Department of Information Engineering - DII
MSc in Artificial Intelligence and Data Engineering
Academic Year 2022/2023

Multimedia Information Retrieval and Computer Vision

Project: Search Engine

Professor:
N. Tonellotto

Autor:
Chianese Andrea
Di Lucia Mirko

Contents

1 Introduction

1.1 Problem description

The project aim to develop a search engine able to index and process a minimum dataset of 8.8 million documents, in particular the dataset that can be found here under the ”Passage ranking dataset” section, first link. This project is related to the TREC evaluation for the MSMARCO dataset.

It was developed for the course ”Multimedia Information Retrieval” of the MsC Artifical Intelligence and Data Engineering at University of Pisa for the 2022/2023 year course.

This is the github repo link containing the final project:

https://github.com/mirkodilucia/search-engine

The project was developed, in pair programming.

1.2 Dataset description

The selected dataset is the Passage Ranking dataset found in the previous link. This dataset comprises approximately 8.8 million documents, totaling about 2.2GB in size. Each document have this format:

tsv:pid, passage

Here, ”pid” is a unique identifier for the document, and ”passage” is the document’s actual text. The document number and content are separated by a tab character, and each line ends with a newline character.

2 Project architecture

2.1 Global architecture overwiev

For testing and logic separation purposes the overall project was constructed based on the subsequent principal part and secondary modules, here the directory tree structure after, from the main github page, selecting **src** and after **main** :

```
main
│                           ├── cli
│                           │   ├── CommandParser.java
│                           │   ├── query
│                           │   │   ├── enums
│                           │   │   │   ├── Mode.java
│                           │   │   │   └── ScoreFunction.java
│                           │   │   ├── QueryHandler.java
│                           │   │   └── scorer
│                           │   │       ├── DAAT.java
│                           │   │       ├── MaxScore.java
│                           │   │       └── Scorer.java
│                           │   └── QueryParser.java
│                           ├── CommandLine.java
│                           ├── compress
│                           │   ├── UnaryCompressor.java
│                           │   └── VariableByteCompressor.java
│                           ├── config
│                           │   ├── ConfigLoader.java
│                           │   ├── CustomXStreamSecurity.java
│                           │   └── model
│                           │       ├── BlockDescriptorConfig.java
│                           │       ├── Config.java
│                           │       ├── InvertedIndexConfig.java
│                           │       ├── PartialResultsConfig.java
│                           │       ├── PreprocessConfig.java
│                           │       ├── ScorerConfig.java
│                           │       └── VocabularyConfig.java
│                           ├── document
│                           │   ├── DocumentIndexState.java
│                           │   ├── DocumentManager.java
│                           │   ├── preprocess
│                           │   │   ├── FinalDocument.java
│                           │   │   └── InitialDocument.java
│                           │   └── table
│                           │       ├── BaseDocumentIndexEntry.java
│                           │       ├── DocumentIndexEntry.java
│                           │       └── DocumentIndexTable.java
│                           │
│                                ├── FlagManager.java
│                                ├── indexer
│                                │   ├── Indexer.java
│                                │   ├── merger
│                                │   │   ├── MergerFileChannel.java
│                                │   │   ├── Merger.java
│                                │   │   ├── MergerPostingIteration.java
│                                │   │   └── MergerWorker.java
│                                │   ├── model
│                                │   │   ├── BaseBlockDescriptor.java
│                                │   │   ├── BaseMergerWorker.java
│                                │   │   ├── BlockDescriptor.java
│                                │   │   ├── Posting.java
│                                │   │   └── PostingList.java
│                                │   ├── spimi
│                                │   │   ├── BaseSpimi.java
│                                │   │   ├── Spimi.java
│                                │   │   └── SpimiListener.java
│                                │   └── vocabulary
│                                │       ├── BaseVocabulary.java
│                                │       ├── entry
│                                │       │   ├── BaseVocabularyEntry.java
│                                │       │   └── VocabularyEntry.java
│                                │       └── Vocabulary.java
│                                ├── Main.java
│                                ├── performance
│                                │   ├── CachePerformance.java
│                                │   └── Performances.java
│                                ├── preprocess
│                                │   ├── PreProcessor.java
│                                │   └── Stemmer.java
│                                └── utils
│                                    ├── FileChannelHandler.java
│                                    ├── FileHandler.java
│                                    ├── MemoryUtils.java
│                                    └── UnaryCompression.java
```

Instead in the **test** is possible to find all the JUnit test used for testing the main functionalities of the project.

