UNIVERSITÀ DI PISA

Department of Information Engineering

Search Engine

Multimedia Information Retrieval and Computer Vision

Maria Inês Simões

ACADEMIC YEAR 2023/2024

Contents

1 Introduction **3**
1.1 Project Description 3
1.2 Organization of the repository 3

2 User Interface **4**

3 Indexing **4**
3.1 Document Processing 5
3.2 SPIMI 6
3.3 Merging 6
3.4 Compression 7

4 Query Processing **8**
4.1 Scoring Algorithms 8
4.1.1 DAAT Disjunctive 8
4.1.2 DAAT Conjunctive 9
4.1.3 MaxScore 9
4.2 Ranking Algorithms 9

5 Performance **10**

6 Limitations **11**

1 Introduction

This report sets out to outline the design, implementation, and evaluation of an indexing and query processing system for document information retrieval. The project aims to create a system capable of efficiently indexing extensive text document collections and executing ranked queries over the index created. The report discusses the system's architecture, including document processing, indexing, query execution, and user interface. The system's performance using a standard collection is evaluated, and potential enhancements are discussed.

1.1 Project Description

Information retrieval systems are crucial in managing and accessing vast amounts of relevant textual data. This report presents and explains a project focused on building the data structures needed for the retrieval of information and a query processing system for document retrieval. The project's primary objectives include indexing a large dataset, implementing efficient query execution algorithms, and providing an easy-to-use interface for query input and result retrieval. In the following sections of this report, each part of the project will be described in greater detail.

The project was implemented in Java, and the developed code is available on GitHub, which was used to manage the project development over time.

The document collection used for this project is the MS MARCO (Microsoft Machine Reading Comprehension) Passages Collection 2020 dataset. This collection contains 8,841,823 records, is 2.9GB in size, and is organized in the following format:

$$<\textbf{pid}>\backslash t<\textbf{text}>\backslash n$$

The algorithm's performance is assessed by comparing it to the TREC Deep Learning Track 2020, which is a standard collection. This collection comprises 200 passages, with 12KB in size. The validation file for the results of these passages contains the query ID, the document ID, and the corresponding scores.

1.2 Organization of the repository

The project includes the following main directories:

- *data:* Contains the MSMARCO Passages collection, output documents from indexing (including InvertedIndex, Lexicon, and DocumentIndex, merged and not merged) and evaluation input and output files.
- *documentation:* Includes the project description as well as the final report of the project developed.
- src:
  - *common*: Contains the TerminalDemo, Flags, Paths, and Util;

  - *compression*: Contains classes related to index compression, including the Unary and Variable Byte classes;
  - *evaluation*: Contains classes related to the performance evaluation of the query processing;
  - *indexing*: Contains classes related to the indexing component of the project, such as Merging, Indexing, and IndexUtil;
  - *preprocessing*: Contains classes related to the pre-processing component of the project, such as RemoveStopWords, Stemmer, and TextCleaner;
  - *queryProcessing*: Contains classes related to the query processing component of the project, such as Ranking, and ScoringStrategy.

The main data structures of the project include:

- *InvertedIndex*, *Lexicon*, and *DocumentIndex* to help save the information of each one of the chunks in an organized matter and later save it into a file;
- *TermStats* (indexing) and *TermDictionary* (query processing), which save term information such as offsets and collection and document frequency;

2 User Interface

The project includes a simple terminal user interface that prompts the user to choose one of three actions: indexing, query processing, or evaluation. Each option leads to different subsequent questions and results.

- Indexing: The user will receive the three data structures of the index (Document Index, Inverted Index, and Lexicon) as a result.
- Query Processing: The user is queried about different ranking and scoring options to rank the top k documents. The answers are saved as Flags, which are further explained in Section 4. The result includes the top k documents (represented by their document IDs), ordered by their scores.
- Evaluation: The user will have access to the files with the results from the test collection. There are six possible combinations of the three boolean flags of the test collection.

3 Indexing

The indexing component of the project aims to construct the data structures that constitute an index, including the inverted index, lexicon, and document index.

