

# UNIVERSITÀ DI PISA

Master Degree in Artificial Intelligence and Data  
Engineering

Multimedia Information Retrieval and Computer Vision

## **MIRCV Project**

Group Project for Multimedia Information Retrieval and  
Computer Vision

Academic year 2023-2024

**Lorenzo Nelli, Gianluca Antonio Cometa, Andrea Sottile**

**Github repository:**

<https://github.com/andreaSottile/MIRCV-Project-2024>

# 1. Introduction

The goal of this project is to build an efficient inverted index for the MSMARCO "Passage ranking" dataset, which is a large collection of text passages used for training and evaluating information retrieval systems.

The MSMARCO (Microsoft MACHine Reading COmprehension) dataset is widely recognized for its application in benchmarking search engines and question-answering systems.

It consists of millions of passages derived from web documents, and it presents a challenging dataset due to its size and the diversity of content.

The "Passage ranking" subset used in this project includes approximately 8.8 million passages, each of which is associated with a unique identifier and a text field containing several sentences.

The primary objective of this project is to create an inverted index that enables efficient search and retrieval of passages based on user queries.

The project involves several key steps:

1. **Data Preprocessing:** The raw text data undergoes preprocessing to standardize it for indexing. This includes tasks such as tokenization, stemming, stopword removal, and handling special characters.
2. **Inverted Index Construction:** The core of the project lies in constructing the inverted index, which maps terms to the passages in which they appear. This involves building a document table, a lexicon, and associated postings lists that store term occurrences.
3. **Query Processing:** Once the index is built, the system is capable of processing user queries, retrieving relevant passages, and ranking them based on their relevance to the query terms.
4. **Performance Optimization:** The project also explores techniques to optimize the performance of both indexing and query processing, including the application of basic compression techniques to reduce the size of the index and improve query speed.

This document will provide an in-depth explanation of the program's capabilities, its internal workings, performance metrics, limitations, and the key functions and modules that make up the system.

By the end of this document, readers should have a comprehensive understanding of how the inverted index is constructed and utilized to facilitate efficient information retrieval from a large-scale dataset like MSMARCO.

Compared to the provided specification all mandatory requirements have been implemented, the project also provides a textual interface with which to test different features, including: index or re-index of the collection, the choice to apply stemming or stopword removal etc.

In addition, all optional requirements have been implemented, in particular:

- We introduced compressed reading, performing run-time decompression without the need to write to hard disk;
- Index compression using unary or gamma compression at user's choice, going to perform encoding of docIDs using GAP encoding;
- We have implemented BM11, BM15 and BM25 as scoring functions in addition to TFIDF;
- We also implemented Query Processing Efficiency through the introduction of ternary search on the lexicon file that contains the offsets of the inverted index file with a related `NextGEQ_line()` that provides the offsets related to the position of the cursor on the file;

In addition, we ran into several problems with scalability and creation time of the inverted indexes on the overall collection, to do this we introduced:

- a **multiprocessing** mechanism that actually allowed us to parallelize the creation of the inverted indexes while maximizing CPU utilization on the server that hosted the execution. We used for our work a machine with 2 x Intel Xeon E5645 @ 2.40GHz 12 Cores and 24 Logical Processors.

This approach then provided a mechanism for partitioning our dataset to create the inverted index portions which, upon completion were merged with dedicated functions of merge indexes.

Regarding the evaluation-related part of Query Handler, we realized that the evaluation times with the TREC framework required, for each index, a rather long execution time.

For this reason we introduced:

- additional **caching** mechanism that actually allows the lexicon with related configurations to be saved in memory for a inverted index (with stemming, no stopword etc..) this definitely saw an overall improvement in performance in disk access.

## 2. Modules

This section will show the modules that make up the core of our project, including a brief summary of the main functionalities.

