UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Artificial Intelligence and Data Engineering

Search Engine

Project Documentation

Martina Marino

Roberta Matrella

Academic Year: 2022/2023

Contents

1 Introduction

The primary objective of this project is to implement a system capable of effectively handling a substantial corpus of textual documents. The Java-based system aims to create an inverted index, enabling users to formulate queries and swiftly access pertinent results.

1.1 Dataset used

The project draws upon a document collection available on this page, serving as the primary reference dataset. This collection comprises more than 8.8 million documents including diverse linguistic and formatting characteristics. Processing this dataset, a fundamental aspect is the resolution of some challenges, including the handling of non-ASCII characters and addressing formatting anomalies in raw data.

1.2 Code organization

```
main
├── java
│   └── it.unipi.dii.aide.mircv
│       ├── compression
│       ├── data structures
│       ├── index
│       ├── query
│       ├── test
│       ├── utils
│       └── Main.java
└── resources
```

The code organization follows a hierarchical structure adhering to a design convention that separates various components into packages and sub-packages. Specifically, the application is organized as follows:

- **compression**: contains code related to data compression (Unary and Variable Bytes).
- **data structures**: consist of definitions and implementations of custom data structures used for constructing and managing the inverted index.
- **index**: holds files relevant to the two-phase process of building the inverted index.
- **query**: contains code that handles query processing and search operations within the index, along with additional data structures.
- **test**: this package houses test code used to verify the functionality of various application components and assess performance.

- **utils**: offers utility classes and common functions for use throughout the application, including the `TextProcessor.java` file.
- **Main.java**: main entry point for the application execution.
- **resources**: contains various resources used by the application, such as the collection file, the file for query testing, and all files generated during the index building phase.

This hierarchical code structure eases project management and maintenance as it logically organizes the code and segregates different functionalities into separate packages. Inside the `rescources` directory must be present the `collection.tar.gz` file together with the `stopwords.txt` used for text preprocessing if the relative flag is enabled. During the test phase must be also present the test file (the `msmarco-test2020-queries.tsv` has been used for this purpose).

1.3 Fundamental data structures

- `CollectionStatistics`: holds the statistics of the entire collection, such as the total number of documents and the total document length.
- `Dictionary`: contains a HashMap for <term, DictionaryElem> couples
- `DictionaryElem`: for every unique term some information are maintained. They are the document frequency and collection frequency, the offsets to recover document ID and term frequency in the relative files, the dimension in bytes of the compressed document IDs and term frequencies, the inverse document frequency and finally the offset and the length of the skip list.
- `DocumentElem`: is the base object of the document table
- `Flags`: is useful to maintain the choices relative to the options selected by the user, such as stopwords removal and stemming, compression and debug mode.
- `Posting`: The document ID and frequency of a specific term in the dictionary are included in this element that composes a posting list.
- `SkipElem`: contains the information relative to a skip block, so the maximum document ID contained, the offset of the posting and two integers representing the number of posting or, if compression is enabled, the length of the compressed blocks to read.
- `SkipList`: is the structure managing skip information and holding maximum score values.

1.4 Compression

Compression has allowed to reduce the amount of data to be stored for the index. Two different algorithm have been used for document ID and term frequencies that compose the posting list of each term. It is convenient since the dimension of the inverted index file is very large and its reduction allows to reduce both storage space and computation time. The two algorithm used are:

- Unary
- Variable Bytes

1.4.1 Unary

The Unary compression has been used to compress the term frequency field of the posting lists. Since Java readings and writes are made for no less than one byte, encode one integer at a time means wasting memory. Instead of filling the byte with 0s in case of integers that need less than one byte, the integers are encoded as a list of integers in order to add additonal 0s only if the list is not encodable in a multiple of 8 bits.

1.4.2 Variable Bytes

This type of compression have been used for the compression of the document ID field of the posting lists. This algorithm have been chosen because the document ID can be very high, in the worst case equal to the number of documents in the collection. For this reason, the Variable Bytes algorithm is one of the most suitable. Exploiting this algorithm, it is possible to reduce the storage size of these numbers, stored in integers in the code, to the minimum number of bytes necessary for their binary representation.

