IR System - Index Implementation

Fabio Buchignani,
Franco Terranova,
Jacopo Cecchetti

Multimedia Information Retrieval and Computer Vision

Master's Degree in Artificial Intelligence and Data Engineering

University of Pisa

Contents

Chapter 1

IR System - Index Implementation

Information retrieval is the science concerned with the structure, analysis, organization, storage, searching, and retrieval of information.
Searching information is one of the most performed activities nowadays; 98% of all internet users use a conventional search engine at least once a month.
The most fundamental data structures used in IR are indexes, data structures designed for massive-scale search.
The scope of this project is the implementation of an index structure based on the document collection *Passage ranking dataset* available on this page: `https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020`.
This solution allows to handle information retrieval in front of a massive scale of documents, starting from the design of the data structures needed, implementing a scalable indexing and going towards query processing.
Java is the programming languages used for the implementation of this project.

1.1 IR System Architecture

The IR System's architecture is composed of the following elements:

- Indexing component, indexing and preparing the collection in a format suitable for a fast information retrieval.
- Search component, accepting queries from users representing their information need and performing the matching operation with the index structure.
  A set of documents will be retrieved and scored according to the usefulness, or probability of relevance, with respect to the query.

1.2 Index Overview

The main component of the index data structure is the inverted index, data structure that provides a mapping between terms and their locations in documents.
The inverted index uses the following auxiliary data structures:

- Lexicon, containing the term statistics across the whole collection
- Document index, containing information on documents and their statistics
- Collection statistics, providing general statistics about the whole collection

1.3 Document Collection

The document collection *Passage ranking dataset* available on this page: `https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020` has been taken in consideration for the implementation of our solution.
The collection contains 8.8M documents for about 2.2GB in size.
Each document is represented with a different line that respects the following format:

{docno}\t{content}\n

Chapter 2

Index Construction

The index construction can be performed specifying with a specific compile flag if the binary format or the ASCII format (used for debugging) should be used for the construction of the index.
The binary format is used by default, if the format is not specified.
The Indexer component will decompress the *collection.tar.gz* using the *GzipCompressorInputStream* and *TarArchiveInputStream* classes. In this way the document collection will be uncompressed during parsing. The *Unicode* standard of representation has been used, and in particular UTF-8 has been chosen as encoding scheme, in order to deal also with text that uses non-ASCII character sets.

2.1 Errors Handling and Text Processing

Empty pages are controlled calculating the length of each document content and checking that it should be different than 0.
Malformed characters are handled with the *BufferedReader* class that uses the internal exception *MalformedInputException*, and perform a replacement of the malformed character as default operation.
Malformed lines, with a format different than the one described in Chapter 1.3 are handled opportunely with the following regex [.+\t.+\n].
Each document will be normalized, turning it into lower case, and then tokenized using the regular expression ˆ[a-zA-Z0-9], replacing all punctuation and strange characters into spaces, and then splitting using the space character.
Tokens will be skipped if considered as stopwords, using a predefined list of english stop words.
The token will then be stemmed using the Snowball's english stemmer.
The posting will then be created if not present, and the document identifier will be added to the term's posting list.
Once all documents will be processed the collection statistics will be updated storing the number of documents and the average document length, subsequently used by the ranking algorithm.

2.2 Memory Management

The index construction has been implemented based on the Single-pass In-memory indexing (SPIMI) algorithm. This solution will allow us to scale our index construction storing intermediate results on disk and taking into account hardware constraints.
The amount of main memory available is obtained using *memoryMXBean*, the management interface for the memory system of the Java virtual machine, which provides information about the amount of heap memory available using the *getHeapMemoryUsage* method.
After processing each document we control if the amount of memory used is greater or equal than a percentage of memory available, MEMORY_FULL_THRESHOLD
_PERCENTAGE, and if this happen we write the partial data structures to the disk and clear them.
Until the memory will reach a second percentage value, MEMORY_ENOUGH_THRESHOLD
_PERCENTAGE, we are going to use the *System.gc()* method in order to suggest the Java Virtual Machine to expend effort toward recycling unused objects and make the memory they currently occupy available for quick reuse.
For our implementation, we used respectively 75% and 25% as threshold percentages.

