UNIVERSITY OF PISA

MULTIMEDIA INFORMATION RETRIEVAL AND COMPUTER VISION

PROJECT

Search Engine

Bergami Giovanni

Bruchi Caterina

Grillea Francesco

A.Y. 2023-2024

`github.com/francescogrillea/SearchEngine_MIRCVProject`

Contents

Chapter 1

Overview

1.1 Description

The project aims to realize a search engine in JAVA on the "MSMARCO Passages collection" of 8.8 million documents of size 2.5GB.

The project is developed in 2 separate phases:

- **Offline Phase**: Creates the index, composed of lexicon, inverted index and document index.
- **Online Phase**: Processes queries in 2 different modes (conjunctive and disjunctive) using 2 different scoring functions (BM25 and TFIDF) per user request.

At last we have an evaluation of the performances of the search engine on the TREC_EVAL framework.

1.2 Project Structure

The project is divided in two folders: src and data. In this chapter is described the first, in the next the latter.

1.2.1 src Folder

```
src
└─ main
   └─ java/org
      └─ common
         └─ encoding
```

```
|
|
|     |
|     |
|     |__ evaluation
|     |__ offline_phase
|     |__ online_phase
|           |__ query_processing
|           |__ scoring
|__ test
   |__ java/org
```

- **Main Folder**: contains the realization of the search engine, so all the classes for the offline and online phases and some utils classes.
  - **Common**:contains the classes for the implementation of the index, including the auxiliary structure for the handling of the index in memory. Inside there is also the **encoding folder** containing the classes for the implementation of the compression.
  - **Evaluation**: Contains the class that runs the tests on TREC_EVAL.
  - **Offline Phase**: Contains the classes for the preprocessing of the texts and the realization of the SPIMI indexing.
  - **Online Phase**: Contains the classes for the query processing phase, hence the scoring and ranking functions.
    - **Query Processing**: Contains the algorithm for DAAT (both conjunctive and disjunctive) and MaxScore.
    - **Scoring**: Contains the classes that implements the TFIDF and BM25 scoring
- **Test Folder**: contains the classes for the testing of the application.

1.2.2 data Folder

Here an explanation of the structure of the files present in the data folder:

```
data/
|__ analysis/
|   |__ memoryDump-MainClass.hprof
|   |__ memoryDump-MainClass[-p].hprof
|__ evaluation/
|   |__ trec_eval-9.0.7
|   |__ 2020qrels-pass.txt
|   |__ msmarco-test2020-queries.tsv.gz
|__ intermediate_postings/
    |__ doc_index/
```

```
|
|   |
|   |__ index/
|   |__ lexicon/
|__ lexicon.bin
|__ doc_index.bin
|__ index.bin
|__ cleanup.sh
|__ collection.tar.gz
|__ collection_subset.sh
|__ stop_words_english.txt
```
where

- `analysis` folder contains two memory dump of the query execution phase in order to prove that, even keeping the whole lexicon in main memory, the occupation of the whole main class is below 512MB (see Chapter 5.1).
- `evaluation` folder are stored all the components to produce the results given by the evaluation phase.
- `intermediate_postings` contains the whole intermediate files generated during the SPIMI algorithm.
- `lexicon.bin`, `doc_index.bin` and `index.bin` are the final data structures stored in a suitable format generated in the merge phase of the SPIMI algorithm. Are the files used by the query execution phase.
- `cleanup.sh` is an utility bash script that removes all data structures stored on disk (even the intermediate ones).
- `collection.tar.gz` the dataset.
- `collection_subset.sh` is an utility bash script that generates a subset of `n` elements of the dataset. Adding the flag `-r`, the documents will be picked randomly.
- `stop_words_english.txt` is the list of 851 english stopwords used in stopword removal process.

Chapter 2

Data Structures

In the following section we're going to take an overview of all data structures used in the project.

2.1 Posting List

The `PostingList` class represents a posting list for a given term in lexicon and is made up by the following variables..

Instance Variables

- `List<Integer> doc_ids`: a list of document IDs associated with the terms in the posting list.
- `List<Integer> term_frequencies`: a list of term frequencies, representing the number of occurrences of each term in the corresponding documents.
- `List<SkippingPointer> skipping_points`: a list of skipping pointers, which are used to optimize search operations by skipping over a certain number of entries.
- `int block_size`: indicating the number of elements between two skipping pointers. Is assigned in the `initPointers()` method to the squared root of the number of postings.

2.1.1 Skipping Pointer

The `SkippingPointer` class represent the information of a block used for efficient traversal of posting lists. Is made up by the following variables

Instance Variables

- `max_doc_id`: the maximum docID in the corresponding block.
- `block_length_docIDs`: block size (in bytes) of the corresponding docID list.
- `block_length_TFs`: block size (in bytes) of the corresponding term frequency list.

2.2 Lexicon

The `Lexicon` class implements a mapping between terms and the corresponding posting list's position in index file(s). Is made up by the following variables.

