UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Artificial Intelligence and Data Engineering

MIRCV Project

Project report

Luana Bussu

Luca Caprioli

Academic Year: 2022/2023

Contents

1 Introduction

Information retrieval is a fundamental aspect of digital landscape, involving the organization and retrieval of information.
This project deals with the creation of a scalable search engine using Java, specifically focusing on the Passage ranking collection comprising 8.8 million documents available via the Microsoft repository. Divided into three modules, the project aims to construct a robust search system:
Index Construction: This involves creating essential data structures supporting search functionalities. These structures include the Dictionary, capturing unique terms and their statistics, the Document Index for storing document-related data, the Collection Index holding general collection information, and the Inverted Index, mapping terms to the documents in which they appear.
Query Processing: Our system includes a program capable of receiving user queries through a simple command line interface. It processes these queries, returning a list of most relevant document IDs with the corresponding score.
Performance Evaluation: A critical aspect is evaluating the effectiveness of our data structures. We evaluate their performance using a collection of predefined queries, measuring various parameters such as indexing time, query processing efficiency and effectiveness and memory usage.

The project aims to build and analyze the performance of a search engine. It involves detailed indexing and processing of queries trying to create an efficient information retrieval system for a large document collection.

The code is available in the following Github repository:
https://github.com/lubussu/MIRCV-project

2 Architecture

The architecture of our project is made up of several data structures managing information within the collection:

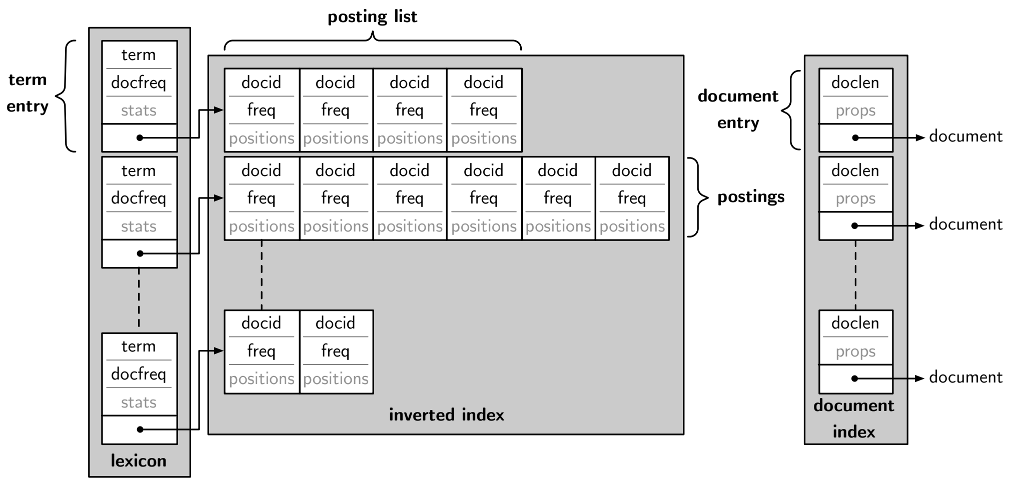

Figure 1: Typical inverted index components

2.1 Document Table

The purpose of the Document Table is to store information about individual documents through the 'DocumentElem' class. The information include:

- docNo: a string serving as the document identifier.
- docId: an integer providing a unique identification for each document in the collection.
- lenght: indicates the number of terms contained in the document following preprocessing.

2.2 Collencion Info

The objective of the CollectionInfo is to maintain information about the entire collection such as the number of documents in the collection and the total lenght of the documents computed as the sum of the length of each document. These details are useful in computing other statistic during the query processing.

2.3 Dictionary

The dictionary (or Vocabulary) contains crucial details about terms like document frequency, term statistics and the the pointer to the corresponding entry of the InvertedIndex that allow to identify the correct PostingList.

2.4 Inverted Index

The InvertedIndex (or PostongList) stores, for each term, the list of the documents where the term appears and the corresponding frequency (stored within the Posting class).

3 Text Preprocessing

The Text Preprocessing phase is a critical preparatory step for indexing text data in the search engine. It involves several operations aimed at transforming text into a standardized format suitable for indexing and retrieval purposes.

3.1 Text cleaning and Tokenization

Throuh using regular expressions, the text is standardized by:

- Removal of URLs
- Removal of HTML tags
- Reduce to lower case text
- Removal of punctuation
- Removal of extra-space

After text cleaning, tokenization is performed by splitting text into into individual words or tokens by splitting on spaces.

