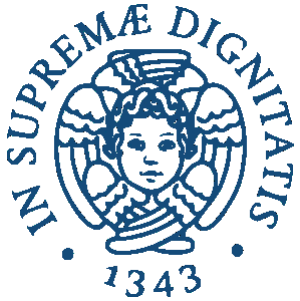

University of Pisa

School of Engineering

Master of Science in Artificial Intelligence and Data Engineering

Project documentation

YASE

MS-MARCO Search Engine

Work group:
Matteo Del Seppia
Federico Minniti
Tommaso Baldi

Academic year 2022-2023

Contents

1 Introduction **1**

2 Modules **2**
2.1 Indexing first phase: building partial indexes 2
2.1.1 Compressed reading 2
2.1.2 Tokenisation 3
2.1.3 Processing chunks 3
2.2 Indexing second phase: merging 3
2.2.1 Partial Index Merging 4
2.2.2 Index Compression 4
2.2.3 Computing the term upper bounds 4
2.2.4 Skipping postings 4
2.3 Query Execution 5
2.3.1 Posting List Iterator 5
2.3.2 Scoring Function 5
2.3.3 Types of Queries 6
2.4 Data Structures 6
2.4.1 Document Index 6
2.4.2 Lexicon 7
2.5 Command Line Interface 7

3 Performance **9**
3.1 Indexing 9
3.2 Query Processing 9
3.3 Effectiveness evaluation 9

4 Limitations **11**

1 Introduction

YASE is a simple search engine. It is capable of building an index and providing search results for the MSMarco document collection, available at **this GitHub page**.

The application is composed of two following modules:

- **Indexer**: creates the data structures, such as the inverted index, lexicon and document index;
- **Query processor**: supports free text queries, either *conjunctive* or *disjunctive*, and returns the search results to the user, together with a score and a rank.

YASE adopts simple compression techniques to reduce the size of the inverted index, in order to be efficient in terms of memory usage while maintaining fast search response times. The user can take advantage of the functionalities of YASE through a user-friendly command line interface (CLI).

In the following sections, the design and implementation of each of these components will be presented in details and then we will evaluate the performance of the search engine by using standard metrics from IR literature. In the final section we will briefly list the limitations of our search engine.

2 Modules

2.1 Indexing first phase: building partial indexes

During the first phase of the indexing process, the collection is processed in a single pass. Documents are read and decompressed one at a time, but they are processed in small batches that we called *chunks*. In the default configuration of the indexer, each chunk contains $N = 1000$ decompressed documents. Once a chunk has reached size N, its documents are processed in parallel and the current in-memory *partial index* is updated.

The in-memory partial index is composed of two structures: a partial in-memory inverted index and a partial in-memory document index. The in-memory partial inverted index is a HashMap containing *(term, posting list)* entries, while the in-memory document index is stored in a synchronized List to handle concurrent insertions by multiple threads. Whenever, during the indexing process, the partial document index and partial inverted index occupy more than 75% of heap space, the memory is freed by serializing them to disk in ad-hoc disk based data structures, designed to be used during the merge phase without being entirely reloaded in memory.

In particular, the in-memory partial inverted index is saved to disk by serializing the partial posting lists to an inverted file and storing their offsets in a disk based ordered array file with fixed-size entries handled by the class PartialLexicon. Each partial posting list in the partial inverted file is uncompressed and has the following format:

$$< \text{docid}_1, \text{tf}_1, \text{docid}_2, \text{tf}_2, ... >$$

The in-memory partial document index is also serialized to a disk based ordered array file with fixed-size entries, handled by the class DocumentIndex.

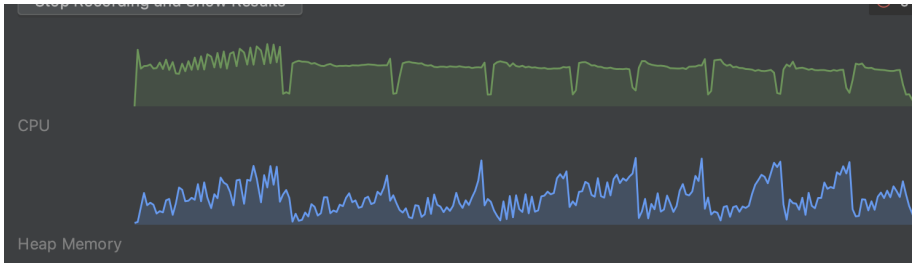

Figure 1: JVM profiling during the building of the partial indexes. Heap memory occupancy reaches 75% and then drops because partial structures are written on disk.