Instance Variables

- `HashMap<String, TermEntryList> lexicon`: the mapping between a term and its corresponding posting list information. Note that for a term we can have multiple TermEntry instances: this speeds up computation while merging the intermediate postings lists because we have the location of all posting lists of that term along all the intermediate index files.
- `int size = 0`: number of terms in the lexicon, this is useful in the `add` method in order to keep an association between term and posting list while creating intermediate index.

2.2.1 TermEntryList

The `TermEntryList` class represents a list of TermEntry object. However, in the final merged lexicon, the `TermEntryList` has only one `TermEntry` for each term.

Instance Variables

- `int term_index`: an index associated to the respective term.
- `List<TermEntry> termEntryList`: the list of `TermEntry` instances representing the location of posting list(s) across all the (intermediate and not) index files.

2.2.2 TermEntry

The `TermEntry` class represents metadata about terms in information retrieval systems, such as the position of the posting list of the relative term inside the index file. Is made up by the following instance variables, however besides to getters, setters and `serialize()`, no additional methods are required.

Instance Variables

- `int chunk_index`: the chunk index where the term is located. Useful in merging intermediate postings lists.
- `long offset`: the offset within the file where the term's posting list starts.
- `long length`: the length of the term's postings within the file.
- `int document_frequency`: the document frequency of the term in the collection, i.e., how many documents that term contains.
- `float tfidf_upper_bound`: the upper bound for TF-IDF scoring of the term.
- `float bm25_upper_bound`: the upper bound for BM25 scoring of the term.
- `int BYTES`: the number of bytes required to serialize a `TermEntry` object.

2.3 Document Index

The `DocIndex` class represents the document index structure for storing document information. Is made up by `HashMap<Integer, DocInfo> docIndex` which keeps the association between doc_id and its document information; however besides to add, getters, setters and `serialize()`, no additional methods are required. Is important to underline that this data structure is used only while generating intermediate index, in all other cases the document information are retrieved directly by file through the `DocIndexReader` class.

2.3.1 DocInfo

The `DocInfo` class represents information associated with a document. Is made up by the following instance variables, however besides to getters and `serialize()`, no additional methods are required.

- `int pid`: the original identifier of the document.
- `int length`: the length of the (processed) document.
- `int BYTES`: the number of bytes required to represent a serialized DocInfo object

2.3.2 Encoding

We chose variable byte encoding for doc_ids and unary encoding for term frequency to save space. The procedure are implemented in the classes with the same name.

Chapter 3

Offline Phase

Here we will describe the classes used to build the inverted index and the auxiliary structures:

3.1 SPIMI Algorithm with MultiThreading

The SPIMI algorithm is made up by two phases:

- **Intermediate Index Generation**: Generate intermediate indexes for each chunk of documents. The chunk size is setted hardcoded to 20.000 documents[1] and each chunk is processed by a thread in order to speed up and parallelize computation. Then each thread writes its own posting lists, lexicons and document indexes to disk. Multithreading fits this application very well, synchronization mechanism weren't even required.
- **Merge**: Merging all intermediate indexes generated in the previous phase. First merges intermediate lexicons together, producing for each term a list of TermEntry objects: this will speed up computation during the merge of posting lists, since for each term has already the position of its posting lists among all intermediate files. The document index merging is simply the concatenation of the intermediate ones.

3.2 Readers and Writers

For each of the above data structure, ad hoc classes have been created in order to write and read efficiently binary objects from disk.

[1]To avoid OutOfMemoryError during execution, the program will wait 1 second if the free memory amount is lower than 20%.

3.2.1 PostingListReader

The `PostingListReader` class provides utility methods for reading and writing `PostingList` objects to and from an index file specifying the encoding techniques to be used during compression process. Is important to notice that `read` methods take as argument a TermEntry in order to locate and read the posting list data directly from the index file; on the other hand `write` methods returns a TermEntry instance in which is contained all information of where that posting list has been stored.

3.2.2 LexiconReader

It just implements `writeLexicon` and `readLexicon` methods.

3.2.3 DocIndexReader

The `DocIndexReader` class provides utility methods for reading and writing `DocIndex` and `DocInfo` structures from/to disk. Besides the `writeDocIndex` and `readDocIndex` methods, we implemented

- `DocInfo readDocInfo(int doc_id)`: reads DocInfo for a specific document ID from the DocIndex file.
- `List<Integer> getPids(List<Integer> doc_ids)`: retrieves pids for a list of document IDs.
- `int readN(String doc_index_filename)`: reads the number of entries (N) in a DocIndex, but since the docIndex file is sorted by doc_id, and the doc_ids are incrementally, `N` is the total number of documents.

Chapter 4

Online Phase

This chapter will describe the classes needed to perform the query processing.

4.1 Scoring Classes

There are 2 classes for computing the score of documents, one for the BM25 and the other for TFIDF.

Depending on the user choice, the builder for the desired class will load in memory some collection statistics while initializing the query processing procedure, to avoid multiple read on disk during the ranking. In particular:

- BM25: loads all the document lengths and the total number of documents present in the collection
- TFIDF: only loads the total number of documents present in the collection

4.2 Algorithms

4.2.1 DAAT

There are 2 different classes implementing the DAAT, one for conjunctive queries and another for the disjunctive. Both of them in their builder load the lexicon from the disk to speed up the computation of the queries.