1.5 Data Preprocessing

Data preprocessing is the initial step both in the index building phase, in which it is applied to the collection of documents, and in the query phase, in which it is applied to the input user query. The preprocessing optionally includes the removal of stopwords (common words that do not contribute significantly to information retrieval) and stemming.
The class `TextProcessor.java` provides text preprocessing functions, such as tokenization and removal of stopwords. It ensures that the text data is cleaned and transformed for efficient indexing. In detail:

- **Text cleaning**: removes noise and unwanted elements from the text, including URLs, HTML tags, punctuation, non-ASCII characters, and extra whitespaces. This ensures that the text is in a clean and consistent format.

- **Tokenization**: splits the cleaned text into individual words, or tokens. These tokens serve as the basis for constructing the inverted index.
- **Stopword removal**: optionally removes common stopwords, enhancing the quality of the index.
- **Stemming**: optionally applies stemming to the remaining words, reducing them to their root form.

2 Inverted Index Construction

The inverted index maps terms to the documents in which they appear, along with other statistics. In the application, the construction of the inverted index is divided into key phases, including data preprocessing, partial index building (exploting SPIMI algorithm), and merging of partial indices. The Java classes responsible for this process are located in the `it.unipi.dii.aide.mircv.index` package.

2.1 SPIMI Algorithm

The SPIMI algorithm (Single Pass In-Memory Indexing) is employed for indexing large collections processing each document in the collection and building partial indices in memory. In this project it is implemented in the `PartialIndexBuilder.java` class. First of all, each document in the collection is preprocessed using the `TextProcessor` and its characteristics are stored in the document table. Each time a new term is encountered, it is stored in the partial vocabulary, while if it is already present, only its document frequency and collection frequency are updated. During this phase, vocabulary, document table, and inverted index are kept in memory until a predefined memory usage threshold is reached. Once this condition is met, the algorithm sorts the partial dictionary in lexicographic order and then persists the data stored in memory structures to disk and frees up the memory. Both the vocabulary and the inverted index are stored on disk in blocks. To facilitate this process, a structure named `dictionaryBlockOffsets` is used to keep track of where to start reading from the dictionary file for each dictionary block. Furthermore, within each partial dictionary, two fields are maintained, each indicating the starting point of the partial posting: one for term frequency and another for the document ID. These are stored in separate files. After processing all the documents, the algorithm calculates and stores collection statistics, including the total number of documents and the total document length, on disk.

2.2 Merging Partial Indices

Once the partial indices are built, the next phase involves merging them into a complete inverted index that can be used during the query phase. This is implemented in the project in the `IndexMerger.java` class. Basing on the information stored in `dictionaryBlockOffsets`, one term for each dictionary block, which are lexicographically ordered, is placed into a priority queue along with its corresponding block number. The terms in the priority queue are ordered lexicographically and, if lexicographically equals, by increasing block number. During each iteration, one term is peeked from the priority queue and the associated partial posting list is read from disk. When the term peeked has been already encountered, the information for the current term is aggregated with the information for the previous term. However, if a different term is encountered, the entire posting list is saved to disk. The posting lists are already sorted in ascending order in the partial posting lists, with increasing block number. Therefore, concatenating them is all that is necessary. The algorithm ends when all the terms of each block have been processed. If the user has chosen the compression option, the term frequency is stored using the Unary compression while the document ID is stored using the Variable Bytes compression. If the posting list has a number of postings higher than a specific threshold, the skipping optimization is used. In this case, the posting lists are divided in sublists and their relative information are saved into skip elements.

3 Query Processing

The Query Processing module the component of the project responsible for handling user queries and retrieving relevant documents from the document collection. It encompasses various classes and algorithms tailored to deliver effective query processing capabilities. The primary objective is to facilitate the interaction between users and the underlying document collection. This interaction involves submitting queries and receiving a set of documents that are most relevant to the query. All the functionalities reguarding the query phase are implemented in `it.unipi.dii.aide.mircv.query` package. The module manages this process while offering a range of configuration options. The module comprises the following key components:

3.1 Query Management

This core functionality handles the execution of user queries, controlling aspects such as the number of results to return and the specific query type, which can be either disjunctive or conjunctive. Users can also choose between different scoring methods, specifically TFIDF or BM25, as well as between two query processing algorithms: Document-at-a-Time (DAAT) or MaxScore. First of all the necessary data structures are created, the query text is preprocessed and the posting lists of the query terms are read, then the function for the configuration requested is called. Finally, the algorithm presents the results to the user, including document IDs and their corresponding scores. These results are ordered by relevance, with the most relevant documents displayed first. This is implemented in the `Main.java` file.

