UNIVERSITÀ DI PISA

Multimedia Information Retrieval and Computer Vision

Project for A.Y. 2023/2024

Search Engine

Members:

Alice Petrillo

Daniele Giaquinta

Edoardo Focacci

Contents

1 General Information

The project assigned us the task of developing a search engine and presented us with a choice of programming languages: Python, C++, and Java. It also included both mandatory and optional tasks. As for our implementation:

- We choose **Java** for our search engine implementation.
- We implemented **all the optional tasks**.

The project consists of two functional modules:

- The **indexer**.
- The **query processor**.

Both of these modules make use of auxiliary functionalities, such as preprocessing (shared), compression (for the indexer), and decompression (for the query processor). Additionally, there are two other modules of secondary importance: one contains the test code used to evaluate correctness, and the other contains the code used to assess the performance of the search engine.

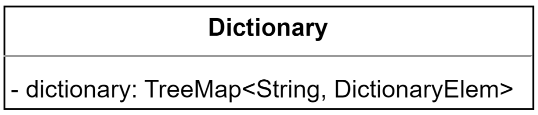
2 Data Structures

Firstly the main data structures created for this project will be discussed, as well as a short description of the auxiliary data structures; overall the data structures exploited are:

- **Dictionary**: contains key-values of term-information
- **DictionaryElem**: contains the term information
- **InvertedIndex**: contains key-values of term-postinglist
- **PostingList**: contains an array of postings
- **Posting**: contains a posting
- **SkipElem**: necessary data to implement skipping for a given term
- **Document**: stores the partial score during query processing
- DocIndexElem: stores pid and document length for a document
- TermBlock: stores a dictionary element of a partial dictionary and its block
- Comparators:
  - ComparatorTerm: alphabetically order terms during merger
  - DecComparatorDocument: order documents by decreasing scores
  - IncComparatorDocument: order documents by increasing scores

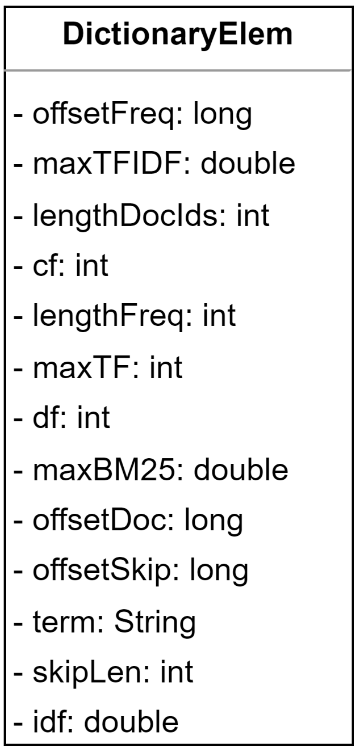
2.1 Dictionary

The dictionary class is quite minimal; it stores a mapping of terms to dictionary elements, but the dictionary elements themselves contain the term strings. This redundancy allows us to utilize the Dictionary data structure only during the indexing phase, without the need to increase complexity in later phases by relying solely on individual dictionary elements and their associated methods.



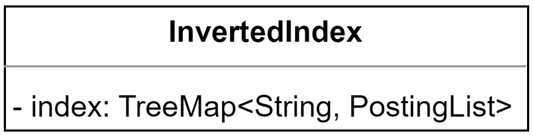
2.2 DictionaryElem

The dictionary elements contains the term itself (which we capped at 40 bytes), all necessary information for query processing (collection frequency, document frequency, term upper bounds for both TF-IDF and BM25, max term frequency, inverse document frequency), skipping information (offset of the skipping data for that term in the skipping file, number of skippers), posting list information (offset to the first docId in the docId file, offset of the first frequency in the frequency file; lengthDocIds and lengthFreq are the number of bytes of a block with compression or the number of elements without compression).



2.3 InvertedIndex

The same observations that apply to the Dictionary data structure also hold true here: it is a basic class that is only utilized during indexing.

