

CS6913 Web Search Engines Homework 2

Dong Li
dl5214@nyu.edu

Oct 2024

Abstract

This report presents the design, implementation, and performance evaluation of a search engine system for processing large datasets across multiple query modes. Key features include efficient index construction, data compression, top-K MaxScore optimization, and a user-friendly web interface. The system is evaluated based on running time, I/O efficiency, and query performance, demonstrating its ability to retrieve relevant documents effectively while delivering a smooth user experience.

Keywords: Search Engine, Inverted Index, Data Compression, Query Optimization, Web-Based Interface

1 Introduction

The objective of this project is to develop an efficient search engine capable of handling large-scale datasets, such as the MS MARCO passage-ranking dataset. The system is required to support the Document-At-A-Time (DAAT) query processing method while maintaining I/O efficiency. The system uses a compressed inverted index structure, which stores document IDs and term frequencies in a compact form. The BM25 ranking algorithm ensures that relevant documents are retrieved for both conjunctive (AND) and disjunctive (OR) queries.

The project consists of three main components: index construction, merging intermediate results, and query processing. The index is compressed using delta encoding and varbyte encoding to optimize storage. Furthermore, a web-based interface and the Top-K MaxScore algorithm [1, 2] for disjunctive queries were integrated to enhance query performance and improve the user experience. This report presents a detailed overview of the system's architecture, workflow, and performance evaluation, followed by a discussion of its strengths and limitations observed during testing and development.

2 Workflow, Configuration, Output and Performance

2.1 Workflow

The process is divided into two main phases: index building and query processing.

2.1.1 Index Building

The index building phase produces three key output files: the final compressed inverted index, the lexicon, and the page table. These files are generated over three steps.

The lexicon stores each word along with its start and end positions in the inverted index, the number of documents containing the word (`docNum`), and the number of blocks (`blockNum`).

The page table contains information for each document, such as the document ID (`docID`), data length, word count, and the position offset (`docPos`) in the dataset.

Step 1: Intermediate Inverted List and Page Table Construction The first step involves reading the dataset and building intermediate inverted lists while simultaneously constructing the page table, as the dataset is read sequentially. The word list is already sorted by red-black tree at this stage. When parsing the data, all words are normalized to lower case.

Step 2: Merging Intermediate List without Compression Next, the intermediate inverted lists are merged into a single index file without compression. This step involves a multi-way merge (in this case, a 178-way merge), using a priority queue to efficiently merge the postings lists. Postings are combined when the same word appears in different files, taking advantage of the pre-ordered document IDs.

The output at this stage looks likes:

word1: docID freq, docID freq, ...

word2: docID freq, docID freq, ...

Step 3: Final Index Compression and Lexicon Construction In the final step, compression is applied to produce the final inverted index. Document IDs are delta encoded, and both document IDs and frequencies are varbyte encoded. The lexicon is built during this step, recording metadata such as the positions and sizes of postings lists.

2.1.2 Output Overview

- Number of intermediate inverted list files: 178
- Page table size: 140.3 MB (without snippet pointers), 234.3 MB (with snippet pointers)
- Total documents in page table: 8,841,824
- Merged inverted list (uncompressed): 3.53 GB
- Final compressed inverted index: 928.1 MB
- Lexicon size: 48.7 MB
- Number of words in lexicon: 1,469,232

2.1.3 Performance

- **Step 1:** Parsing data, building intermediate inverted lists, and writing the page table took 1621.62 seconds. Writing the page table to disk (I/O time) took an additional 16.95 seconds.
- **Step 2:** Merging the 178 intermediate lists into a single file (178-way merge) took 1707.48 seconds.
- **Step 3:** Building the lexicon and the final compressed index took 731.31 seconds.

Total time for index building:

Approximately 1 hour and 8 minutes, broken down as:

- Step 1: 27 mins
- Step 2: 28.5 mins
- Step 3: 12.2 mins

2.2 Query Processing

The query processing phase utilizes the **BM25** ranking algorithm to compute document relevance. The expected output includes the document ID, BM25 score, and the original text content of each relevant document.