### Document Processing

This module focuses especially on the file and collection access methods. The compressed read mechanism is implemented in this module with the use of the “**extract\_dataset\_from\_tar**” function, as well as all fetch functions like “**fetch\_n\_data\_rows\_from\_collection**” from the dataset that then allow through parameters not to read the entire collection.

### Preprocessing

Each feature is a separate function, independent from others.

This is the order they’re called:

- removal of special characters and punctuation signs (substitution with blank spaces)
- conversion to lowercase
- tokenization
- stopwords removal (flag to disable it)
- stemming (flag to disable it)

### Compression

We implemented 3 ways of working with index compression:

- no compression at all
- unary compression
- gamma compression

It’s possible to choose which algorithm to apply before starting the indexing phase, as a parameter that can be changed from the user interface. We also implemented the related decode functions.

We noticed when we started the execution of the project that the unary compression is unfeasible for the disk space of the result.

### Cache

We have implemented a Cache mechanism that allows us to go and save in memory the previous output of any query with a specific configuration. So the mechanism acts on the Lexicon and Index by going to save in memory, after the first execution, the result of the “**fetch\_posting\_list**” that is called whenever we go to execute a query.

This allows us, where it is already cached, to avoid unnecessary disk reads by going to fetch directly from memory. This is an enhancement that we introduced when we started working for the evaluate part of the TREC.

## Evaluation

The evaluation can be performed on 2019 or 2020 TREC collections.

The function “trec\_eval” is available online, so we have only implemented the basic data conversion to assemble the parameters accordingly.

The results are discussed in the following pages.

## Interface

The interface allows us to create and manage a CLI interface to perform index searches, configure search parameters, and manage index configuration to process queries.

Some of the main points include:

- **Parameter Access Functions:** The “get\_parameter, get\_index\_title, flag\_restart\_needed, and clear\_restart\_needed” functions allow access and modification of global parameters. These are used to configure and update the index and search criteria;
- **CLI Menu Management:** “print\_menu” and “handle\_selection” build and navigate a colorful CLI menu that allows the user to select configurable options. The “handle\_selection” function performs the corresponding action or accesses a submenu;
- **Parameter Update Function:** “update\_parameter” updates user-specified search parameters, triggering the “index\_restart\_needed” flag to force index reinitialization when needed.

```
Menu:
1: Change parameters
   1: file_count: 500
   2: method: disjunctive
   3: scoring: BM25
   4: k: 10
   5: skip_stemming: False
   6: allow_stop_words: True
   7: compression: no
   8: index_title: PIPPO
   9: Search algorithm: ternary
  10: Back to main menu
2: Save parameters (Actual Index Name: PIPPO | Restart Needed:False)
3: Load parameters (Actual Index Name: PIPPO)
4: Query: ""
5: Execute | Restart Needed:False)
6: Exit

Select an option:
>? |
```

## Inverted Index

The inverted index module is the core module of all our work and implements the basic functions to support the data structure that enables its operation.

This module includes the declaration of the Class and several sub-functions that work together to create a highly optimized index.

- **Inverted Index Creation:** The indexing process starts with the “index\_setup” function, which creates the configuration index file containing all the parameters necessary for the index to function properly.  
The second step of the process involves the “scan\_dataset” function that iterates through the dataset, extracting and preprocessing text from each document to have the tokens that need to be included in the index.  
Specifically, for each unique token the index contains a postings list that records all the passages (or documents) where the token appears.  
The inverted index structure is fundamental to the program’s ability to quickly retrieve relevant documents in response to queries.  
Other really important function is:
  - `save_on_disk`: Saves the generated index to disk, ensuring that it can be loaded and reused in future sessions without the need for reindexing.
- **Document ID Assignment:** Each passage is assigned a unique document ID during indexing. This ID is a consecutive number unique for the index;
- **Postings List Management:** The posting lists are ordered by docID and stored with GAP Encoding and the compression algorithm selected in the configuration;
- **Lexicon Management:** The lexicon is a critical part of the index, mapping each unique token to its corresponding postings list. For each token in the collection it’s generated a lexicon entry with:
  - *Token*
  - *Document Frequency*
  - *Offset of the posting lists in the inverted index file*