3.2 Stopwords removal and stemming

Additionally, the code removes stopwords, commonly occurring words like "the", "is", etc. from the tokenized text. These stopwords are sourced from an external file.
Stemming is a linguistic process that reduces words to their root form and is performed using the *EnglishStemmer* library. This helps merge different variations of the same word, contributing to more accurate recovery.

Here an example of a complete TextPreprocessing:

"The article explores various methods for data cleansing, such as removing URLs like https://example.com, handling HTML tags, ensuring lowercase consistency, eliminating punctuation marks, and condensing extra spaces."

$\downarrow$

[articl],[explor],[various],[method],[data],[cleans], [remov],[url],[handl], [html],[tag], [ensur],[lowercas],[consist],[elimin], [punctuat],[mark],[condens],[extra],[space].

4 Index Construction

4.1 SPIMI

Single-pass In-Memory Indexing is a fundamental algorithm in text indexing that deals with managing large quantities of data during the construction of the index. This algorithm works by processing data in a single pass, keeping only a portion of the data in memory at a time. This approach is particularly useful when working with large collections of documents and have limited memory resources.
While building the index, documents are read and analyzed one by one from the input file. Tokens are extracted from each document and information relating to the document identifier and the number of terms is recorded after the preprocessing phase within the Document Table.
Next, the construction of the inverted index takes place. The process involves maintaining a dictionary and a posting list for each term. During this phase, the presence of terms in the dictionary is checked: if a term is new, an entry is created in the dictionary and a new posting list for that term; if the term is already present, the information in the dictionary and in the associated posting list is updated (such as increasing the number of documents in which it is present).
Since index construction may exceed available memory capacity, a block-write-to-disk approach was adopted to free up memory space. During this phase, memory usage is constantly monitored and, when necessary, blocks are written to disk, keeping the complete list of processed terms in memory.

4.2 Merging Phase

After the indexing phase, information about a single term can be sparse in several blocks, so a merging operation is needed both for the Dictionary and for the Inverted Index. This process involves searching and updating the data contained in the various written blocks, combining the information relating to the same term and the respective posting lists. Both processes involve scrolling term by term of the list containing all the terms of the collection ordered alphabetically.
For both merging functions, a number of file channels are opened equal to the number of blocks written. For each term present in the list, a search is performed for the corresponding term in each of the files containing the indexed data.
During dictionary merging, the goal is to merge information about each term from different blocks, updating or creating corresponding entries in the final dictionary. This process also allows the calculation of the IDF for each term.
Similarly, during posting list merging, the various partial posting lists from the various files are aggregated and combined. This process allows you to obtain a complete posting list for each term, integrating additional information such as the maximum TF-IDF and the maximum BM25 for each term.

The merging operation allows to reconstruct a complete dictionary for each term and aggregate the various partial posting lists coming from the different files, combining them into a single complete list for each term. However, even during merging, the problem of managing the entire index in memory persists. As a result, we take the block-writing-to-disk approach again, but this time organizing the data in complete alphabetical order and maintaining complete posting lists for each term. This strategy allows you to maintain efficiency in the use of memory, saving the final index in an organized and ready way for effective operation in the indexing system.

The mentioned writing of the Inverted Index can be done either traditionally or in a compressed manner to save disk space.

4.3 Posting Lists Cache

After the merging phase, a cache is constructed to keep the largest posting lists in the index, specifically those associated with the most frequently occurring terms. This strategy aims to speed query processing by increasing the probability of having the most common posting lists in memory, minimizing disk reads.
The construction of this cache follows a simply approach: it involves sorting the terms in descending order of Document Frequency (an attribute within the dictionary) and adding the posting lists into the cache until the maximum allowable memory limit is reached.
When the program starts, the entire Dictionary and cached posting lists are read into memory. This step allows faster access to recurring terms during query execution, improving overall query processing speed.

5 Compression

Compression is applied to the Inverted Index, which constitutes the major memory usage. Specifically, for highly common terms, there can be particularly lengthy posting lists where the document's docid and its corresponding frequency need to be specified.
To store data more efficiently on disk, the option of compressed writing has been adopted. To simplify this process, posting lists have been stored by first recording all the docids and then all the frequencies. This enables the use of different compression algorithms as the two types of integers include significantly diverse ranges of values.

5.1 Unary Encoding

Unary Encoding is a simple compression technique used in information theory and data compression. It operates by representing each numerical value with a sequence of '1's followed by a '0'. The count of '1's before the '0' indicates the value being encoded.

