UNIVERSITÀ DI PISA

MSc in Artificial Intelligence and Data Engineering

Multimedia Information Retrieval and Computer Vision Project Report

Team members:

Matteo Dal Zotto

Leonardo Bargiotti

ANNO ACCADEMICO 2023/2024

Indice

Capitolo 1

Introduction

1.1 Description

The goal of this project was to build a simple search engine based on an inverted index and capable of execute some queries on it, retrieving the best documents for each query. Starting from the MSMARCO passages collection (which is available at the following link:
https://msmarco.z22.web.core.windows.net/msmarcoranking/collection.tar.gz) we developed our project executing three main steps:

1. The indexing of the documents
2. The processing of the queries
3. The evaluation of the performances

1.2 Project Structure

The project is divided in two folders: src and data. In this chapter is described the first, in the next the latter. There is a file Configuration with the paths used in the project and the values of the constants for the formulas we used (like k1 for BM25).

```
src
├── main
│   └── java
│       ├── index
│       ├── query
│       └── performanceEvaluation
└── test
    └── java
        ├── index
        └── query
```

The module **index** contains the Java classes that implement the base structures of the project (such as Lexicon, Posting List, ...), the classes that deal with compression and decompression of the data, a file FileUtils which implements some methods that deal with creation and removal of directory, and a file Preprocesser which contains methods that allows us to preprocess the collection and the queries (stemming, stop words removal,...). The module also contains the SPIMI class, which perform the SPIMI algorithm to create the intermediate inverted indexes and the document index, and the Merger class, which merges the intermediate indexes in order to create the final inverted index and the lexicon, both of the classes save the outputs on the disk. There's an IndexingMain class which starts the execution of SPIMI and Merger.

The module **query** contains the class DAAT, which implements the homonym algorithm for disjunctive and conjunctive executions, the class MaxScoreDynamicPruning, which implements the execution of the MaxScore pruning optimization and the class Scorer to calculate the TFIDF and BM25 scores.

The module **performanceEvaluation** contains a class Main to test the performance of the information retrieval system. All the modules contain the relative testUnits, except performanceEvaluation, which test the main functionalities of the methods using Junit.

Capitolo 2

Indexing

2.1 Overview of Indexing Process

The indexing module is one the two main parts of the information retrieval system, designed to handle large-scale data and convert it into an organized, searchable structure. This process involves several stages, from initial document processing to the creation of a comprehensive inverted index.

2.2 Single-Pass In-Memory Indexing

Single-Pass In-Memory Indexing or SPIMI is an algorithm that parses documents and turns them into a stream of term-docID pairs

2.2.1 SPIMI Algorithm (Class: Spimi)

Spimi (Class: Spimi): The Spimi class implements the spimi method which process the input file (for this project the msmarco collection) by reading each line, splitting it into its terms, cleaning the split String and writing the term-docId pair to the memory and then to a file. To ensure a smooth process, the blocks have a maximum size of 20% of the available memory during runtime. This produces several blocks each containing a part of the term-docId pairs.

2.2.2 Merge Algorithm (Class: Merge)

write (Class: Merge): the write method in the Merge class is designed to read the terms from each block made by the spimi algorithm and the lengths of each document and build the docIds file, the freqs file, the skippingBlock file and the Lexicon file. The first and second are written using the respective write methods of the DocIdFile and the FrequencyFile; these methods return an offset which is later used while writing data on the skippingBlock.

2.3 Index Structure and Organization

docIds file
This file contains the docIds of the documents and their respective length

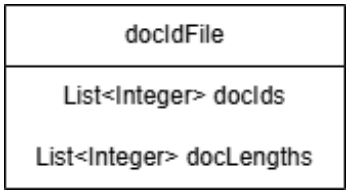

Figura 2.1: docIdFile structure

freqs file
This file contains the frequencies

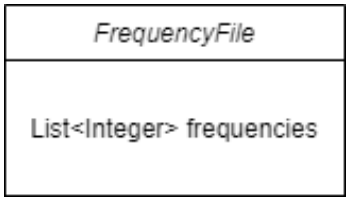

Figura 2.2: FrequencyFile structure

skippingBlock file The skippingBlock file contains the offsets and dimensions of each block of docIds and frequencies, the number of postings of the block and the highest docId contained in the block. These offsets and sizes are used to perform quicker retrieval operations during the queries, skipping multiple rows of the file, while the docIdMax enables the use of nextGEQ.