## Query Handler

Class that connects one InvertedIndex object to all the possible operations that the user could launch. There's a default InvertedIndex connected to the QueryHandler when the program starts its execution. The main features offered by this class are:

- interfaces for the InvertedIndex's functions and the management of Posting Lists
- interface functions to edit all the parameters relative to indexes and queries
- query function and the related part to compute the score
- buffer to store the results of a query
- the ternary search in the lexicon file

## 3. Indexing Phase

In the first phase of indexing we realized that with the entire collection the indexing time would be very long and could not fully utilize all processor cores.

To improve this phase therefore we approached a multi-processing mechanism that involved 5 main points:

- **Index Setup:** Imports various modules to extract, preprocess, and compress the dataset. Configures and checks for pre-existing indices on disk to avoid duplication;
- **Parallelized Dataset Processing:** Uses “open\_dataset\_multiprocess” to split the dataset into sections and process each portion in parallel, speeding up index construction;
- **Chunk Creation and Management:** The “read\_portion\_of\_dataset” function processes each document, generates posting lists (references to documents for each term), and, with “close\_chunk”, saves these lists in manageable-sized chunks;
- **Chunk Merging and Compression:** The “merge\_chunks” function combines the chunks into a single index. It uses gamma or unary compression and gap encoding to minimize space usage;
- **Final Writing and Cleanup:** “write\_output\_files” saves the final index and lexicon to disk, deleting temporary chunks to optimize disk usage.

At the end of this phase then in output we get a set of inverted indexes each inherent to its own portion of the dataset, we then implemented a dedicated **MERGE** part that would

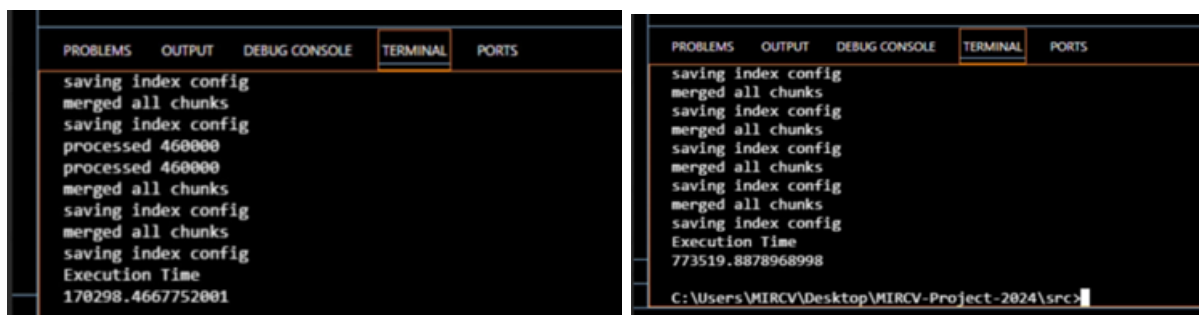
allow us to merge all portions of the index executed in multi-process mode, thus obtaining a single global index.

The 5 main points are:

- **Initial Setup:** The source and target folders, compression parameters, and some global variables to track stemming and stopword options are defined.
- **Opening Files:** All partial files to be merged, including the index and lexicon files, are opened. Each partition is then analyzed to identify any missing documents, which are retrieved and added to the index if necessary.
- **Creating the Global Stats File:** Individual statistics files are combined to create a single file that describes the global collection statistics, with documents sorted by their identifiers.
- **Merging Lexicon and Posting Lists:** The code reads each token from the various partial lexicon files, extracts posting lists, and converts them into a standardized format. It then creates a global, ordered posting list, with optional compression to reduce file size.
- **Writing Output Files:** The posting lists and final lexicon are saved to the target index and lexicon files. The code calculates and prints the total execution time for the merge process.