Number	Code
0	0
1	10
2	110
3	1110
4	11110
5	111110

Figure 2: Unary Encoding Example

Unary Encoding, while conceptually simple, isn't optimal for efficient space utilization, especially when encoding larger numbers. As the encoded sequence length scales linearly with the represented value, it leads to longer representations for larger numbers, consuming more storage space. Consequently, this technique finds utility primarily in compressing frequencies characterized by smaller numbers.
However, for DocIDs that span a range from 0 to 8 million, a different encoding technique is necessary.

5.2 Variable Bytes Encoding

VariableByte Encoding is a more space-efficient compression technique used for encoding DocIds of the Inverted Index.
In VariableByte Encoding, integers are broken down into a series of bytes where the high-order bit (most significant bit) in each byte is used as a continuation flag. The remaining seven bits of each byte are used to represent a portion of the original integer, with the last byte signifying the end of the value.

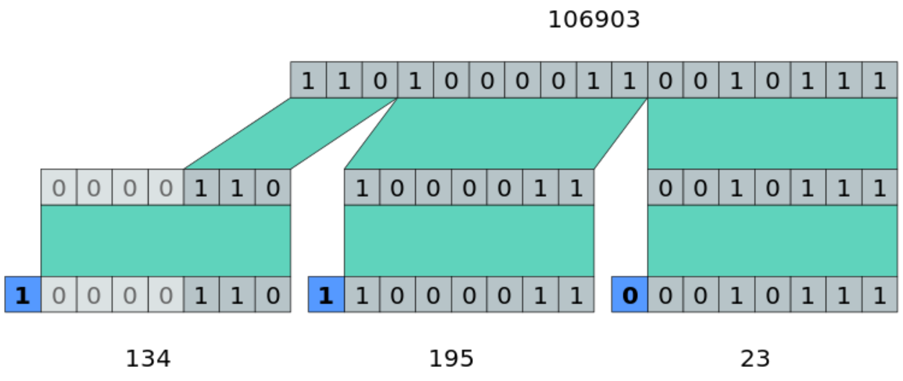

Figure 3: Variable-Byte Encoding Example

VariableByte Encoding is more space-efficient compared to Unary Encoding, especially for larger integers, as it can represent numbers using a variable number of bytes, minimizing the space required to store smaller values while scaling efficiently for larger ones.

6 Query Processing

The Query Processing phase allows the user to submit a query to the system to retrieve the top k relevant documents in the collection. As first thing, the query is submitted to the same text preprocessing of the documents described above; after that the user can choose the query execution modes choosing between:

- Conjunctive or Disjunctive Query
- TFIDF or BM25 score
- MaxScore or DAAT algorithm if the Disjunctive mode has been chosen.

In the world of information retrieval, the Query Processing phase connects what a user asks for with what's stored in the search engine's database. Its main job is to figure out and find the right documents that match what the user is looking for. The following sections describe the two query execution modes.

6.1 Conjunctive Query

The initial phase of query execution involves retrieving all the posting lists associated with the query terms. These terms are organized based on the increasing order of the Document Frequency parameter in the Dictionary. The posting lists are then accessed from memory or cache, possibly partially, considering skipping points if implemented.
The Conjunctive Query goal is to identify documents containing all the specified query terms. The process starts by examining the shortest posting list and iterates through its docIDs, checking for their presence in other posting lists. When a match is found in all the posting lists, the document score is updated and in the end, only the top-k document IDs are returned.

6.2 Disjunctive Query

In contrast, the Disjunctive Query aims to identify documents containing at least one of the specified query terms and then return the top-k results. The initial phase involves retrieving all the posting lists corresponding to the query terms from memory or cache. Subsequently, the DAAT or MaxScore algorithm is executed to process these lists.

DAAT

The Document-at-a-time algorithm starts by retrieving the minimum document ID among the posting lists associated with the query terms. Then, it iterates through the documents in ascending order of their IDs until it reaches the maximum document ID in the lists. Within each iteration, the algorithm maintains a priority queue to hold the top-k documents based on their scores. It computes the score for each document by traversing the posting lists related to the query terms.

For each posting list, it checks if the current document ID matches the document ID in the list. If there's a match, it computes the score for that document. The pointers in the posting lists are then updated to the next document.
The algorithm stores the documents with non-zero scores in the result priority queue, ensuring it maintains only the top-k documents with the highest scores. It proceeds to the next document ID based on the smallest ID found in the posting lists.
Finally, the DAAT algorithm returns a priority queue containing the top-k documents ranked by their scores.