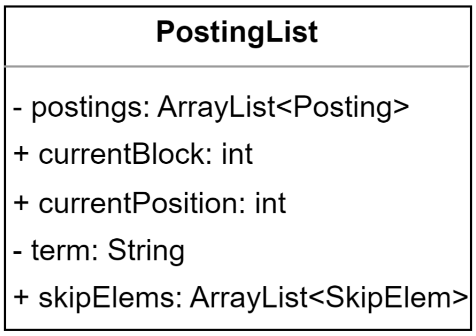


InvertedIndex

- index: TreeMap<String, PostingList>

2.4 PostingList

This data structure primarily consists of an array of postings and the term itself (to avoid using the inverted index class again). All other fields are necessary for implementing skipping. The 'currentBlock' field indicates which of the $\sqrt{n}$ blocks the posting list is in, while 'currentPosition' indicates the position of the posting within the current block ($\sqrt{n}$ postings per block). Additionally, the 'skipElems' array stores all the skipping information for the posting list. When a user inputs a query, we retrieve this list from the skipping file and store it in RAM until the query is processed.

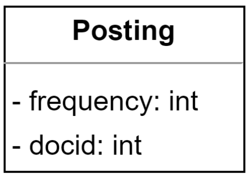


PostingList

- postings: ArrayList<Posting>
+ currentBlock: int
+ currentPosition: int
- term: String
+ skipElems: ArrayList<SkipElem>

2.5 Posting

This class is self-explainatory and does not contain much in term of fields.

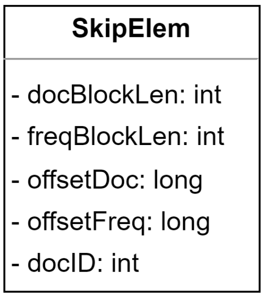


Posting

- frequency: int
- docid: int

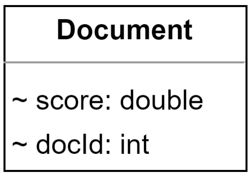
2.6 SkipElem

The data structure used for implementing skipping includes the following components: 'docId,' which actually represents the last docId in the block according to the standard; the lengths represent the number of bytes used for storing both docIds and frequencies in the block (which differ due to compression); and the offsets denote the starting position in the index file for these information (which differ due to compression). All of these details are stored in a file during the merging phase and are subsequently retrieved from the file and stored in RAM (the PostingList contains an array of SkipElem) during query processing whenever a term is part of the query.



2.7 Document

This class is used to store and update the partial score of a document during query processing. The fields of this data structure are fully utilized only in the context of the Document-at-a-Time (DAAT) query processing. With MaxScore and Conjunctive query processing, only the score calculation methods are utilized, as the partial score is treated as a local variable within the algorithms, since we adhered closely to the pseudo-code as much as possible.



3 Preprocessing

The same preprocessing is applied to each document during the SPIMI subphase and to each query after the submit. The transformations applied are:

- **Lowercase Conversion:** Converts all text to lowercase for uniformity.

- **URL Removal:** Removes URLs using regular expressions.
- **HTML Tag Removal:** Strips away HTML tags.
- **Punctuation Removal:** Eliminates punctuation marks by replacing them with spaces.
- **Non-ASCII Unicode Character Removal:** Deletes characters outside the standard ASCII range.
- **Whitespace Normalization:** Replaces multiple consecutive whitespace characters with a single space.
- **Tokenization:** Splits the text into individual words based on spaces.
- **Stopword Removal (Optional):** Filters out common words if the respective flag is set.
- **Stemming (Optional):** Applies stemming to reduce words to their root form if the respective flag is set.

Here is an example:

- **Before preprocessing:** *The Quick Brown Fox Jumps Over the Lazy Dog! Visit https://example.com for more details. <b>Important</b> dates: 23/04/2023.*
- **After preprocessing:** [*quick, brown, fox, jump, lazi, dog, visit, detail, date, 23, 04, 2023*]

4 Indexing

The indexing phase is divided into two subphases:

- **SPIMI** algorithm;
- **Merging** phase.

4.1 SPIMI Algorithm

During this phase, the code constructs a temporary inverted index divided into blocks. Operating under the assumption that a complete inverted index can't be constructed in RAM and subsequently stored in a file, a two-step process is required. SPIMI divides the three entities—the dictionary, the docId file, and the frequencies file—into blocks based on available memory. In this phase, **we employ a distinct method for storing partial dictionaries**, i.e. we do not store all the fields of DictionaryElem. This is because, during the SPIMI subphase, the system lacks all the necessary information to compute term upper bounds and other details, which will be calculated during the merging subphase. Therefore, we do not include padding data in the storage.