The documents are decompressed one at a time by opening an input stream to the compressed file, decompressing it using GZIP, and then parsing it as a TAR

archive. The algorithm reads the file with a BufferedReader, one line at a time. The division of the collection in blocks is done by monitoring the memory usage, and if it is over a certain threshold, which was set as 10% of the memory available, the partial data structures are written to the disk, and the memory is freed. The indexing process involves the following steps:

1. Separate the document ID and the text of the document through pattern matching;
2. Process the text from the document (further described in section 3.1);
3. Create and fill intermediate data structure files with the information provided by each block;
4. SPIMI algorithm for memory management (further described in 3.2);
5. Merge final data structures with compression (further described in 3.3 and 3.4, respectively).

The indexing results, saved in the *data/output* folder, include the partial data structures files created while reading each document and the merged and compressed version of the data structures.

The final merged data structures have the following structure, where "|" merely indicates a space in the final document:

- **LexiconMerged:** *term* | collection_frequency | document_frequency | offsetInvertedIndex | max_term_upper_bound | offsetDocIds | offsetFrequency | endOffset
- **InvertedIndexMerged:** *term* | documentID_1| ... | documentID_N | frequency_1 | ... | frequency_N
- **DocumentIndex:** *documentID* | document_length

3.1 Document Processing

The project's document processing component prepares the text for indexing and query processing. Each string will receive the following treatment: cleaning the text, performing tokenization, removing stopwords, and performing stemming. These processes must be performed in this specific order to have the expected results.

1. **Text Cleaning:** Remove capitalization, URLs, HTML rags, and non-alphanumeric characters, including punctuation. Reduce consecutive characters repeated three or more times to only two characters. Remove sequential spaces and any leading or trailing spaces.
2. **Tokenization:** Split the text into tokens based on the spaces between words, given that the punctuation has already been removed;

3. **Stopword removal and stemming:** Remove the terms included in a list of stop words. This list comprises the Natural Language Toolkit (NLTK) of the English Language and other standard stop word collections used in natural language processing. Stem the terms with the Porter Stemming Algorithm provided by the Java class *org.tartarus.snowball.ext.PorterStemmer*

3.2 SPIMI

Single-Pass In-Memory Indexing (SPIMI) is a common technique used in information retrieval systems to construct inverted indexes and other data structures from a collection of text documents. It operates on the principle of processing each document in the collection exactly once and building the inverted index entirely in memory during a single pass through the documents. This approach allows intermediate results for each Index data structure to be stored in the disk as the algorithm scans each collection document.

For each document, the algorithm will check if the memory is full by calculating the percentage of the memory and comparing it to a threshold of 10%. If the memory is considered full, the partial data structures will be sorted and written to the disk with the name of the data structure plus a number that indicates the block number inside the collection. The memory is then cleared by creating a new instance of the data structures and calling the garbage collector to remove the unused objects. If the memory is not full, the algorithm goes through each token of the current document and checks if the term already exists in the lexicon. The new information is written to the data structures if the term is not present in the local memory. If it is already present, the information is updated. The algorithm ends when all the documents and tokens have been processed.

3.3 Merging

After the algorithm has finished reading each document, the files are merged into one single file for each data structure. These files will be saved in the folder *data/output/merged* with the names *LexiconMerged.txt*, *DocumentIndexMerged.txt*, and *InvertedIndexMerged.txt*.

Each data structure has a different algorithm to accumulate the information from the multiple blocks created.

- The DocumentIndex structure iterates through the document index block files, reading each line and saving the information. After reading and sorting all entries, the function writes the merged entries to a single output file.

- The InvertedIndex structure uses a TreeMap to store posting lists and a PriorityQueue to manage term entries for efficient merging. An array of BufferedReader objects is created to read from each individual inverted index file, and iterators are initialized to go through each one of these files. As each line is read, terms and their corresponding postings are added to the PriorityQueue.

The function processes the terms by retrieving the first element of the PriorityQueue, merging posting lists when it finds the same term in a different block, and writing the merged postings to the output file. To avoid memory issues, when a term changes, the current postings of the term are written to disk which avoids keeping the whole uncompressed posting list of all the terms in memory.