We used for our work a machine with 2 x Intel Xeon E5645 @ 2.40GHz 12 Cores and 24 Logical Processors.

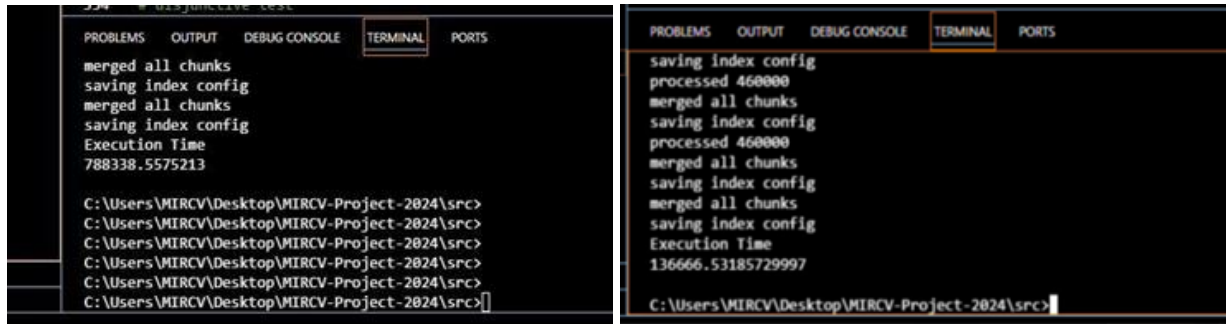
Specifically, we performed 4 macro executions in multi-process of which the timestamp of each can be found below, there are 4 since they are the possible combinations of “stemming\_flag” and “stop\_words\_flag.”



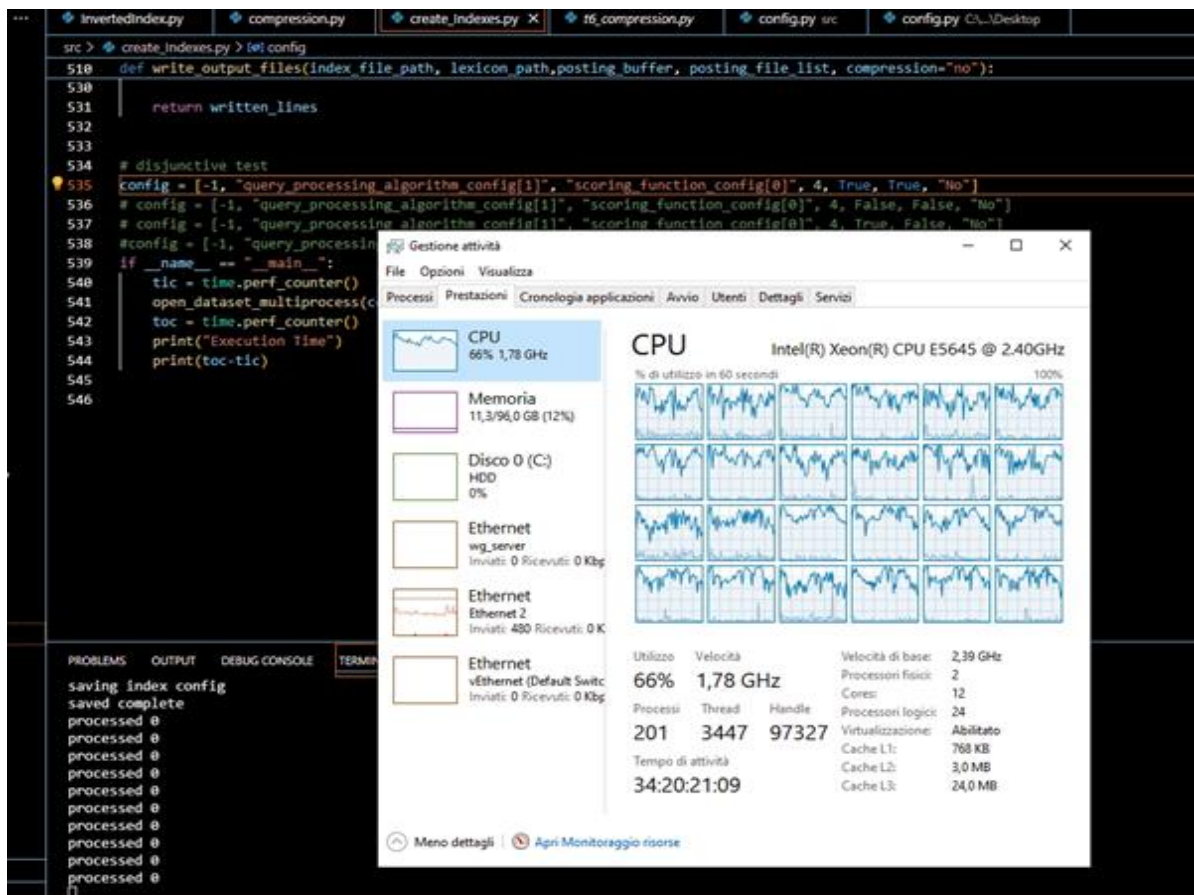
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
saving index config
merged all chunks
saving index config
processed 460000
processed 460000
merged all chunks
saving index config
merged all chunks
saving index config
Execution Time
170298.4667752001