In general the architecture can be summarized in the following core modules:

- **Baseline library:** This part refers to all the basic functions and modules used by the main principal core functionalities, containing functions such as the ones used for compression, for preprocessing or the basic structures like the vocabulary or the scoring strategies and functions or again the flags for example. All this functions are located near the main modules that uses them the most, for logic separation purpose. The directories in this category are: config, utils, preprocess , vocabulary, model, document, compress.

- **Indexer (Spimi and Merger):** This is the core of the search engine, contianing the Spimi and Merger class and also the correlated class and functions, part of this category are the directories: spimi, merger and also the baseline part related to the indexer.

- **CLI** Is the user interface, containing all the class and functions for handling queries from the user and also the functions for scoring like : CommandParser, query and QueryParser.

- **Performance:** This part contains the tests used for assessing performance of the overall system, for the queries and cache test to understand the impact of the use of the the caching. For the query performance test, the results obtained are written in a text file, formatted in a suitable manner to be submitted for the TREC evaluation (trec-eval), in particular CachePerformance and Performances.

3 Preprocessing and compression

3.1 Preprocessing

This is part of the baseline library and in particular the classes of this section are the PreProcessor and Stemmer. The overall process of preprocessing is carried out in three steps :

- **Text cleaning:** we performed the removal of non-Unicode chars, HTML tags, URLs, numbers and punctuations, unnecessary whitespaces (positioned for example as first character before the line, in the middle if major than 1 and at the end of the line) and cutting to only 2 characters all the sequences longer than 3 consecutive equal characters (for example "beginnning" becomes "beginning")
- **Tokenization and normalization:** as for the normal use of tokenization, we split the text line at whitespaces and camel cases and for normalize the test line we reduce to lower cases everything.
- **Stemming and stopword removal:** for this purpose we used the PorterStemmer and exploited a well known english stopwords list to remove not usefull parts (for example the mute vocals at the end of terms)

3.2 Compression

For the compression of docids and frequencies we decided to use:

- **Variable byte compression algorithm:** applied on docids.
- **Unary compression algorithm:** We decided to use this approach for frequencies because as we know, the integers representing them are not random and also the gap between the values is small then we can exploit this to represent the gaps instead of the numbers obtaining a smaller list. In particular we generate and store a byte sequence for each posting list, consisting of bit sequences that encode the frequencies. And this will also permits to store the sequence in a Byte array, if necessary applying padding.

4 Data structures

4.1 Posting list and block descriptors

The posting list and block descriptors are stored on the disk in the subsequent files:

- **data/indexes/inverted_index_docs :** for docIds
- **data/indexes/inverted_index_freqs :** for frequencies
- **data/block_descriptors :**for block descriptors, the skip-blocks contained in the block descriptor contains $\sqrt{n}$ posting lists with n postings, in particular this are the elements, the variables in the blockDescriptor class: **documentIdOffset** (long type), **documentIdSize** (int type), **frequenciesOffset** (long type), **frequenciesSize** (int type), **maxDocumentsId** (int type), **numPostings** (int type).

Where, for the blockdescriptor, the first four elements are for reading the blocks from the memory and the last two are for the skipping, used in the **nextGEQ** and for decompression purposes. The class BlockDescriptor extend the class BaseBlockDescriptor containing basic reading and writing file functions.

Also every file is formatted as ".dat" file for easier reading on the ide.

4.2 Vocabulary and vocabulary entry

The class Vocabulary is a wrapper that extends the BaseVocabulary class that implements the functionalities for extracting a vocabulary entry from file using binary search and also exploit a LRUCache structure for more efficient retrieving of entries already stored. Also we have the BaseVocabularyEntry that represent the vocabulary entry is composed by the subsequent elements:

- **term:** is the term saved in the vocabulary, 64 bytes.
- **vocabularyEntryUpperBoundInfo:** is a class element that contains the statistics of the term as: **maxTermFrequency**, **BM25Dl** (refers to the max document lenght for the BM25 formula), **BM25Tf**(for the max term frequency for the BM25 formula), **maxTfIdf** (max TFIDF score for the terms postings), **maxBm25** (same as maxTfIdf but for BM25).
- **vocabularyMemoryInfo:** is a class element containing **docIdOffset**, **docIdSize**, **frequencyOffset**, **frequencySize**, **numBlock**, **blockOffset**.