The first step in query processing is loading the page table and lexicon into main memory. Once loaded, the query loop listens for user input, which can be provided either through the command line or via the web-based frontend.

2.2.1 Search Modes and Query Types

There are two primary searching modes:

- **Term-At-A-Time (TAAT):** In this mode, the query terms are processed one at a time.
- **Document-At-A-Time (DAAT):** In DAAT, the system processes all query terms simultaneously, scanning through document lists at the same time.

The system supports two query types:

- **Conjunctive (AND) Queries:** These return documents that contain all query terms.
- **Disjunctive (OR) Queries:** These return documents that contain at least one of the query terms.

Upon receiving a query, the system splits it into individual words (referred to as **wordList**) and performs either TAAT or DAAT processing:

- **TAAT Conjunctive:** This mode uses a hash map to accumulate results for documents that contain all terms and remove terms not appearing in every word in wordList.
- **TAAT Disjunctive:** A vector list is used to implement an algorithm similar to counting sort, processing documents that contain any of the query terms.
- **DAAT Conjunctive:** This mode uses a binary-search-based `nextGEQ()` function to perform zigzag searching across document lists.
- **DAAT Disjunctive:** The top-K MaxScore algorithm is employed to efficiently retrieve the top-ranking documents.

Implementation details and data structures are discussed in Part 3.

2.2.2 Performance

It takes approximately 10.85 seconds to load the page table and lexicon into memory, preparing the system for query processing. Sample query execution times for the query 'dog cat' are as follows:

- **TAAT Conjunctive Query:** 0.10 seconds
- **TAAT Disjunctive Query:** 1.90 seconds
- **DAAT Conjunctive Query:** 0.07 seconds
- **DAAT Disjunctive Query (with Top-K MaxScore):** 0.22 seconds

2.3 Configuration

2.3.1 Controlling Flags

The configuration of the program can be controlled using various flags in the `config.h` file. These flags control both the index building and query processing phases of the program. The table below lists the configuration options:

Flag Name	Description
INDEX_BUFFER_SIZE	The buffer size used for reading files, default is 10 MB.
POST_BYTES	Estimated size of posting entries in bytes, default is 10.
AVG_WORD_BYTES	Estimated average number of bytes per word, default is 12.
INDEX_CHUNK_SIZE	Size of each chunk during indexing, default is 20 MB.
POSTINGS_PER_CHUNK	The number of postings stored per block, default is 64.
BLOCK_SIZE	Block size for writing the inverted index, default is 64 KB.
MAX_META_SIZE	Maximum metadata size per block, default is 8 KB.
MAX_DOC_ID	The maximum document ID allowed, default is -1 (unsigned).
FILE_MODE_BIN	File mode, set to 1 for binary format, 0 for ASCII format.
DAAT_FLAG	Search mode: 0 for TAAT, 1 for DAAT.
NUM_TOP_RESULT	Number of top results to return in query output, default is 15.
DEBUG_MODE	Set to 1 to provide additional output during execution.
PARSE_INDEX_FLAG	Set to 1 to build the intermediate inverted list.
PAGE_TABLE_FLAG	Set to 1 to build the page table and write to disk.
MERGE_FLAG	Set to 1 to merge the intermediate indexes into one file.
LEXICON_FLAG	Set to 1 to build and write the lexicon structure.
DELETE_INTERMEDIATE	Set to 1 to delete intermediate index files after merging.
LOAD_FLAG	Set to 1 to load the page table and lexicon into main memory.
QUERY_FLAG	Set to 1 to enable query processing.
FRONTEND_FLAG	1: Flask web interface, 0: command line mode.

Table 1: Configuration flags in `config.h`.

2.3.2 How to Run

The program can be run by adjusting the flags as follows:

- **Index Building Phase:**

- Set `PARSE_INDEX_FLAG` and `PAGE_TABLE_FLAG` to 1 to parse the original dataset and build the intermediate inverted list and page table.
- Set `MERGE_FLAG` to 1 to merge the intermediate index files into a single non-compressed file.
- Set `LEXICON_FLAG` to 1 to build the final compressed index and lexicon.