2.3 Partial Index Writing

Each time the MEMORY_FULL_THRESHOLD_PERCENTAGE is hit, the partial data structures will be stored in the disk, specifying in the file name the block identifier.
The following file system structure will be used in order to store our partial data structures:

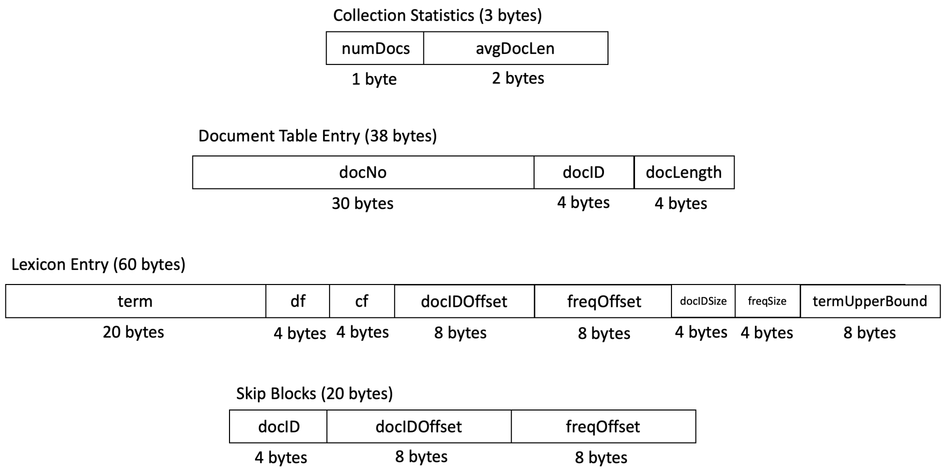
```
resources
├── document_table
│   ├── document_table0.dat
│   ├── ...
│   └── document_tableN.dat
├── inverted_index
│   ├── posting_doc_ids0.dat
│   ├── ...
│   ├── posting_doc_idsN.dat
│   ├── posting_frequencies0.dat
│   ├── ...
│   └── posting_frequenciesN.dat
└── lexicon
    ├── lexicon0.dat
    ├── ...
    └── lexiconN.dat
```

The folders needed for storing partial data structures will be created and deleted at run-time.

2.4 Integer Compression

A variable byte code compressing algorithm has been used to compress the integer values in our index.
Working with byte-aligned codewords can allow us to favor implementation simplicity and decoding speed.

Each integer is split in a suitable number bytes and for each byte 7 bits will be used to store data while the most significant bit will signal the continuation/end of the stream.



2.5 Partial Indexes Merging

After indexing all documents present in the collection, the files containing the partial data structures are merged and then subsequently deleted.
The final file system structure will be the following:

```
resources
 |__document_table.dat
 |__posting_doc_ids.dat
 |__posting_frequencies.dat
 |__collection_statistics.dat
```

The document tables are merged opening one input stream for each partial file, exploiting the ordering of the block index, and transferring the input channel to the output channel pointing to the final merged file.
Considering the merge operation of the posting document ID files, posting frequencies files, and lexicon files, a multi-way merge approach has been used, maintaining a priority queue storing the smallest termID at each step.
Each active block (blocks in which that term is contained) is merged, summing the document frequency and collection frequency of the reference term, and merging the encoded postings of docIDs and frequencies.
A final refinement of the index, after its construction, has been implemented in order to achieve the following goals:

- compute docID gaps and compress posting
- define the skip pointers (more information present in the Section 3.1)
- compute, for each term, its upper bound and collection frequency, used by the MaxScore algorithm.

.

Chapter 3

Query Processing

Our system can use the search component setting a specific flag at startup.
Once the query processor is started, the system will accept conjunctive and disjunctive queries with the following format:

Input Format: [AND—OR] term1 ... termN

Once the user will input a text line, this will be normalized, tokenized and the first token will be used to distinguish the type of query.
Stopwords removal and stemming will be performed on the remaining tokens.
Specific flags in the *application.properties* folder can be set in order to choose if stemming and stopwords removal should be used.
The docid of the top k documents according to the BM25 scoring function will be returned in output, using the following parameters:

- k = 10
- k1 = 1.2
- b = 0.75

The TFIDF scoring function has also been implemented and can be used to answer queries.
After returning the result, the program will wait for the next query to be input.