Before being written to the final file, the document IDs and the frequencies are compressed (further explained in 3.4) with two different compression algorithms. Therefore, it is essential to keep the information of the start and end offset for each one of the lists of numbers. The offset information for each term is recorded in a helper map, which is then written into a text file.

- The Lexicon is computed similarly with the TreeMap and the PriorityQueue. It opens a BufferedWriter to write the merged lexicon and a RandomAccessFile for the merged inverted index file. The TreeMap stores term statistics, and an array of BufferedReader objects are created to read from the individual lexicon files. Each file's lines are iterated over, and terms are added to a PriorityQueue. This way, the merging is based on term order.

  The function reads and merges the term statistics from the lexicon files, updating the term's collection frequency and document frequency as necessary. Additionally, it accesses the offset auxiliary file from the previous step and computes a term upper bound. These values are then written to the merged lexicon file.

3.4 Compression

In the developed project, only the posting lists of each term are compressed in the Inverted Index structure. Two different algorithms for compression have been chosen based on the characteristics of each one of the values in the Posting List, the document IDs, and the term frequencies:

- **Unary:** represents a number n by writing n, followed by a zero.

  This algorithm only produces good compression results when applied to low numerical values. Therefore, it was chosen to compress the term frequencies in the posting list, typically values lower than ten.

- **Variable Byte:** represents an integer by dividing it into groups of seven bits, with the highest bit indicating whether more bytes follow.

  This algorithm is applied to the document IDs of the posting lists. Given the high number of documents, the document IDs are usually high numerical values, and therefore, this algorithm presents a more memory-efficient approach for their compression.

4 Query Processing

The query processing component of this project aims to retrieve the top k documents relevant to a user's query, utilizing their selected ranking criteria.

Upon selecting query processing in the terminal, the algorithm loads the document index and lexicon structures into memory. This in-memory approach significantly decreases query processing speed, as it minimizes file access to only the inverted index file. Additionally, the term details and posting list offsets within the inverted index file being stored in the lexicon structure enhance the retrieval of posting lists.

Next, users are prompted to specify flags that will dictate how the retrieval process and scoring methodology will happen. The algorithm then retrieves the k most relevant documents from the collection, identified by their respective document IDs and sorted by the scoring. Flag options include various choices, such as:

- DAAT or MaxScore for the processing of the posting lists;
  - If DAAT: Conjunctive of Disjunctive for the processing of the query;
- TFDIF or BM25 for the scoring function to use;
- Number of documents to be retrieved (value for k)

4.1 Scoring Algorithms

The user can choose from Document-At-A-Time (DAAT) or MaxScore using the initial flags for the document retrieval approaches.

DAAT emphasizes dynamic scoring. Documents are processed individually in memory, and scores are computed for each document, which allows for more efficient use of the memory and reduces processing overhead.

The algorithm developed is able to perform conjunctive (AND) and disjunctive queries (OR) when DAAT is selected, allowing users to specify whether they wish to receive documents only containing all query terms or at least one of them, respectively.

MaxScore falls into the category of dynamic pruning strategies. It prioritizes documents with higher scores and implements early termination. By prioritizing the retrieval of documents with high scores it is able to optimize query processing. It assumes that if a term upper bound does not reach a certain threshold, which is based on the query terms, the document will not be in the top k documents and therefore it is unnecessary to waste time computing the score with the less important terms.

4.1.1 DAAT Disjunctive

The function initializes and iterates through the posting lists of terms from the query. For each document, a score is calculated by summing up the ranking scores

of terms in the document. When one posting list is exhausted, it is removes it from the list. The function then checks if the document should be placed in the top k documents by comparing the score from the last document inserted, ensuring that the top K results always contain the documents with the highest scores. The function continues this process until all posting lists are empty. Finally, it returns the priority queue containing the top K results.