2.1.1 Compressed reading

The documents are decompressed one at a time as requested in the project requirements. A

$$\text{GZipCompressorInputStream} \rightarrow \text{TarArchiveInputStream} \rightarrow \text{BufferedReader}$$

pipeline is used to read and decompress only one document at a time. Each passage is added to a buffer containing the current chunk's documents. When the current chunk has reached size N, its documents are parsed and processed in parallel and the buffer is cleared.

2.1.2 Tokenisation

The tokenisation of each document follows this pipeline:

- non-ASCII characters and punctuation are replaced by space;
- everything is transformed to lower case;
- the passage is tokenised by splitting on whitespace;
- empty tokens are removed
- tokens longer than *maxTermLength* are dropped. *maxTermLength* is a configuration parameter, set by default to 20;
- stopwords are removed if the stopwords flag is enabled in the configuration file
- stemming with Snowball english stemmer is performed if the stemming flag is enabled in the configuration file

A document's length is set as the number of tokens (non-distinct) after the text preprocessing pipeline.

2.1.3 Processing chunks

Each chunk is processed in parallel by multiple threads, exploiting the functionalities of Java 8 streams. The processing steps for a single chunk of N documents are:

1. a parallel unordered stream of documents is opened, where each document is contained in a String object;
2. a *map* operation is used to process each document. First, the document is tokenized. Then a unique docid is assigned and the docid, docno and length are inserted into the partial document index. The total collection length is also updated. Finally, for each token a couple *(docid, token)* is emitted into the stream, maintaining duplicates;
3. the *(docid, token)* couples are sorted first by token's increasing lexicographical order and then by increasing docid. Consequently, the stream becomes ordered;
4. the *(docid, token)* couples are combined by a group operation, counting duplicate couples to get term frequencies. The stream is now composed of *((docid, term), tf)* elements;
5. each *((docid, term), tf)* entry is inserted in a LinkedHashMap for fast O(1) insertion and to maintain the insertion order. The stream is closed;
6. finally, the LinkedHashMap's content is used to create some partial posting lists containing only postings from this chunk, which are then merged with the posting lists in the current in-memory partial inverted index.

2.2 Indexing second phase: merging

The merging phase is the second phase of the indexing process. It takes as input the intermediate files produced in the previous phase, and outputs the final compressed index.

2.2.1 Partial Index Merging

The merging of the partial inverted indexes is done in the following way:

1. We prepare a **priority queue** which will keep the couples *<LexiconEntry, blockID>*, in order to have always at the head of the queue the lexicon entry of the first term in lexicographical order;
2. We open all the partial indexes' files in parallel, putting the first entry of each partial lexicon in the priority queue;
3. We process all the entries in alphabetic order, by merging the posting lists which refer to the same term end refilling the priority queue with the next entry of the last polled entry's block.
4. When we process all the entries related to the same term we write the posting list in the final inverted index and we update the final lexicon with all the required information such as offsets, upper bounds and document frequency;
5. At the end of the merging process, the global statistics of the collection are updated and stored to disk.

2.2.2 Index Compression

After the merging of partial posting lists associated to a term, the final posting list is compressed and stored on disk. In particular, docids (dgaps) and term frequencies are stored in two different files:

- *dgaps* are compressed using **Variable Byte** (VB), which is simple and has good decompression performance;
- *term frequencies* are compressed using **Unary**, which is an optimal code when the integers to compress are very small, much like our term frequencies.

2.2.3 Computing the term upper bounds

Once a certain posting list has been aggregated in the merging phase, we need to compute the upper bound scores associated to its term, needed by dynamic pruning algorithms like MaxScore and WAND. Hence, the posting list is entirely traversed, computing for each *(docid, tf)* couple its TF-IDF and BM25 score. The max TF-IDF and BM25 scores are then inserted into the lexicon entry associated to the term, ready to be used during query processing.