- **Query Processing Phase:**

- Set `LOAD_FLAG` to 1 to load the page table and lexicon into memory.
- Set `DAAT_FLAG` to 0 for TAAT or 1 for DAAT, based on the preferred search mode.
- Set `QUERY_FLAG` to 1 and `FRONTEND_FLAG` to 0 for command line queries.
- Set `QUERY_FLAG` to 1 and `FRONTEND_FLAG` to 1 to use the web-based frontend. C++ Boost runs on port 9001 and Python Flask runs on port 5001. To access through a web browser, first run C++, then Python Flask, and access 127.0.0.1:5001 using a web browser.

3 Data Structure and Algorithm

The system is divided into several core modules, each responsible for a specific aspect of the search engine workflow. These modules include the *IndexBuilder*, *QueryProcessor*, and a web-based frontend.

3.1 IndexBuilder Module

The *IndexBuilder* module is responsible for building the search index, creating the lexicon structure, and managing the document metadata. It consists of several sub-components, including the *PageTable*, *InvertedList*, and *Lexicon* classes.

3.1.1 PageTable

The **PageTable** is used to store metadata for each document in the collection. Each document entry in the table contains:

- **Document ID (docID)**: A unique identifier for each document.
- **Data Length (dataLength)**: The size of the document, measured in terms of the number of bytes or words.
- **Word Count (wordCount)**: The number of distinct words present in the document.
- **Document Position (docPos)**: The offset to access the docID in the original data set.

The structure of the page table is represented as follows:

$$\{docID_0, dataLen_0, wordCount_0, docPos_0\}, \{docID_1, dataLen_1, wordCount_1, docPos_1\}, \dots$$

3.1.2 Inverted Index

The **Inverted Index** manages the inverted index, which stores the mapping between terms (words) and the documents in which they appear. Each term is associated with a list of postings, where each posting contains:

- **Document ID (docID)**: The ID of a document where the term appears.
- **Frequency (freq)**: The frequency of the term in the corresponding document.

I used a class *SortedPosting* with map (red-black tree) to sort the postings in any intermediate documents in alphabetical order. Plus, owing to the parsing sequence, for the same word, the doc ID in larger intermediate index is also larger, so the merging sort is accelerated.

In the temporary stage (including the stage after merging), the inverted index is implemented using an ordered map, where:

$$word_0 : \{docID_0, freq_0\}, \{docID_1, freq_1\}, \dots, \{docID_{n-1}, freq_{n-1}\}$$

$$word_1 : \{docID_0, freq_0\}, \{docID_1, freq_1\}, \dots, \{docID_{n'-1}, freq_{n'-1}\}$$

$$\vdots$$

After merging the temporary indexes, the final encoded and compressed index is written into blocks.