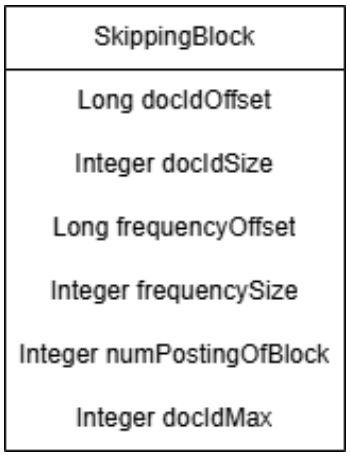

Figura 2.3: SkippingBlockFile structure

LexicondData class A LexiconData class object contains multiple informations about each term processed during the spimi, such as the inverse document

frequency, upperTFIDF (highest TFIDF for the term), upperBM25 (highest BM25 for the term), numBlocks (indicates in how many blocks the term is split) and offset_skip_pointer. The size of the data that is written on the disk is 24 bytes given by 4bytes * (n_float + n_int) + 8bytes * n_long.

Lexicon class A Lexicon class object contains a LexiconData object which is used to write a LexiconData entry to the disk and an LFUCache object which is used to speed-up the retrieval time of frequently requested terms. The purpose of this class is to facilitate the use of LexiconData and host the cache for the lexicon entries.

2.4 Data Encoding and Compression

Unary Encoding (Class: UnaryCompressor): The UnaryCompressor class is designed to provide an efficient encoding mechanism for data that has a skewed distribution, which is often the case with term frequencies in document collections. In unary encoding, a number is represented by a series of 1's followed by a 0. For example, the number 3 is encoded as " 110". This method is particularly effective for encoding smaller numbers, which are common in term frequency data, leading to a more compact representation.

Variable Byte Encoding (Class: VariableByteCompressor): The VariableByteCompressor class implements variable byte encoding, a technique that represents integers in a variable number of bytes, making it highly space-efficient for a wide range of integer sizes. This encoding method is especially advantageous for indexing as it allows for the compression of document identifiers and term frequencies without a fixed-size constraint. Each number is broken down into 7-bit chunks, with the most significant bit in each byte used as a continuation marker. This approach ensures that larger numbers occupy more space, while smaller numbers are stored more compactly, leading to overall space optimization.

Capitolo 3

Query Processing and Evaluation

3.1 Overview of Query Processing

Query processing is responsible for interpreting user queries and retrieving relevant documents from the indexed data. This process involves several key steps, from query parsing to document scoring and ranking.

3.2 Query Parsing and Execution

Query processing (Class: Processer): The Processer class ensures that queries are parsed, preprocessed, and accurately interpreted, aligning user intent with the indexed data.

Handling Query Types: The system is able to process two different query formats, conjunctive (AND) and disjunctive (OR). This capability allows users to select their search criteria and retrieve results that better match their information needs.

3.2.1 Document At A Time (Class: DAAT)

Streamlined Document Assessment: Employing the Document At A Time (DAAT) technique, the DAAT class search each document for the presence and frequency of query terms and compute a relevance score for the retrieved documents. This method enables concurrent evaluation of all query terms for each document compared to other techiques like TAAT.

3.2.2 Dynamic Pruning (Class: DynamicPruning)

Improving Query Efficiency: The DynamicPruning class implements a dynamic pruning algorithm based on the highest attainable document score, which ease

the estimation of document relevance. By Evaluating the score and comparing it with a relevance threshold, the system efficiently filters out documents considered improbable to be pertinent, improving the query execution time.