2.2.4 Skipping postings

Dynamic pruning techniques and conjunctive queries heavily rely on skipping operations, allowing a posting list iterator to position itself at a certain docid in the posting list without reading and decompressing all the postings in the middle. This is achieved by dividing the posting lists longer than a certain size M, in the default configuration set to 1024, in blocks of length $\sqrt{M}$, and saving information useful for skipping at the beginning of the posting list in the inverted file.

In particular, to implement skipping we need the maximum docid and length of each block of docids, and the length of each block of frequencies. Since our docids and frequencies are saved in a non-interleaved fashion on two different inverted files, skip blocks information is saved at the starting offset of the posting list's docids in

the docids' inverted file and at the starting offset of the posting list's frequencies in the frequencies' inverted file.

Our skip blocks format in the docids' inverted file for a posting list having n skip blocks is:

$$< \text{max\_docid}_1, \text{length}_1, ..., \text{max\_docid}_n, \text{length}_n >$$

where $length_i$ is the length in bytes of the i-th compressed block of docids.

The skip blocks format in the frequencies' inverted file for a posting list having n skip blocks is:

$$< \text{length}_1, ..., \text{length}_n >$$

where $length_j$ is the length in bytes of the j-th compressed block of frequencies.

2.3 Query Execution

Query execution is performed by classes in the **querying** package. The results of the query are presented to the user as a ranked list of documents sorted by relevance score.

2.3.1 Posting List Iterator

The query processor exploits the *PostingListIterator* class, which provides the following methods:

- *next()*: moves the iterator to the next posting, decompressing the next docid and the frequency and returning them in the pair *(docid, tf)*. If the iterator has reached the end of the posting list the method will return *null*.
- *nextGeq(int minDocid)*: advances the iterator forward to the next posting with a docid greater than or equal to *minDocid*. If the current docid is lower than *minDocid*, a seek to the start of the block having a max docid greater or equal than *minDocid* is performed, and then the iterator is moved forward using the next() until a docid that is greater than or equal to *minDocid* is reached.
- *getCurrentPosting()*: return the docid and frequency of the current posting.

NOTE: our Unary code is implemented through the class BitSet provided by Java, which internally reverses the order of bits (bits are written from LSB to MSB instead of MSB to LSB). Thus, in order to handle the decompression of the TFs in the posting list, we save in a BitSet the **compressed** representation of the TFs of the current block. The current TF is decompressed only on demand. We assert that this mechanism is relatively not memory expensive because during our tests with the MSMarco collection, the size of the current unary-compressed TFs block was in the worst cases (posting lists of "0", "1", "2", etc.) around 250 bytes. In the average cases it's much less.

2.3.2 Scoring Function

YASE implements both **BM25** (Best Matching 25) and **TF-IDF** (Term Frequency - Inverse Document Frequency) as scoring functions. The BM25 parameters are set to $k_1 = 1.2$ and $b = 0.75$, as suggested during the lectures.

2.3.3 Types of Queries

In YASE, queries can be processed in **conjunctive (AND)** or in **disjunctive (OR)** mode. A conjunctive query must return a list of documents in which all the terms of the query appears at least once, instead a disjunctive query must return all documents in which at least one query term appears. The max number of returned documents is $K = 1000$ by default, but can be changed in the configuration file.

In our system queries can be performed in three different modes:

- *and*: conjunctive mode, implemented with a DAAT variant where the posting list iterators are sorted by ascending order of posting list length and the current docid is taken each time from the current docid of the first (smallest) posting list, exploiting skipping to avoid decompressing intermediate postings of the other posting lists whenever possible;
- *or*: disjunctive mode, implemented with standard DAAT;
- *or+*: disjunctive mode, implemented with DAAT's MaxScore dynamic pruning optimization.

2.4 Data Structures

Other from the inverted index files, our auxiliary data structures are the Lexicon and the Document Index; they are implemented as external file data structures and memory mapped only during the indexing phase.