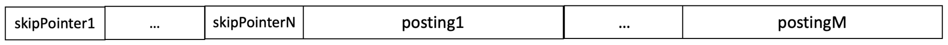
3.1 Posting list interface and skip pointers

The posting lists of the tokens will then be accessed using the following interface:

- openList()
- closeList()
- next()
- getDocId()

- getFreq()
- nextGEQ(d)

In our implementation, the constructor will coincide with the function openList().
Each posting list interface will maintain information about the document id and frequency offset in each corresponding file in order to load and decompress the posting list after each next() operation.
In case the document frequency of the term, which can be retrieved from the lexicon, is higher than SKIP_POINTERS_THRESHOLD (set to 1024 in our project), skip pointers will be used for the associated posting list.
The nextGEQ function will also exploit skipping pointers in order to avoid to load and decompress intermediate postings.



3.2 Supporting data structures

In order to avoid to load all data structures needed to process queries in memory, the following choices have been made.

- The collection statistics will be loaded in memory, since it will be very small.
- The lexicon will not be loaded entirely in memory. Binary search will be used in order to locate the term, exploiting the sorted order of this data structure.
- The document table will not be loaded entirely in memory. The sequence of document identifiers will be exploited to access the correct position of the file.

3.3 Search results caching and Dynamic pruning

A caching strategy based on search result caching has been used to speed up query processing.
This strategy allow us to store the final ranked list of documents for a query, avoiding us to bypass the scoring process and can be expecially beneficial for popular queries.
To speed up query processing, a dynamic pruning strategy using the MaxScore algorithm has been considered in order to avoid scoring documents that will never be able to enter in the final top K results.
Document upper bounds, term upper bounds, a min-heap, thresholds, and essential/non-essential posting lists are used in order to implement this mechanism.

Chapter 4

Java Class Diagram

In the following diagram, we highlight the main components of our project in term of classes and inner methods/fields.