Note that the document index file is built during this phase by appending blocks, which is particularly easy since SPIMI processes one document at a time.

4.2 Merging

Merging is performed one term at a time. During this process, the dictionary blocks are incorporated into a priority queue. This arrangement allows us to extract the first term (which was alphabetically ordered during SPIMI) from each block at every iteration and process the term, even if it is present in more than one block.

Once we have the entire posting list of a term in RAM, we can calculate all the term statistics. Additionally, **this is the stage where we apply skipping support and compression techniques.**

This subphase leaves us with the following files:

- A single dictionary;
- A single docIds file;
- A single frequencies file;
- A file which contains the skipping information for the posting lists.

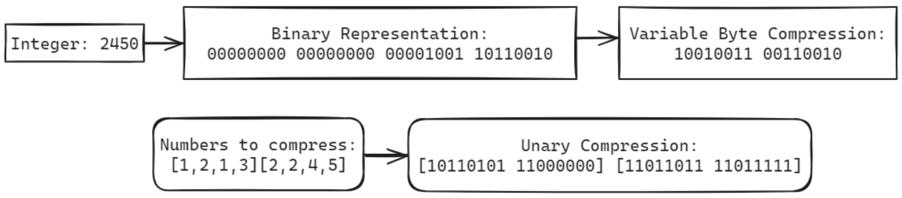
4.2.1 Compression

During the merging subphase, the inverted index is compressed using two different methods for docIds and frequencies, which is why we store them in two separate files.

- **Variable Byte** -> docIds
- **Unary** -> frequencies

It's important to note that compression was applied in 'blocks.' Here, 'blocks' refers to the segments into which a posting list is divided at this stage, and it is not related to the SPIMI blocks. These blocks are necessary because, after the merging phase, we cannot assume that an entire posting list can fit in RAM. The skipping mechanism is based on these blocks and will be discussed later.

While applying compression in blocks for the Variable Byte technique doesn't change anything, **for Unary compression of frequencies, the last byte of a block may contain more than one subsequent zero** in order to logically divide the blocks and avoid having a frequency split between one block and the subsequent one. Unlike the Variable Byte technique with docIds, where a docId cannot end up in the same byte with other docIds, in Unary, a frequency can end up in the same byte with other frequencies. Apart from this, we implemented the compression methods exactly as reported in the literature.


Integer: 2450
Binary Representation: 00000000 00000000 00001001 10110010
Variable Byte Compression: 10010011 00110010
Numbers to compress: [1,2,1,3][2,2,4,5]
Unary Compression: [10110101 11000000] [11011011 11011111]

In the examples provided above, Variable Byte compression is straightforward, while for Unary compression the reported case shows how two subsequent blocks are treated.

5 Files Overview

At the end of the indexing we find ourselves with the following files:

- A **Dictionary** file, which contains all the terms and their statistics;
- A **docIds** file, which may be compressed using VariableByte;
- A **frequencies** file, which may be compressed using Unary;
- A **docIndex** file, which contains PIDs and document lengths;
- A **skipping** file, which contains the skipElems for the posting lists;
- A **collectionInfo** file, which stores the collection length, the average document length and the flags which are set when the user starts the program with indexing (more about this in the readme file).

6 Query Processing

Query processing can be performed in both disjunctive and conjunctive modes. The disjunctive mode is implemented using both the standard Document-at-a-Time (DAAT) algorithm and dynamic pruning techniques. These optimizations leverage skipping and are implemented using the MaxScore algorithm.

The query string undergoes the same preprocessing as the documents, irrespective of the scoring algorithm or mode used. Additionally, users have the **option to choose between TF-IDF and BM25 scores**. For MaxScore, the computation of terms upper bounds for both metrics was needed, and it was conducted during the merging subphase.

Every algorithm and mode **returns only the top k documents**, ordered by scores. The value of k is an input parameter that the user can choose. To maintain the list of k documents to be returned and keep it updated, **we utilize two priority queues** within each document scoring method. One queue contains documents ordered by decreasing score (which is the queue used for returning the results), and the other is ordered by increasing scores (used for removing the least relevant document when adding new ones).