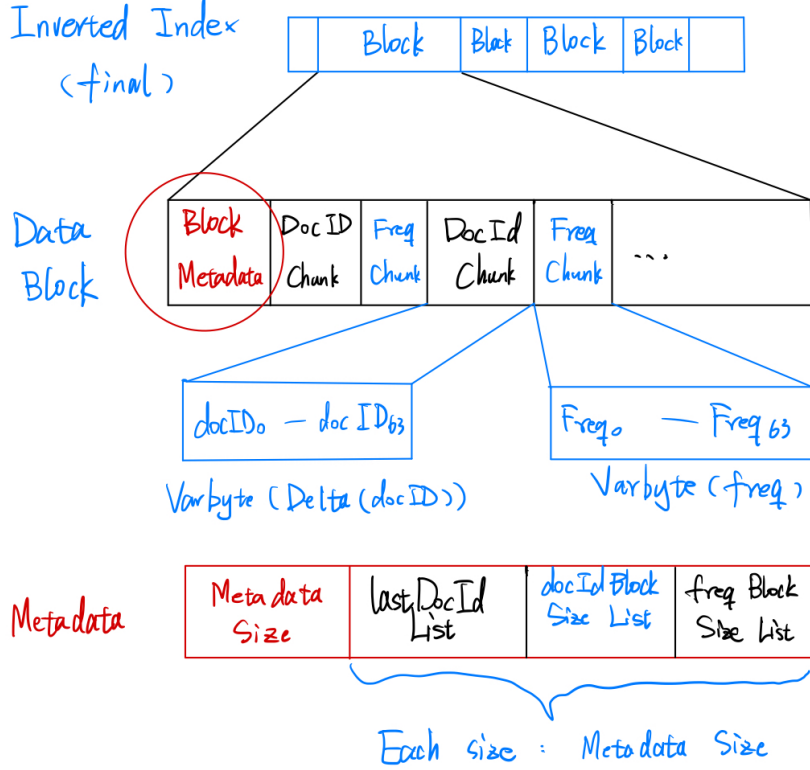


Figure 1: Structure of the Inverted Index

The final inverted index is compressed and organized in a block-based structure, as illustrated in Figure 1. Each term has its own inverted list, which is divided across one or more blocks. Each block contains either a full or partial inverted list for a single term.

Each block is **no larger** than 64 KB and consists of the following components:

- **Metadata:** This section is uncompressed and contains four parts:
 - **MetadataSize:** Records the number of entries in the other three lists in this block.
 - **lastDocIdList:** Stores the last docID from each docID chunk within the block.
 - **docIdSizeList:** Records the size of each docID chunk in the block.
 - **freqSizeList:** Records the size of each frequency chunk in the block.

The length of `last_docID_list`, `docID_size_list`, and `freq_size_list` is the same, which is the value of `Metadata_size`.

- **DocID Chunks:** Each chunk contains 64 document IDs. The first document ID is stored as-is, while subsequent document IDs are stored as the delta (difference) relative to the previous one.
- **Frequency (Freq) Chunks:** Each chunk contains 64 frequencies corresponding to the document IDs in the same chunk.

VarByte encoding is used to compress both the document ID and frequency chunks, resulting in variable-sized chunks depending on the content being encoded.

3.1.3 Lexicon

The **Lexicon** serves as a directory for the inverted index. It stores metadata about the position and size of each term's postings list. Specifically, for each term, the following information is recorded:

- **Begin Position (beginPos):** The offset in the index where the term's postings list starts.
- **End Position (endPos):** The offset where the postings list ends.
- **Document Count (docNum):** The number of documents that contain the term.
- **Block Count (blockNum):** The number of blocks required to store the term's postings list.

The *Lexicon* structure is built during the final index generation process. Each term's metadata is inserted into the lexicon using the `Lexicon::insert` function, and the term's start and end positions in the index are recorded using the `Lexicon::build` function. The lexicon structure looks like:

$$\begin{aligned}
 word_0 &: \{beginPos_0, endPos_0, docNum_0, blockNum_0\} \\
 word_1 &: \{beginPos_1, endPos_1, docNum_1, blockNum_1\} \\
 &\vdots
 \end{aligned}$$

3.2 QueryProcessor Module

The *QueryProcessor* is responsible for processing user queries and retrieving relevant documents from the inverted index. It supports two query types: *Conjunctive (AND)* and *Disjunctive (OR)*. The module is optimized for I/O efficiency, ensuring minimal memory usage and fast query response times.

3.2.1 Loading Metadata, Open List and Decoding

Before processing any query, the **QueryProcessor** loads the **PageTable** and **Lexicon** into main memory. Once the query terms are identified, the **QueryProcessor** opens the relevant postings lists from the inverted index using the metadata provided by the lexicon and page table. Basic decoding operations for delta encoding and VarByte encoding are applied to restore the original document IDs and frequencies.

3.2.2 Query Modes, Types and Implementation

For **Term-At-A-Time (TAAT)** queries, the data is directly operated on at a binary level, meaning that the BM25 score is calculated simultaneously when decoding the data. For **conjunctive** (AND) queries:

- A hash map is built based on the term with the fewest documents (`minTerm`).
- For each additional term, the hash map is updated by checking whether a document exists in both the `minTerm` and the current term.
- If a match is found, the score is updated. If no match is found, the document is removed from the hash map due to the conjunctive requirement.

