodeInt methods are used together with binary buffers to perform encodings and decodings as-needed.

Index Builder Menu

- Change input file: Allows to change the input collection file.
- Build Index: Used to build the uncompressed Inverted Index with the selected input collection
- Compress Inverted Index: Used to build the compressed Inverted Index with the selected input collection
- Clean Output: Deletes all files and directories inside the output folder of the application
- Enable/Disable Filtering: Allows to swap between an index build which filters stopwords and includes stemming or a build which includes the stopwords and does not perform stemming on words.
- Exit: Terminate the execution of the application.

Search Engine Menu

- /help/ Print the menu with tooltips for each available command.
- /mode/ Swaps between conjunctive search and disjunctive search.
- /filter/ Disables or enables the use of stemming and the inclusion of stopwords in the inputted search query. Note that this change may take some time for the application to load again the following data structures: Filtered Inverted index, Filtered lexicon, Filtered doctable.
- /prune/ Disables or enables dynamic pruning (Using MaxScore) allowing for a faster search. Note that this mode puts the program in BM25 scoring mode and disjunctive search method automatically. Scoring mode and disjunctive search won't be able to be changed until the pruning mode is disabled.
- /compressed/ Enables or disables the usage of compression over the posting lists. Note that this change may take some time for the application to load again the following data structure: Compressed Inverted index, lexicon and doctable.
- /score/ Swaps between TFIDF scoring algorithm and BM25 scoring algorithm for the ranked search of the inputted query.
- /file/ Allows the usage of a prepared file, containing a list of queries (separated by new lines) to perform multiple queries at the same time. The output of this command will be stored in a separated output file containing the top K scores for each line of query.
- /exit/ Terminate the execution of the application.

FILE SIZES

The following table shows the file sizes for the different builds of the inverted index.

	Doc Table	Inverted Index	Lexicon
Filtered/Uncompressed	162 MB	773 MB (DocID) + 783 MB (Frequency) = 1.55 GB (Tot)	40 MB
Filtered/Compressed	162 MB	742 MB (DocID) + 197 MB (Frequency) = 0.93 GB (Tot)	46 MB
Unfiltered/Uncompressed	162 MB	1.4GB (DocID) + 1.4 GB (Frequency) = 2.8 GB (Tot)	48 MB
Unfiltered/Compressed	162 MB	1.3 GB (DocID) + 0.35 GB (Frequency) = 1.65 GB (Tot)	55 MB

LIMITATIONS

- The debugging functionality in the Index Manager is extremely slow and the functionality was not used in the latest steps of the project and left untested since it was substituted by the Tester class and for this reason, it may fail to generate debug files in particular scenarios, so use it carefully.

- The project is based on a static collection made of short documents and uses the assumption that vocabulary and document tables together with query terms's posting lists can fit in the main memory.

- Since the collection is static, updates and deletions are not implemented, simplifying the overall structure of the search engine and the data structures used in the offline phase.

- The system’s scalability is not completely maintained but could be easily implemented without changing the logic of most of the components in the project.

- Dynamic pruning has only been developed for the BM25 scoring function to simplify the file system.

PROCESSING TIME

INDEX MANAGER CORE TIMINGS

FILTERED [build + compression] times	UNFILTERED [build + compression] times
~ 626s + 268s	~ 686s + 392s

CONJUNCTIVE QUERY PROCESSING TIME

The timings are measured using the BM25 scoring function for which also pruning has been developed

QUERY	FILTERED [Compressed]	UNFILTERED [Compressed]
the hello world	4ms [6ms]	54ms [76ms]
the beauty and the beast	2ms [4ms]	86ms [101ms]
FILE: msmarco-test2019-queries (200 Queries)	802ms [1279ms] Single query: ~4ms [~6ms]	11544ms [12890ms] Single query: ~57ms [~64ms]

DISJUNCTIVE QUERY PROCESSING TIME

The timings are measured using the BM25 scoring function for which also pruning has been developed

QUERY	FILTERED [Compressed]	UNFILTERED [Compressed]
the hello world	83ms [86ms]	1322ms [1567ms]
the beauty and the beast	34ms [39ms]	1511ms [1783ms]
FILE: msmarco-test2019-queries (200 Queries)	15405ms [17216ms] Single query: ~77ms [~86ms]	239988ms [311617ms] Single query: ~1.2 Sec [~1.6 Sec]

DYNAMICALLY PRUNED

QUERY [BM25 Only]	FILTERED [Compressed]	UNFILTERED [Compressed]
the hello world	3ms [4ms]	52ms [60ms]
the beauty and the beast	1ms [2ms]	82ms [104ms]
FILE: msmarco-test2019-queries (200 Queries)	2506ms [2686ms] Single query: ~12ms [~13ms]	22875ms[31709ms] Single query: ~114ms [~158ms]