3.2 Additional Data Structures

Underlying data structures play an important role in efficiently processing user queries. The module manages document tables, which store document information, and a dictionary, which maintains term statistics. These data structures are essential for lookup and retrieval operations. The query processing module performs query execution by applying various algorithms and strategies.

3.3 Score Calculation

To determine the relevance of documents, the module calculates scores using the chosen scoring method (TFIDF or BM25). The resulting scores guide the ranking of documents and help identify the most relevant matches. The module supports two scoring methods:

- **TFIDF (Term Frequency-Inverse Document Frequency):** this method calculates document relevance by considering the frequency of query terms within documents.
- **BM25:** BM25 is an advanced scoring method that incorporates term frequency, document length, and collection statistics to assess document relevance.

3.4 Query Processing Algorithms

The module offers two primary query processing algorithms:

- **Conjunctive Query Processing:** this algorithm retrieves documents that contain all the query terms, effectively implementing the logical "AND" operation.

  In this case the posting lists related to query terms are ordered by increasing document frequency (df). This ordering ensures that processing starts with the term that has the fewest matching documents, which typically leads to a faster query execution. For each document ID in the shortest posting list, the algorithm checks whether the other posting lists also contain the same document ID. If all posting lists have the same document ID, it means that the document contains all query terms. For each matching document, the algorithm computes a relevance score using either the TFIDF or BM25 scoring mechanism. These scores are based on the inverse document frequency (IDF) of the terms and their respective term frequencies in the document. The algorithm maintains a priority queue of results, ordered by score in decreasing order. This queue contains the top-k documents that satisfy the conjunctive query. After k documents are added to the results, the next ones are added only if their score is higher than the lowest one (the first of the priority queue). Once all documents are processed, the algorithm retrieves the top-k results from the priority queue. These results represent the documents that best match the conjunctive query, with their associated relevance scores.

- **Document-at-a-Time (DAAT) Query Processing:** DAAT processes documents one at a time, making it particularly efficient for handling large collections. It retrieves results incrementally based on the query's terms and scoring.

  The algorithm sets up a priority queue to store the results in decreasing order of relevance. This queue will contain the top-k documents that satisfy the disjunctive query condition that subsequently are reordered in decreasing order to be returned to the user. The algorithm begins by identifying the document with the lowest document ID among the postings of all query terms. This is done by iterating through the posting lists and looking at the first document ID in each posting list. For each current document, the algorithm scores the document's relevance based on the query terms present in that document. The relevance score can be computed using various scoring functions, such as TF-IDF or BM25. The algorithm maintains the priority queue of results. If the queue has fewer than k results, the current document and its associated score are added to the queue, otherwise it is added only if it has a score higher than the priority queue lowest one. After processing the current document, the algorithm seeks the document with the next lowest document ID among all the posting lists. This is done by advancing to the next posting in each posting list. The algorithm continues processing documents and scoring them until there are no more documents to examine. It terminates when it cannot find the next lowest document ID in any of the posting lists. Once the document processing is complete, the priority queue contains the top-k documents that satisfy the disjunctive query condition. These results are ordered by increasing relevance and can be presented to the user.

The choice of scoring method can significantly impact the relevance ranking of query results.

3.5 Query optimization: MaxScore

In scenarios where query processing speed is crucial, the module employs the MaxScore algorithm. This optimization technique utilizes upper bounds to streamline query execution while maintaining result quality. The Query Processing module simplifies the complex task of matching user queries to the document collection. It provides flexibility, efficiency, and advanced scoring options, allowing users to tailor the query processing to their specific needs. The algorithm sets up a priority queue to store the results in decreasing order of relevance. In order to execute the algorithm, the posting lists of the query terms are ordered by increasing score. Each term has associated a maximum score value, that is the maximum score reached by the documents of its posting list. The algorithm begins computing the upperbound for each query term, given by the maximum score of the previous ones, that is stored into an array list `ub`. Exploiting the upper bounds, the posting lists are divided in essential and non-essential. The first time the current document ID chosen is the minimum between all the posting lists, as described in the

DAAT algorithm. Then, for each document ID is computed the score as the sum of the scores of the essential posting list scores and, therefore, if the score summed with the non-essential upper bounds is above a treshold, the computation continues considering the non-essential posting lists score, otherwise it is skipped. As in the DAAT algorithm case, the priority queue is ordered in increasing order and, if the score of the new document is higher of the lower one in the queue, the new one is inserted in the result queue and the lowest is removed. At the end, the results are inserted in a priority queue ordered by increasing order in order to retrieve the documents in increasing document ID order.