2.4.1 Document Index

The document index is an implementation of a disk based ordered array file with fixed size entries. Entries of the file are logically represented by the DocumentDescriptor class, containing a docid, a docno and the length. When the descriptor is saved to disk as an entry it has the following format:

```
<docno, length>
```

The docid is not stored as it is implicitly defined by the position of the entry in the file. The docno is a string that a fixed, configurable size. The DocumentIndex class has the following member functions:

- *DocumentIndex(...)*: constructor, prepares the data structures to write/read from disk
- *prepareToAnswerQuery()*: prepares the document index to answer query requests, seeking at the beginning of the associated file and initializing a BufferedInputStream to quickly traverse the file during DAAT;
- *getAtIndex(docid)*: get the i-th document descriptor stored in this document index
- *getAtGEQIndex(docid)*: returns the last descriptor read from the file or uses the BufferedInputStream to read the i-th document descriptor stored in this document index. This method is used during query processing as we know the file will be read in increasing order of docid;
- *putAllMMap(docid)*: stores a list of ordered DocumentDescriptor entries to disk, using memory-mapping for faster I/O;
- *closeFile()*: close the disk based array file.

2.4.2 Lexicon

The lexicon is a simple implementation of a disk-based open addressed hash table, using double hashing as collision resolution strategy. Each entry of the hash table either is empty or contains a serialized LexiconEntry object with the following format:

```
<term,documentFrequency,offsetDocid,offsetTermFrequency,
                                    upperBoundTFIDF,upperBoundBM25>
```

Where:

- *term*: the actual term (padded with spaces so all entries have a fixed size);
- *documentFrequency*: the length of the associated posting list;
- *offsetDocid*: the starting offset in the docids' inverted file of the associated posting list;
- *offsetTermFrequency*: the starting offset in the term frequencies inverted file of the associated posting list;
- *upperBoundTFIDF*: the term upper bound with respect to TF-IDF;
- *upperBoundBM25*: the term upper bound with respect to BM25.

The Lexicon has the following methods:

- *Lexicon(...)*: constructor, prepares the data structures to read/write from disk;
- *get(key)*: retrieve a lexicon entry stored in the lexicon through double hashing. If cache is enabled, checks whether the lexicon entry is already present in cache. If not, it retrieves the lexicon entry from the file on disk, adding it to cache;
- *putAllMMap(entries)*: this function is called during the merging phase to store the list of entries into the disk-based hashtable;
- *closeFile()*: close the disk-based hashtable.

2.5 Command Line Interface

YASE can be launch with three possible options:

- **-i** or –**index**: Create the index by processing the collection at the following path: "SearchEngine/src/main/resources/collection.tar.gz"

```
$ java -jar yase.jar -i
INFO  Indexer - Start indexing.
INFO  Indexer - Start processing the collection.
INFO  Indexer - write the block n.0
INFO  Indexer - write the block n.1
INFO  Indexer - write the block n.2
INFO  Indexer - write the block n.3
INFO  Indexer - write the block n.4
INFO  Indexer - write the block n.5
INFO  Indexer - write the block n.6
INFO  Indexer - write the block n.7
INFO  Indexer - Update collection statistics.
INFO  Indexer - Create document index.
```

```
INFO  Indexer - Create lexicon.
INFO  Indexer - Update collection statistics
INFO  Indexer - End indexing (5min).
```

- **-t** or –**test**: Creates a results file ready to be used to evaluate the effectiveness of the search engine with $trec_eval(Indexisneeded)$.

```
$ java -jar yase.jar test
INFO  Yase - Testing Queries -- writing results to 'results.txt'
INFO  Yase - Test completed, mrt: 44
```

-s or –**search**: Start using the search engine (Index is needed). At the beginning the user must set the scoring function which he wants to use, the kind of research that he wants to perform and the number of results to show.

```
INFO  Yase - Starting warm-up
INFO  Yase - 150 queries, total ms: 6457
INFO  Yase - Warm-up OK
 ______
|      |
| YASE |
|______|


Select scoring function: [tf-idf, bm25]
> bm25
Select type of queries: [and, or, or+]
> or+
Select number of results to print (NB: K=1000):
> 10
Write your query: [_exit to close and _reset to change settings]
> manhattan project
Showing 10 results out of 1000
rank docno score
1          2 29,503741
2    3615618 29,456713
3    2036644 29,287449
4    3870080 29,142097
5    2395250 28,985348
6    4404039 28,926638
7    3607205 28,772912
8    7243450 28,626078
9    3689999 27,701166
10    3870082 27,444392
54 ms (K=1000)
```

3 Performance

All tests have been done on a base Macbook Air M1 (2020).