4.2.2 MaxScore

This class implements the dynamic pruning technique of MaxScore, its builder loads into memory the lexicon from the disk to speed up the computation of the queries.

4.2.3 Scoreboard

This class implements the min-heap used for the top-k ranking and for the update of the threshold for pruning.

This specific reader will be used only in the online phase to implement skipping and block decompression so it's been described here and not with the other readers

4.3 PostingListBlockReader

The `PostingListBlockReader` class provides functionality for block-wise reading of posting lists, where each block is accompanied by a `SkippingPointer`. The reader can iterate through the posting list blocks and retrieve information about document IDs and term frequencies for a given term. Each term has its own `PostingListBlockReader` instance. Is made up by the following instance variables and methods:

4.3.1 Methods

Here are the most significant methods of the class, to better explain its functioning:

- `boolean nextPosting()`: moves to the next posting within the current block or reads the next block if necessary. Returns true if a new posting is successfully moved to, false if the end of the posting list is reached.

- `Integer nextGEQ(int doc_id)`: retrieves the term frequency associated to the next document ID that is greater than or equal to the specified doc_id. Returns the term frequency associated to the doc_id document, or 0 if not found: note that `getTermFrequency` is used internally.

Chapter 5

Performance

All experiments and benchmarks detailed in this chapter were executed on the same machine (MSI Prestige 14 Evo A11M) in order to ensure a standardized and well-defined environment for our experiments. The machine used for benchmarks has the following specs:

- OS: Microsoft Windows 11 Home 64 bit Ver.2009(OS build 22000.675)
- CPU: 11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz
- Memory: 16 GB @ 2133 MHz, 8 $\times$ 2048 MB, LPDDR4-4267
- Graphics: Intel(R) Iris(R) Xe Graphics, 1024 MB
- Disk: SSD, SAMSUNG MZVL2512HCJQ-00B00, 476,94 GB

Also, all the results are expressed in terms of both mean values and standard deviations, which are derived from `5` independent executions of the experiments. Also, the `README.md` file specifies the commands to compile and execute the program.

Index Construction

The table below summarizes index construction phase, where the flags `-p` (pre-processing) and `-c` (compression) indicate respectively if stemming and stop-word removal must be applied, and if index must be compressed. Also, Index, Lexicon and DocIndex are the size (in MB) of the respective data structures written on disk.

flags	# Terms	Index (MB)	Lexicon (MB)	DocIndex (MB)	Time Elapsed
none	1.369.123	2690	54.8	101	00:15:09 ± 01:30
-c	1.369.123	1340	54.8	101	00:17:38 ± 02:22
-p	1.170.498	1430	46.3	101	00:06:30 ± 01:34
-p -c	1.170.498	738	46.3	101	00:07:35 ± 01:16

Table 5.1: Index construction performance

Query Execution

The following results are generated issuing the Search Engine on all the 200 queries provided in the `msmarco-test2020-queries.tsv.gz` benchmark collection and only the first `k=20` results are returned.

mode	score	MAP	P@20	NDCG	Time Elapsed (s)
DAAT Conj.	TFIDF	0.139	0.329	0.262	0.023 ± 0.020
DAAT Conj.	BM25	0.141	0.351	0.266	0.021 ± 0.017
DAAT Disj.	TFIDF	0.125	0.355	0.245	0.034 ± 0.030
DAAT Disj.	BM25	0.174	0.448	0.311	0.035 ± 0.030
MaxScore	TFIDF	0.132	0.367	0.257	0.028 ± 0.026
MaxScore	BM25	0.182	0.460	0.323	0.026 ± 0.024

Table 5.2: Index construction performance

After the submission of our project we noticed a bug in the computation of the Disjunctive DAAT and MaxScore scores for the documents in TREC_EVAL which resulted in the scores being slightly different in the pruned version and in the classic disjunctive version, the minor issue due to a wrong variable update has now been solved. Here are the result for the fixed version.

mode	score	MAP	P@20	NDCG	Time Elapsed (s)
DAAT Conj.	TFIDF	0.139	0.329	0.262	0.023 ± 0.020
DAAT Conj.	BM25	0.141	0.351	0.266	0.021 ± 0.017
DAAT Disj.	TFIDF	0.132	0.367	0.257	0.034 ± 0.030
DAAT Disj.	BM25	0.182	0.460	0.323	0.035 ± 0.030
MaxScore	TFIDF	0.132	0.367	0.257	0.028 ± 0.026
MaxScore	BM25	0.182	0.460	0.323	0.026 ± 0.024

Table 5.3: Fixed Index construction performance

5.1 In-Memory Structures

In order to speed up query execution, we decided to keep the whole lexicon in main memory. The below table summarizes the retained size[1] of the lexicon in main memory estimated by IntelliJ Profiler. However to have detailed information, consult the two memory snapshots saved in `data/analysis` folder.

flag	Lexicon (MB)	Total (MB)
none	452.14	487.57
-p	388.43	423.86

Table 5.4: Retained Size of Lexicon if loaded in memory

[1]The size of the object plus the size of all objects referenced only by the first object, recursively.