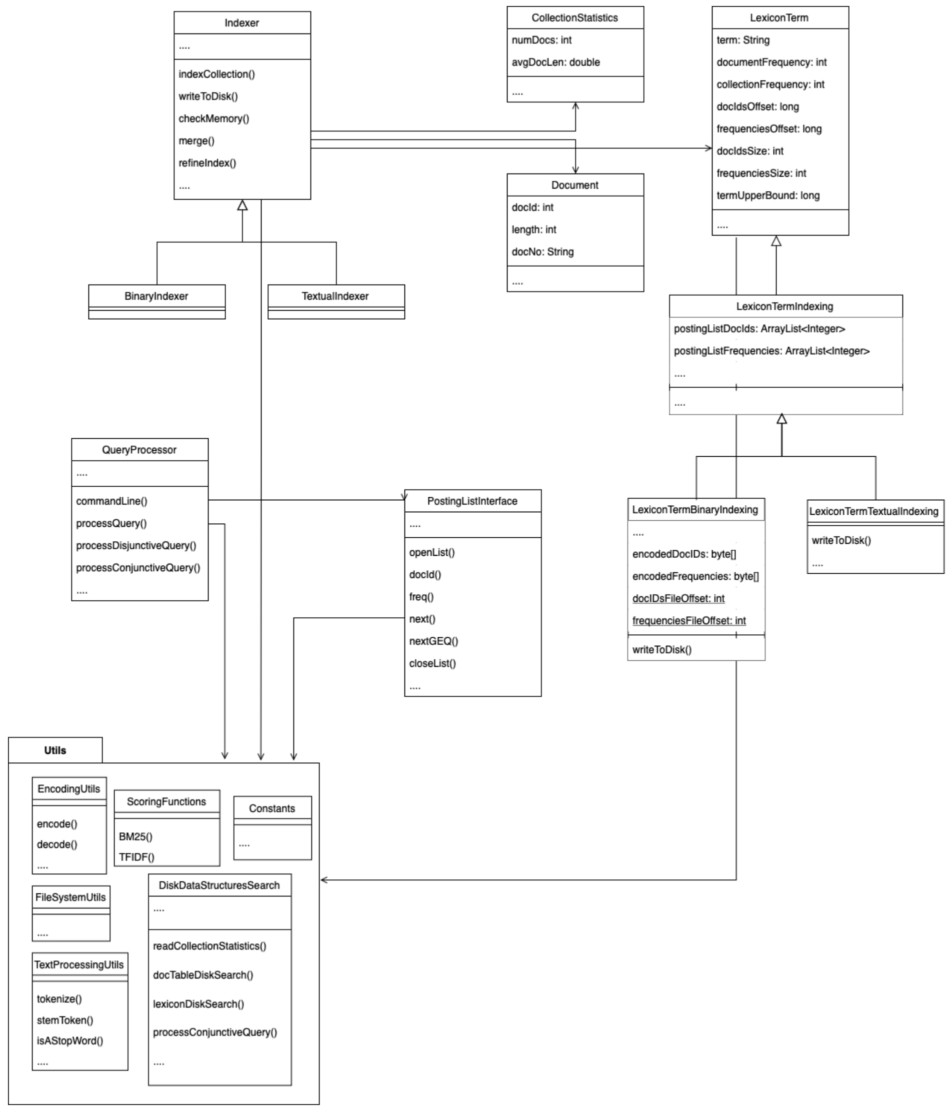


Figure 4.1: UML Class Diagram

Chapter 5

Evaluation - Effectiveness & Efficiency

5.1 trec_eval

trec_eval is the standard tool used by the TREC community for evaluating an ad hoc retrieval run, given the results file and a standard set of judged results. We used the files *queries.dev.small* and *qrels.dev.small*, available on this page: `https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020` in order to evaluate our solution against the Terrier search engine.
The results obtained are the following:

trec_eval metrics		
Metric	Terrier	Current solution
mAP	0.1929	0.1935
RR	0.1962	0.1974
nDCG@10	0.2290	0.2324
nDCG@100	0.2913	0.2857

5.2 Time statistics

In the following table we highlight the time needed to index and process query on the provided collection.

Time statistics	
Metric	Time Needed
Indexing	16 min 30 sec
Time needed for executing 6890 test queries	8 min 45 sec
Time needed for executing 6890 test queries without stopwords removal and stemming	30 min
Average time needed for executing each test query	0.08 sec

We decided to use the *Snowball stemmer* that is more effective but increased the indexing time w.r.t. other stemmers.
Consequently to a comparison in retrieving 1 million of documents, between FileChannel and MappedByteBuffer, we decided to use the latter.

Time needed to retrieve 1 million of documents	
Solution	Time Needed
FileChannel	2.746 seconds
MappedByteBuffer	0.151 seconds

5.3 Files' size

In the following table we highlight the size of the files resulting from the merging operation highlighted in the Section 2.6.

Files' size	
File	size
document table.dat	336MB
lexicon.dat	77MB
posting_doc_ids.dat	341MB
posting_frequencies.dat	230MB

Using gaps while storing posting document ids allowed us to reduce the size of the posting_doc_ids.dat file from 866MB to 341MB.
A test with low heap memory has been performed, modifying the heap memory to 16MB, so simulating the system in a low-memory environment.
28 minuts were required to answer the 6980 test queries.

5.4 Conclusions and limitations

The solution proposed can be an effective solution for an index structure based on the mentioned collection.
The following limitations have been highlighted:

- Working with byte-aligned codewords favor implementation simplicity and decoding speed, but sacrificing compressing effectiveness.
- A different compression algorithm than variable-byte, more suited for small numbers, could have been used for frequencies, such as the Unary compression.
- We selected the *Snowball stemmer*, available at `https://www.geeksforgeeks.org/snowball-stemmer-nlp/`, that works better but requires more time during the indexing phase.
- The stemmer used is not available on Maven and for this reason the stemmer was included in the resources folder and requires a different installation