3.2.3 Conjunctive and Disjunctive Queries

In **conjunctive** queries, which necessitate of the presence of all terms within a document, the system uses rigorous matching standards, retrieving only documents containing all query terms. Differently, **disjunctive** queries, allows the presence of any term, expanding the search to documents which contain at least one of the query terms.

3.3 Scoring and Ranking

Computing Document Relevance (Class: Scorer): The Scorer class implements two distinct scoring models TF-IDF (Term Frequency-Inverse Document Frequency) and BM25 (Best Match 25), both of which compute the importance of query terms within each document.

Ranking Structure (Class: TopKPriorityQueue): The TopKPriorityQueue class, a extended version of the PriorityQueue Java class, is used to keep the correct order of the results. This extended priority queue maintains a limited collection of the highest-scoring documents, often referred to as the top K results.

Presentation: After the scoring phase, the documents housed within the TopKPriorityQueue are presented to the user based on their relevance scores, with higher scores receiving precedence.

3.4 Optimization

To improve the execution time of query is fundamental to have an efficient memory usage and a system to traverse the posting list in a non-linear way. The system uses two techniques to address this challenge: the LFUCache (Least Frequently Used Cache) and the SkippingBlock.

Frequently Accessed Data (Class: LFUCache): The LFUCache class functions on the premise that frequently queried terms or documents should be quickly accessible in order to minimize latency and reduce query response times. The LFUCache ensures that the most relevant data remains readily available in memory.

Streamlining of Index Traversal (Class: SkippingBlock): The SkippingBlock class is used to streamline the index traversal process during query execution, allowing the system to skip over unused segments of posting lists. This reduce the execution time of queries, removing the need to linearly search through the posting lists.

Capitolo 4

Performance Evaluation

4.1 Evaluation Methodology

All tests and benchmarks present in this chapter were executed on the same machine (iMac M1) in order to ensure a standardized and environment for our tests.

4.2 Indexing Performance

4.2.1 SPIMI Time

In the following section we analyzed varying SPIMI processing times across different configurations, with the most rapid processing occurring when both compression and text preprocessing are enabled.

- **Without Compression and Preprocessing**: $49 : 46 \pm 1 : 30$ minutes
- **With Compression, Without Preprocessing**: $50 : 23 \pm 1 : 30$ minutes
- **Without Compression, With Preprocessing**: $17 : 26 \pm 1 : 30$ minutes
- **With Compression, With Preprocessing**: $17 : 12 \pm 1 : 30$ minutes

4.2.2 Merge Time

In the following section we analyzed varying Merge processing times across different configurations, with the most rapid processing occurring when both compression and text preprocessing are enabled.

- **Without Compression and Preprocessing**: $47 : 39 \pm 1 : 30$ minutes
- **With Compression, Without Preprocessing**: $50 : 36 \pm 1 : 30$ minutes
- **Without Compression, With Preprocessing**: $21 : 40 \pm 1 : 30$ minutes
- **With Compression, With Preproccsing**: $21 : 22 \pm 1 : 30$ minutes

4.3 Storage Space Efficiency

In the following table we analyzed the storage space efficiency across different configurations. It is easy to see that the absence of preprocessing and compression produce a significant increase of the files sizes.

Compression/Preprocessing	No	Yes
No	• docIds: 2.7 GB • freqs: 675.5 MB • lexicon: 66.2 MB • docTerms: 35.4 MB • skipping: 92.5 MB	• docIds: 1.28 GB • freqs: 320.3 MB • lexicon: 55.3 MB • docTerms: 35.4 MB • skipping: 67 MB
Yes	• docIds: 1.61 GB • freqs: 62.6 MB • lexicon: 66.2 MB • docTerms: 35.4 MB • skipping: 92.5 MB	• docIds: 762.7 MB • freqs: 28.5 MB • lexicon: 55.3 MB • docTerms: 35.4 MB • skipping: 67 MB

4.4 Observations

As observed from the results, the Merge and SPIMI times are practically identical for each configuration.