MaxScore

The MaxScore approach is a dynamic pruning algorithm used to optimize query evaluation. Its goal is to improve efficiency in identifying the most relevant documents for a specific query by avoiding score computation for documents that won't be part of the final top-k results.

The MaxScore algorithm operates based on precomputed maximum scores for individual query terms. It begins by examining posting lists associated with query terms, using these maximum scores as upper bounds.
While processing the posting lists, MaxScore accumulates partial document scores, starting from the document with the minimum docID among those present in the posting lists of query terms.
During evaluation, MaxScore identifies two categories of posting lists: essential and non-essential. Essential lists represent query terms directly influencing the document's score under consideration. Non-essential lists, relying on the concept of precomputed maximum scores, are evaluated only if the accumulated score exceeds a certain threshold; otherwise, they are discarded.
Through this dynamic selection and utilization of precomputed maximum scores, MaxScore avoids computing scores for documents that have little chance of entering the final top-k results. This saves computational resources significantly, improving the overall efficiency of information retrieval processes.

Algorithm 3.1: The MaxScore algorithm

```
Input  : An array p of $n$ posting lists, one per query term,
         sorted in increasing order of max score contribution
         An array σ of $n$ max score contributions, one per query term,
         sorted in increasing order
Output : A priority queue q of (at most) the top $K$ ⟨docid, score⟩ pairs,
         in decreasing order of score
MaxScore(p,σ):
    q ← a priority queue of (at most) $K$ ⟨docid, score⟩ pairs,
        sorted in decreasing order of score
    ub ← an array of $n$ document upper bounds, one per posting list,
         all entries initialised to 0
    ub[0] ← σ[0]
    for i ← 1 to $n - 1$ do
        ub[i] ← ub[i − 1] + σ[i]
    θ ← 0
    pivot ← 0
    current ← MinimumDocid(p)
    while pivot < $n$ and current ≠ ⊥ do
        score ← 0
        next ← +∞
        for i ← pivot to $n - 1$ do                              // Essential lists
            if p[i].docid() = current then
                score ← score + p[i].score()
                p[i].next()
            if p[i].docid() < next then
                next ← p[i].docid()
        for i ← pivot − 1 to 0 do                                 // Non-essential lists
            if score + ub[i] ≤ θ then
                break
            p[i].next(current)
            if p[i].docid() = current then
                score ← score + p[i].score()
        if q.push(⟨current, score⟩) then                          // List pivot update
            θ ← q.min()
            while pivot < $n$ and ub[pivot] ≤ θ do
                pivot ← pivot + 1
        current ← next
    return q
```

Figure 4: MaxScore pseudo-code

A further improvement involves incorporating skipping points within the posting lists.

6.3 Skipping Points

The skipping procedure is a technique employed to optimize memory usage by storing only a subset of query term posting lists and fetching additional parts as required, rather than holding an entire list in memory. During the indexing process, the Skipping Points are created containing information about each block in the posting lists:

- maxDocId: The largest document ID within the block.
- blockStartingOffset: The starting position of the SkipElem in the SkipInfo file.
- block size: The number of postings contained in the block.

To iterate along the Posting items of posting lists related to query terms, the PostingList class implements two functions:

- **next()**: function to advance the current pointer within the posting list to the next posting. It facilitates the traversal through the posting list by moving to the subsequent posting, allowing efficient access to each document entry in the list. If there are no more postings available, the function sets the current posting to null.
- **nextGEQ(id)**: function for advancing the pointer within the posting list to the next posting with a document identifier greater than or equal to the given id. This function efficiently searches for the next entry satisfying the specified condition, making the search for higher or equal documentID optimized. If it doesn't find a matching posting or reaches the end of the list, it sets the current posting to null. This function benefits significantly from Skipping Points: by identifying the SkipBlock containing the specified ID through maxDocID checks in each SkipElem, it reads and decompresses only the necessary block, skipping unnecessary ones. This approach significantly improve query performance.

7 Performance Evaluation

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

7.1 Building Phase

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

Compression	**No Compression**
11min	11min

Table 1: Building Time

For both cases of compressed and uncompressed indexes the building times are the same.

	Compression	**No Compression**
Dictionary	212 MB	212 MB
Inverted Index	1,29 GB	1,65 GB
Tot	1,5 GB	1,86 GB

Table 2: Building Size

Regarding the dimensions of the two types of indexes it comes without surprise that the compressed one has reduced size of 0,36 GB compared to the uncompressed one.