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
saving index config
merged all chunks
saving index config
merged all chunks
saving index config
merged all chunks
saving index config
merged all chunks
saving index config
Execution Time
773519.8878968998
C:\Users\MIRCV\Desktop\MIRCV-Project-2024\src>
```





Index name				computer	Index ID	index.txt Size (MB)	Indexing time		
Operation	compression	stopword	stemming				(seconds)	(minutes)	(hours)
full indexing	uncompressed	keep	no	2 x Intel Xeon E5645 @ 2.40GHz 12 Cores	0	87*20 (20 index file multproc)	788338.55	13138.97583	218.9829306
full indexing	uncompressed	no	no	2 x Intel Xeon E5645 @ 2.40GHz 12 Cores	1	62*20 (20 index file multproc)	170298.466	2838.307767	47.30512944
full indexing	uncompressed	keep	yes	2 x Intel Xeon E5645 @ 2.40GHz 12 Cores	2	82*20 (20 index file multproc)	773519.88	12891.998	214.8666333
full indexing	uncompressed	no	yes	2 x Intel Xeon E5645 @ 2.40GHz 12 Cores	3	58*20 (20 index file multproc)	136666.53	2277.7755	37.962925
MERGE	no	keep	no	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	0	1738	1322.7927	22.046545	0.3674424167
MERGE	unary	keep	no	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	0	unfeasible size for HDD	FAIL	0	0
MERGE	gamma	keep	no	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	0	479	1,517	25.28347	0.4213911667
MERGE	no	no	no	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	1	1246	1506.3237	25.105395	0.41842325
MERGE	gamma	no	no	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	1	395	1481.3999	24.6899833	0.4114999722
MERGE	no	keep	yes	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	2	1652	1,005	16.75692167	0.2792820278
MERGE	gamma	keep	yes	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	2	430	1,226	20.42995667	0.3404992778
MERGE	no	no	yes	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	3	1163	993.005	16.55008333	0.2758347222
MERGE	gamma	no	yes	AMD Ryzen 5 5600x @ 4.6 GHz 6 Cores	3	347	1221.5123	20.35853833	0.3393089722



## 5 – Results

We performed two types of evaluation for the generation and comparison of the results. First we select 10 queries from the TREC eval on the index named "index\_main" and execute them with different configurations through the CLI interface. "index\_main" is the index created with the whole collection, removing the stopwords and applying the Porter Stemmer.

The results as shown in the table below:

Trec	Query Id	Query Text	Exec Time	k	stemming	stop word	scoring	Search algorithm	query token junction
2020	132622	definition of attempted arson	11,1087	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	999466	where is velbert	0,0036	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	997622	where is the show shameless filmed	236,7636	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	99005	convert sq meter to sq inch	82,6481	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	985594	where is kampuchea	0,0335232	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	978031	where is berlin center	125,0765705	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	945835	when does ace hardware open?	140,058676	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	940547	when did rock n roll begin?	429,4267998	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	938400	when did family feud come out?	117,5031256	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	918162	what was darwin's greatest contribution to evolutionary theory?	14,3921494	20	skip: false	allow: false	BM11	ternary	disjunctive
2020	132622	definition of attempted arson	0,044779	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	999466	where is velbert	0,0033964	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	997622	where is the show shameless filmed	0,5164673	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	99005	convert sq meter to sq inch	0,2575236	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	985594	where is kampuchea	0,0302593	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	978031	where is berlin center	0,4579587	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	945835	when does ace hardware open?	0,2727416	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	940547	when did rock n roll begin?	0,6867461	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	938400	when did family feud come out?	0,2301813	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	918162	what was darwin's greatest contribution to evolutionary theory?	0,0530176	20	skip: false	allow: false	BM25	ternary	conjunctive