Preprocessing has a much more significant impact, significantly reducing execution times.

Compression, on the other hand, does not notably affect the execution times. However, as shown by the file sizes in the next section, compression dramatically reduces the size of the data, highlighting that its main advantage lies in space management rather than processing speed.

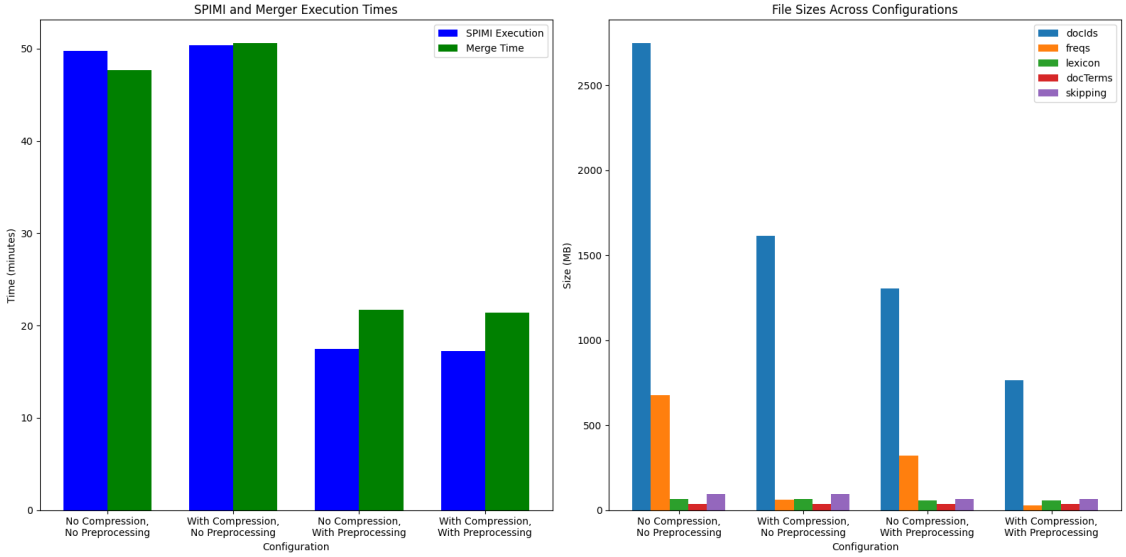

Figura 4.1: Time and Storage Across Configurations

4.5 Retrieval Effectiveness and Query Execution Time

The query results are included only in cases where preprocessing is enabled, both with and without compression.

Without preprocessing and without compression, an error occurs when attempting to open the DocId file. This is due to the file size exceeding 2GB, causing an error in the `MappedByteBuffer` library. [1].

In cases where preprocessing is not enabled but compression is, the execution time is excessively high and cannot be compared to the two cases presented below.

4.5.1 Query Execution Time

To conclude, in the following table it is possible to see the different execution times for both scoring method, with and without cache and with and without Dynamic Pruning.

As shown in the table below 4.1, compression does not impact the results in any way. Additionally, unsurprisingly, the use of the cache slightly improves the performance.

Configuration	Method	Cache	Mean	Std Dev
Both Compression and Preprocessing Enabled	TFIDF DAAT	Without Cache	36.77	29.35
	TFIDF DAAT	With Cache	24.58	22.41
	BM25 DAAT	Without Cache	25.05	22.84
	BM25 DAAT	With Cache	24.43	22.37
	TFIDF DP	Without Cache	24.67	22.55
	TFIDF DP	With Cache	24.29	22.05
	BM25 DP	Without Cache	25.19	22.76
	BM25 DP	With Cache	25.09	22.84
Without Compression, With Preprocessing	TFIDF DAAT	Without Cache	33.5	27.15
	TFIDF DAAT	With Cache	20.68	18.91
	BM25 DAAT	Without Cache	20.75	19.84
	BM25 DAAT	With Cache	20.43	18.37
	TFIDF DP	Without Cache	20.27	18.85
	TFIDF DP	With Cache	20.09	18.45
	BM25 DP	Without Cache	21.12	20.06
	BM25 DP	With Cache	20.49	18.54

Tabella 4.1: Query Times(ms) for Different Configurations

[1]For more details, see the bug report: https://bugs.openjdk.org/browse/JDK-6347833

4.5.2 Query Evaluation Metrics

The following TREC metrics were employed to evaluate retrieval effectiveness.