3.6 Skipping Lists

The skipping list is a data structure used to improve the efficiency of processing large posting lists in the context of an information retrieval system. It allows for faster skipping over irrelevant postings to reach relevant postings, reducing the number of postings to examine during query processing. This concept is exploited by the `SkipList` and `SkipElem` classes that contain everything that is needed to exploit this mechanism. During the merging phase, while putting together the partial data structures, a consideration is made about the length of the complete posting list of the term being examined: if it is longer than a specific threshold, the skipping list related to that term is created. In particular, if n is the number of postings, $sqrt(n)$ skip blocks are created. It should be noted that this result is approximated by the excess, so all the elements will have equal length, except for possibly the last element. The skip list is composed by a number of skip elements equal to the number of skip blocks, with the information reported in 1.3. The skipping lists are stored on disk in a dedicated file. In query mode, the skipping lists of the query terms having long posting lists are loaded from disk (together with the partial posting list related to that skip block) and used to optimize time and memory. When searching for a specific document ID, in fact, there is no need to iterate all the posting lists: comparing the target document ID with the max document IDs of every skipping element already in memory it is only necessary to load from disk the partial posting list corresponding to that skipping element having the maximum document ID greater than the one to find with the use of the `nextGEQ()` function (note that the posting lists are always ordered in increasing document ID). This function also calls **`next()`** until it finds that document ID or the first one greater than it. If the right partial posting list is already available in memory (the one related to the skipping block with maximum document ID higher than the target), the `next()` function can be called in order to iterate over postings. The `next()` function has even another objective: if its parameter is set to true and it arrives to the last posting, the next partial posting lists (if present) has to be loaded. This is necessary in the conjunctive case for the query term of having the shortest posting list because it is the only one list that has to be sequentially iterated until a document ID results not present in at least one term's posting list is found, but also in DAAT algorithm.

4 Performance results

This section shows the results obtained from the execution of the application.

4.1 Indexing

Below is a comparison of the index size and the index time build with or without compression of the inverted index and with or without stopwords removal and stemming.

Structure	**Compression**		**No compression**	
	Sws	**No Sws**	**Sws**	**No Sws**
Document ID	1.007 MB	1.620 MB	1.870 MB	1.320 MB
Term frequencies	37 MB	65 MB	1.870 MB	1.320 MB
Vocabulary	103 MB	120 MB	944 MB	2.950 MB

Table 1: Size of the principal structures in different configurations

As expected, Document ID and Term frequency file size are the same in the case of no compression because both fields in the posting lists are stored in integers. In the compression case, the Term frequency file takes up less memory space due to the fact that the Term frequencies are low while the Document ID can be very high and can require an higher number of bytes. The vocabulary size is lower with the stopwords removal and stemming and compression.

Compression		**No compression**	
Sws	**No Sws**	**Sws**	**No Sws**
55 min	22 min	36 min	26 min

Table 2: Average execution times of for disjunctive queries

On the other hand, stopwords removal and stemming increments the time required for the index building while the compression reduces it a bit. Both compression and stopwords removal and stemming increase the performance during query phase other than reduce considerably the file sizes.

4.2 Query

In this section are present the execution time of the different query algorithms using the msmarco-test2020-queries.tsv file both using TFIDF and BM25 as scoring function.

4.2.1 Disjunctive query

Below is a comparison of the execution times of DAAT and MaxScore using the two different types of score.

DAAT		**MaxScore**	
TFIDF	**BM25**	**TFIDF**	**BM25**
36 ms	72 ms	14 ms	16 ms

Table 3: Average execution times of for disjunctive queries

As expected, MaxScore achieved better performance than DAAT both with TFIDF and BM25.

4.2.2 Conjunctive query

In this section are shown the average execution times with TFIDF and BM25.

TFIDF	**BM25**
13 ms	17 ms

Table 4: Average execution times of for conjunctive queries

5 Evaluation of the system

To assess the results of the implemented system, *trec_eval*, a tool that evaluates the precision and completeness. of information retrieval systems against a set of reference data (ground truth), was employed. The standard collections used for this phase are TREC DL 2020 queries and TREC DL 2020 qrels.

Metric	**terrier-BM25**	**Current solution**
RR	0.6186	0.8052
nDCG@10	0.4980	0.4865

Table 5: Evaluation results