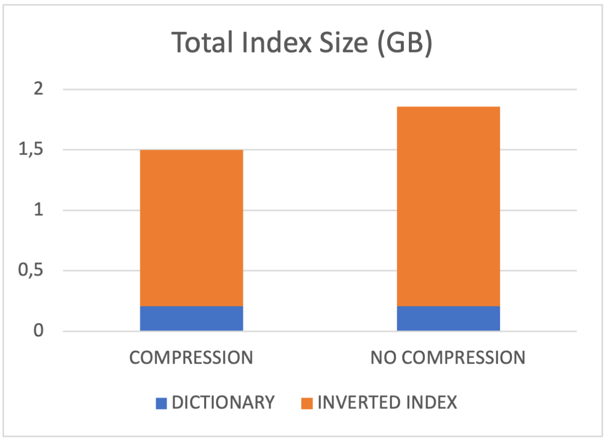

Figure 5: Size of Temporary Index

7.2 Merging Phase

For the merging phase both the time to build the final index and its size where compared with the addition of skipping lists.

Compression		**No Compression**	
Skip	**No Skip**	**Skip**	**No Skip**
3 min	2 min	2 min	1 min

Table 3: Merging Time

As expected the building time increases progressively with the addition of compression and the skipping lists construction.

	Compression	**No Compression**
Dictionary	106,3 MB	106,3 MB
Inverted Index	1,25 GB	1,62 GB
Tot	1,36 GB	1,72 GB

Table 4: Merging Size

In terms of size, without compression the index is bigger with skipping lists because of the addition of the SkipInfo file.

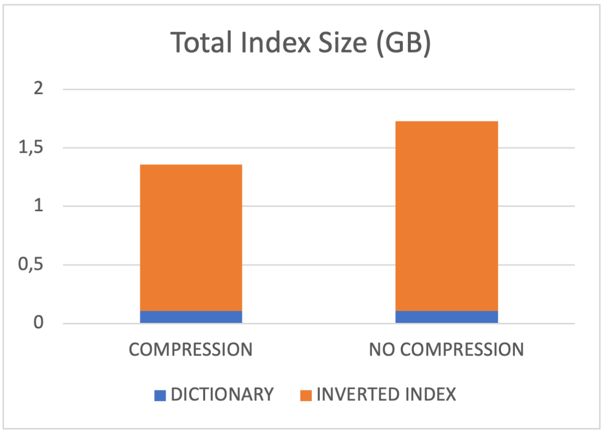

Figure 6: Size of Final Index

7.3 Evaluation of the system

Below are the results of the performance evaluation in terms of efficiency and effectiveness. Regarding efficiency, the test was conducted using the TREC DL 2020 queries with:

- stemming;
- stop-words removal;
- compression;

	Total Time	**Average Time**
Skipping	1s	7ms
No Skipping	1s	9ms

Table 5: Conjunctive Query Efficiency Performances

In the case of conjunctive query efficiency, while being obviously faster than its disjunctive counterpart, there is a difference of 2ms between the usage of the implemented skipping lists and using directly the posting lists.

	DAAT		**MaxScore**	
	Total Time	Average Time	Total Time	Average Time
Skipping	8s	42ms	2s	14ms
No Skipping	9s	45ms	9s	45ms

Table 6: Disjunctive Query Efficiency Performances

Regarding the disjunctive performance efficiency on the testing queries, it was conducted using both the algorithms implemented, DAAT and MaxScore. From the evaluation emerged that without the utilization of the skipping lists their times are more or less the same but, when they are introduced, there is a substantial difference in favour of the MaxScore algorithm of 28ms.

The implemented search engine's effectiveness was evaluated through the use of trec_eval [1], a tool that evaluates the performance of a retrieval system against a set of reference data. The evaluation was performed using the standard TREC DL 2020 queries and TREC DL 2020 qrels collections in the following way:

- no stemming;
- score BM25;

- disjunctive mode;

Metric	Present Solution	terrier-BM25
DCG@10	0.5468	0.4980
MAP	0.1512	0.3021
RR	0.7960	0.6186

Table 7: System Effectiveness Performances

The results presented in Table 7 are compatible with results from other well-established systems underscoring the effective performance and credibility of the developed system.

References

[1] Nick Craswell, Bhaskar Mitra, Emine Yilmaz, and Daniel Campos. Overview of the trec 2020 deep learning track, 2021.

[2] Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. In *Proceedings of the [nome della conferenza o della pubblicazione]*. [Nome dell'editore, se applicabile], anno di pubblicazione.