4.1.2 DAAT Conjunctive

Initially, the function retrieves and sorts the posting lists for each query term by size, so it only checks the term with the least amount of postings. The main processing loop begins by fetching the first posting from the smallest posting list. For each document, it verifies if the document ID is present in all posting lists using a binary search. If so, it calculates the document's score by iterating through each posting list and summing the contributions from each term. The loop continues until all relevant postings are processed and, in the end, returns the top k documents.

4.1.3 MaxScore

The function begins by initializing and sorting the posting lists in increasing order of the term upper bound of each term. It then computes the cumulative upper bounds for terms in the query to calculate the threshold values. The function iterates through the essential posting lists, calculates the scores using the DAAT approach, and continues with the non-essential posting lists. Before calculating the score for the non-essential posting lists, it checks if the sum of the current score with the cumulative term upper bound would be higher than the current threshold. If not, the function doesn't compute the score from the non-essential lists as the document could not make it into the top k documents. It then updates the threshold and pivot based on the top results obtained so far. The process continues until all relevant documents are examined and returns the top k results.

4.2 Ranking Algorithms

The document retrieval is prepared to compute two ranking algorithms: TF-IDF (Term-Frequency - Inverse Document Frequency) and BM25 (Best Matching 25).

TF-IDF assigns weights to terms based on their frequency in the document and their rarity across the document collection. It measures the importance of a term in a document relative to its occurrence in other documents in the corpus.

BM25 is a probabilistic model that calculates the relevance score between a query and a document based on term frequency and document length, considering the frequency of query terms within a document, the overall document length, and the inverse document frequency of the query terms across the entire document collection. The BM25 parameters are set to $B = 1$ and $K1 = 1.5$.

5 Performance

The present section of the report aims to present the performance results of the algorithms developed. The following table provides information about the time required to index the entire document collection and the sizes of the data structures created. The SPIMI row presents the time needed to read all the documents and the size of all uncompressed data structures, which are still divided into chunks. The Total row displays the time required to read, sort, compress, and merge all the structures, along with the final size of each structure.

	Time	DocId Size	Inverted Index Size	Lexicon Size
SPIMI	15 min 55 sec	109MB	1.77GB	81.3MB
Total	20 min 34 sec	108MB	732MB	43MB

To test the average response time for all the retrieval algorithms, the queries from the standard collection TREC DL 2020 are used and ranked with the system developed. The collection consists of 200 queries. The table presents the average processing times for three query processing methods using the two ranking functions implemented.

Query Processing		Average Time
DAAT Conjuntive	TFIDF	17.9ms
	BM25	14.5ms
DAAT Disjunctive	TFIDF	85.9ms
	BM25	87.8ms
MaxScore	TFIDF	32.4ms
	BM25	46.1ms

The results demonstrate that the DAAT Conjunctive method presents the fastest performance for both ranking functions since most documents are discarded from the beginning if they do not contain all the query terms. In contrast, DAAT Disjunctive performs significantly slower, indicating that handling disjunctive queries is more time-consuming given the larger quantity of documents. MaxScore presents a middle ground. However, since it was made with disjunctive query, it is more important to compare it to DAAT Disjunctive. The early termination significantly improved the algorithm's efficiency, with an almost 55% decrease in time.

Overall, BM25 tends to be slightly faster than TFIDF in the DAAT Conjunctive method, whereas in other methods, the performance differences between TFIDF and BM25 are less pronounced. This comparison highlights the efficiency of DAAT Conjunctive for query processing and the trade-offs in speed when using more complex query methods like DAAT Disjunctive and MaxScore.

6 Limitations

The project presents some limitations that could be developed to better time and memory management, as well as the quality of the search engine in general:

- Skipping: The project developed does not apply to skip in the query processing. Adding this feature, which was not planned since the beginning of the structuring of the project, would require changing both the indexing and the query processing structures to accommodate the skipping information as well as the function that would apply the skipping during the query processing. For that cause, it was not implemented.

- Caching: In the future, a caching feature for the queries as well as the uncompressed posting lists could potently improve the results for query processing.

- Alter the code to be more cohesive in all data structures. Some data structures represent the same concept, such as TermStats and TermDictionary as well as improve the memory usage of each function.