- **Precision at 10 (P@10)**
- **Recall at 10 (R@10)**
- **Normalized Discounted Cumulative Gain at 10 (ndcg@10)**
- **Mean Average Precision at 10 (map@10)**

4.5.3 Retrieval Effectiveness

From the Figure 4.2 is possible to see the scores obtained for each selected metrics with two different configurations, with compression and without compression, for both scoring functions.

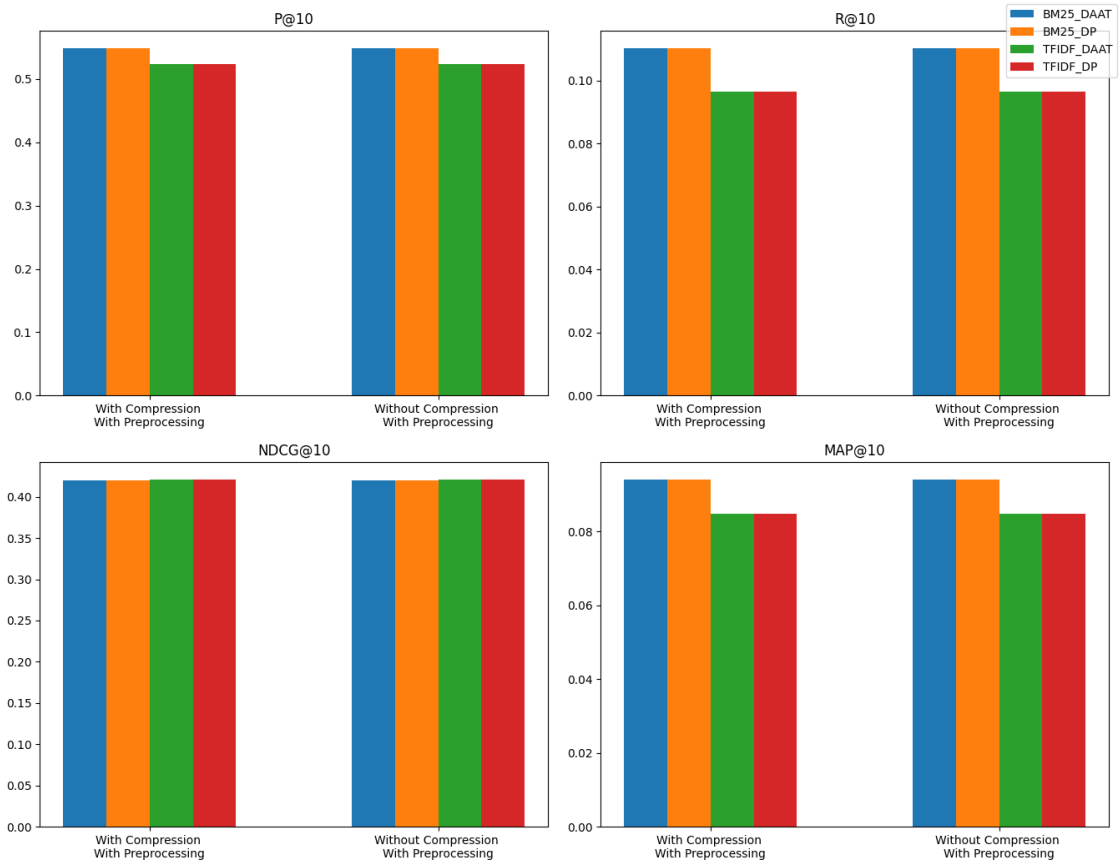

Figura 4.2: Retrieval Effectiveness