2020	132622	definition of attempted arson	1,3293	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	999466	where is velbert	0,1751	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	997622	where is the show shameless filmed	145,9705	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	99005	convert sq meter to sq inch	37,4828	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	985594	where is kampuchea	0,046384	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	978031	where is berlin center	62,4501537	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	945835	when does ace hardware open?	76,3879771	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	940547	when did rock n roll begin?	309,8478072	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	938400	when did family feud come out?	55,5112045	20	skip: false	allow: false	BM15	skipping	disjunctive
2020	918162	what was darwin's greatest contribution to evolutionary theory?	2,5738406	20	skip: false	allow: false	BM15	skipping	disjunctive

The results obtained generally show that with respect to execution time the conjunctive queries turn out to be much faster than the disjunctive ones as we expected, however, the choice of search algorithm also affects, in the case of skipping, it affects positively with respect to execution time.

While for the same scoring method the variation on execution time is minimal.

On the other hand, for the second type of evaluation we used the evaluate\_indexes module we implemented that uses the TREC 2019 and 2020 framework by going to test the indexes we generated from the MSMARCO collection.

For execution, we opted for a configuration with a Conjunctive retrieval model using TFIDF and set `topk=20`, applying it to both the TREC 2019 and 2020 datasets.

In the output data, we observed matches between the pairs of query-document results generated by our IR system and those labeled as relevant in the provided dataset. However, our analysis of the 2019 queries, run in conjunctive mode, revealed that while the system

identified a set of relevant documents for each analyzed query-document pair, these documents did not rank within the top `k` results in the assessment qrels. As a result, they could not be evaluated with TREC's metrics, limiting the metric-based analysis of our system's effectiveness.

*This is the link for the [drive folder](#) that contains the whole indexes and the TREC eval results.*

## 6 - Conclusions and Possible Improvements

The project focused on building an inverted index for the MSMARCO Passage Ranking dataset, illustrating effective methods for handling large-scale information retrieval.

This system effectively processes user queries through several steps: data preprocessing, efficient indexing, and query handling, using both classical and modern techniques to ensure high performance.

Key achievements include a functional inverted index capable of processing complex queries, leveraging relevance models like BM25 and TF-IDF to rank results effectively.

Despite its strengths, the system has limitations in scalability and handling complex queries. Future improvements could include distributed indexing to scale across multiple machines, supporting semantic search for better query understanding, enabling real-time indexing for dynamic data environments, and developing a user-friendly interface.

As information security becomes increasingly important, adding also features such as encrypted indexing, secure query processing, and robust access controls could be vital for deploying the program in sensitive or regulated environments.

In sum, the project successfully demonstrates the power of classical indexing methods on a large dataset and lays a foundation for future enhancements, contributing valuable insights and techniques that can inform the ongoing development of efficient search systems in the field of information retrieval.