The BaseVocabularyEntry is extended by the VocabularyEntry that contains functions for operating with the vocabulary entry.

5 Indexing and scoring

5.1 Indexing

As we know the indexer is composed by the Spimi and Merger algorithm implemented by the the classes Spimi and Merger that extends the BaseSpimi and BaseMerger those contains the basic functionalities like channel opening functions etc..The Indexer class is the ”main” that calls the algorithms functions. The Spimi algorithm read the documents collection and the build the DocumentIndex storing it on disk. The DocumentIndex is composed by DocumentEntry, similar to the vocabulary reasoning, and each document is represented by a ”docid”. Here the partial inverted indexes produced by the Spimi algorithm is composed by a posting list docids file, a frequencies file and a partial vocabulary file. Instead the Merger algorithm, implements the merging of the partial inverted indexes, reading each term and concatenate the corresponding posting lists in increasing docid order, also use the same mechanism for the partial vocaulary and final vocabulary and finally stores the results on disk.

5.2 Scoring

The scoring process can be performed in different ways, in particular we implemented as scoring strategies the DAAT and MaxScore and as scoring functions the TFIDF and the BM25. If the TFIDF is selected we use as term-upper-bound the maxTfidf value instead for the BM25 we use the value maxBM25Dl and the maxBM25tf because it maximize the BM25 formula ratio (tf/(tf=dl)) and also its easier to compute. As said before this values are stored in the vocabulary entry for each term and calculated in the idnexing phase.

6 Performance and conclusions

Here we report the test performances results obtained by our project and also a final evaluation comparing our results with trec-eval.

6.1 Indexer

The tests on the indexer were performed to put in evidence the impact of the preprocessing and also the compression in terms of processing time, sum of the Spimi and Merger execution time and size of the resulting docids and frequencies files :

Compr/Prepr	**Time(min)**	**Docids(MB)**	**Freqs MB)**
False/False	50	1287	1287
True/True	41	638	30
True/False	46	1211	60
False/True	40	678	678

The differences in term of execution time are, as expected, significantly different when compression alone but also with preprocessing are involved, this is because entire integers are written all at once, whereas in compression scenarios, sequences of bytes are written instead and also is true that preprocessing maybe introduce some overhead but also reduce significantly the dimension of the files and this speed up the process.
Compression also significantly benefits the size of the final docids and frequencies file, in particular the last one due to the Unary Compression algorithm.

6.1.1 Queries execution time

To understand our system performances in terms of query response time, we used the "queries.dev.tsv" containing approximatly 100kk queries from the MS-MARCO dataset. First we decided to take into account the performance obtained comparing the score functions execution time TFIDF and BM25 with DAAT and MaxScore as scoring strategies.
Also we decided to test performances of MaxScore with BM25 and TFIDF against the trec-eval (trec.nist.gov/trec-eval/) and in the fourth and fifth column we reported the results in term of mean average precision and p@10 corresponds to the number of relevant results among the top 10 retrieved documents.

Score stragey	**mean**	**Std.dev**	**map**	**p@10**	**Score f**
MaxScore	22	23.62	0.0638	0.0078	BM25
DAAT	43	49.03			BM25
MaxScore	20	19.61	0.0675	0.0074	TFIDF
DAAT	38	40.12			TFIDF

6.1.2 Preprocessing and caching

Last we decided to test the impact of the implemented LRUCache of the vocabulary with the use of preprocessing that significantly impact query time response. The tests where obtained using MaxScore and TFIDF because they were the overall fastest ones.

Cache/Prepr	**Time(ms)**
False/False	175
True/True	19
True/False	170
False/True	22

6.1.3 Conclusions

As we can see from the last test the impact of the LRUCache implemented in the vocabulary is not so significant instead, the preprocessing highly affect the query response time, decreasing it by a lot so maybe using a different strategy for caching will make it more efficient, for example we could exploit adding a caching level for the query results because some terms are searched more frequently than others. Also we understand that our performances against the trec-eval are relatively low, for further investigate this problem we tried to test with and without preprocessing and, as we thought, preprocessing also affect negatively performances in terms of p@10 and map. So another improvement could be a less aggressive preprocessing or maybe more similar to the one used for trec-eval for example by the Terrier search-engine. Last implementing multithreading on the Spimi operations could increase the speed of the Indexer.