3.1 Indexing

The total indexing time is around 5 minutes, using chunks of size $N = 1000$ (default). In particular, the first phase takes around 4 minutes, while the merging phase takes around 1 minute.

At the end of the indexing process our index files, built using the default configuration, will look like:

- **lexicon.yase**: 91MB
- **document_index.yase**: 106MB
- **index.docids.yase**: 324MB
- **index.tfs.yase**: 45MB

3.2 Query Processing

After indexing is completed, the query processing module is ready to be started. The query processing module always performs at the startup a brief warm-up of 150 queries (around 8 seconds), and after that a live CLI is showed to the user. During query processing none of our index files is loaded in memory or mapped in memory, as we wanted to maintain a certain control on the amount of memory the module uses to answer queries.

Note that the point at which the lexicon entries are retrieved, whether from the cache or directly from the lexicon, was considered as the starting point for measuring the execution time of each query.

Our performance in terms of query processing time was assessed by averaging the response time for each query in the *queries.dev.small.tsv* file provided by MSMarco, and returning for each query the top $K = 1000$ documents. More in detail, we found that:

- our mean response time with DAAT+MaxScore is around 42-45ms;
- our mean response time with DAAT is around 85-88ms;
- our mean response time for conjunctive queries is around 7-10ms;

3.3 Effectiveness evaluation

To evaluate the effectiveness of our search engine we used the **trec_eval** software, developed by the Text REtrieval Conference (TREC). The queries and *qrels* used to evaluate our system were contained respectively in the *queries.dev.small.tsv* and *qrels.dev.small.tsv* files provided by MSMarco.

The effectiveness measures we got from trec_eval were compared to the ones obtained by Terrier on the same queries. Both our system and Terrier use Snowball as english stemmer and the same list of stopwords.

Regarding the scoring function, both YASE and Terrier in these tests use their implementation of BM25 with $k_1 = 1.2$ and $b = 0.75$. Terrier's BM25 scores are slightly different from ours, as it employs an additional parameter $k_3 = 8$, expressing the importance of the presence of an additional occurrence of a term in the query.

```
$ ./trec_eval -m ndcg_cut -m map -m recip_rank -m recall.10,100,1000 -M1000 \
    \ qrels.dev.small.tsv results.txt

map                     all 0.1928
recip_rank              all 0.1961
recall_10               all 0.3861
recall_100              all 0.6792
recall_1000             all 0.8711
ndcg_cut_5              all 0.1975
ndcg_cut_10             all 0.2298
ndcg_cut_15             all 0.2444
ndcg_cut_20             all 0.2544
ndcg_cut_30             all 0.2663
ndcg_cut_100            all 0.2915
ndcg_cut_200            all 0.3023
ndcg_cut_500            all 0.3113
ndcg_cut_1000           all 0.3159
```

Metrics on dev-small				
Engine	mAP	RR	nDCG@10	nDCG@100
YASE	.1928	.1961	.2298	.2915
Terrier	.1929	.1962	.2299	.2913

4 Limitations

A first, clear limit of this search engine is that it does not consider the order of the query terms and their proximity within the documents.

Another possibility of improvement regards our current BM25 scoring function: we could improve it by taking in consideration the times a certain term appears in the query, using the k_3 parameter known from literature and already implemented by Terrier.

A further limitation of our query processing module is that it does not account for synonyms or related terms, treating all terms as independent. This may result in inaccurate scoring of document relevance when the search query does not exactly match the words in the documents.

The search engine's simple tokenization method has other several drawbacks, such as removing punctuation without considering its meaning, which might result in loss of information for hashtags, dates, abbreviations and prices. Additionally, it does not take into account multi-word expressions. These problems can be addressed through subword tokenization techniques.

Also, to achieve better compression performance, there would be a need of implementing additional classes handling the I/O of individual bits, which would simplify the implementation of advanced compression techniques such as Gamma/Delta for docids.