6.1 Binary Search and Skipping

Several binary searches are performed under the hood to enhance the efficiency of the document scoring methods:

- Binary search of the term in the dictionary
- Binary search of the correct block when skipping
- Binary search of the correct docId when skipping

The subdivision of posting lists into blocks is implemented such that the PostingList class has a constructor which takes a term string as input and just loads the first block of posting list after decompressing if necessary.

Skipping support is built during indexing where if a posting list contains more than 1024 elements, it is split into $\sqrt{n}$ blocks of $\sqrt{n}$ postings each, given n as the total number of postings in the posting list.

While in DAAT, skipping is not exploited since the algorithm performs an exhaustive computation over all the posting lists (and when a block is exhausted, the subsequent one is automatically loaded), in MaxScore and conjunctive query processing, skipping is part of the algorithms; the **NextGEQ** method is implemented inside the PostingList class, and thanks to the double binary search, the total complexity of a skip is: $O(\log(\sqrt{n})) + O(\log(\sqrt{n})) = O(\log(n))$.

When a posting list is initialized, i.e., its first block, all the skipping information is retrieved from the skipping file using the offset and length contained inside the DictionaryElem, and they are stored in a list as a field of the PostingList class. Every **SkipElem** contains the offset of the beginning of the block (for both docIds and frequencies), its lengths, and the last docId of the block, which is necessary for the docId search during skipping.

7 Performance and Document Relevance

In the indexing module, we measure the time required for indexing under various conditions and also compare the size of the final files. For query processing, we calculate the average query processing time and some metrics using a test set of queries. More details are provided below.

7.1 Indexing Performance

Indexing was conducted with and without compression, as well as with and without stemming and stopwords removal, in order to compare differences in execution time and the size of the final files.

	Compression && Cleaning	Compression && !Cleaning	!Compression && Cleaning	!Compression && !Cleaning
Time	16 minutes	15 minutes	18 minutes	19 minutes

Note that these execution time vary depending on the machine used for the indexing; we were able to get shorter times with one of our three laptops and longer times with another.

Now let's see what the size of the files are based on the situation:

	Compression && Cleaning	Compression && !Cleaning	!Compression && Cleaning	!Compression && !Cleaning
Dictionary	135,325KB	157,296KB	135,325KB	157,296KB
DocIds	653,756KB	1,307,682KB	656,113KB	1,328,353KB
Frequencies	73,555KB	150,089KB	656,113KB	1,328,353KB
SkippingFile	29,011KB	43,679KB	29,011KB	43,679KB

7.2 Document Relevance and Query Processing Performance

When discussing queries, we measured the average processing time for each query processing method using a test set of queries from the '*msmarco-test2020-queries.tsv*' file, which was published on the same website as the test collection.

To evaluate our results, we used the same queries and the provided expected relevant documents from the same website. We downloaded the 'trec_eval' program and compared our results with the expected ones to compute NDCG, precision, and MAP.

	DAAT	MaxScore	Conjunctive
Avg.Time	27ms	14ms	7ms

Here are the evaluation results produced by the 'trec_eval' program using $k = 10$ as the number of output documents:

NDCG	P@5	P@10	MAP	Reciprocal Rank
0.2561	0.5778	0.5426	0.1369	0.7614

8 Areas for Improvements

Let's start with our observations above: we achieved a fairly decent NDCG score, a good reciprocal rank score (indicating that the first relevant document appears on position 1.31 on average, though it doesn't account for how much exactly such document is relevant), and a decent Precision score. The MAP score, while not terrible, is perhaps the least favorable among these metrics.

We believe that improving our search engine's effectiveness in retrieving more relevant documents can be achieved by refining our preprocessing phase. There may be room for improvement in terms of stopwords lists, better stemmers, and other operations we may not have considered. A better preprocessing phase could result in a different dictionary that potentially enhances document retrieval efficacy.

Another area for potential enhancement is query processing times. For instance, we could introduce a caching mechanism that stores commonly queried terms and their offsets in the dictionary. This would not only eliminate the need for binary searches when searching for cached terms but could also speed up other binary searches. For instance, if we've cached the words 'apple' and 'coconut,' and we're searching for 'banana', we would know that we only need to perform a binary search between the offsets of 'apple' and 'coconut.'

However, we decided against implementing such a caching mechanism because we found our query processing times to be already quite low, without the need for additional optimizations.