For **disjunctive** (OR) queries in TAAT mode:

- A dynamic array (vector) of size equal to the number of documents is used, similar to counting sort, instead of using a hash map (which is costly).
- For each term, the array is updated by incrementing the value at the index corresponding to the document ID.
- A min-heap is used to maintain and retrieve the top-K results. A `DocScoreEntry` class is defined, with a custom comparator to rank entries by BM25 scores.

For **Document-At-A-Time (DAAT)** queries, the document ID lists are decoded first. For **conjunctive** queries:

- A binary-search-based `nextGEQ()` function is used to perform a zigzag search, which quickly finds the next greater-than-or-equal document ID in all postings lists.

For **disjunctive** queries in DAAT mode, the **Top-K MaxScore algorithm** [1] is used to optimize performance via early termination:

- Since the document ID list for each term is decoded in advance, a score list is created for each term, making the maximum score for each term known.
- Based on the maximum score, essential and non-essential terms are identified, allowing the algorithm to avoid fully computing the disjunctive union. This allows it to focus on acquiring the top-K results at a lower cost.
- For example, when querying "dog cat", TAAT mode took 1.90 seconds, while DAAT mode with the Top-K MaxScore algorithm completed in 0.22 seconds.

3.3 Encoding and Decoding Scheme

In the system, encoding occurs during the index building phase, and decoding takes place during query processing. The following details outline how Delta and VarByte encoding are applied, as well as how metadata facilitates efficient retrieval.

3.3.1 Encoding during Index Building

During index building, document IDs and term frequencies are encoded using a combination of Delta encoding and VarByte encoding:

- **Delta Encoding for DocIDs:** Instead of storing absolute document IDs, the difference (delta) between consecutive docIDs is stored. This reduces the storage size as smaller deltas can be represented with fewer bytes.
- **VarByte Encoding:** Each document ID and frequency is encoded using VarByte encoding, which uses the least significant 7 bits of each byte for data, while the most significant bit (MSB) indicates if more bytes follow.

The encoded data is then divided into chunks, each containing up to 64 document IDs and their corresponding frequencies. Metadata for each block (containing multiple docID and freq chunks) is also stored, including:

- **Metadata Size:** The number of postings (document IDs and frequencies) stored in the block.
- **Last DocID in the Block:** This helps in skipping blocks during query processing by allowing quick determination if a block can be skipped when searching for a docID greater than a certain value.
- **DocID and Frequency Block Sizes:** The sizes of the encoded document IDs and frequencies, which enable precise positioning within the block for efficient decoding.

3.3.2 Decoding during Query Processing

When a query is processed, the system leverages the metadata stored with the index to efficiently decode postings lists:

- **Lexicon Lookup:** For each term in the query, the lexicon provides the starting and ending positions of the term's postings list in the index, as well as the number of documents and blocks associated with the term.
- **Block Metadata:** The system reads block metadata, such as the last document ID and the sizes of the docID and frequency chunks, to locate and decode the relevant parts of the postings list. If the query term requires document IDs larger than a certain value, the system can use the 'lastDocId' to skip over blocks where no relevant documents exist.
- **VarByte and Delta Decoding:** VarByte decoding is applied to retrieve the original document IDs, and delta decoding restores the absolute document IDs by summing the deltas. Frequencies are decoded directly from VarByte-encoded data.

3.3.3 Content Retrieval

After decoding the relevant document IDs, the system uses the **PageTable**, which provides the document’s metadata, including the **docPos** (document position). The **docPos** allows the system to efficiently locate the content of the document in the large collection data set without loading the entire dataset into memory. By seeking directly to the **docPos**, the content associated with the decoded **docID** is retrieved and included as part of the search result, ensuring I/O-efficient content retrieval.

3.4 Frontend Module

The frontend module provides a web-based interface for users to interact with the search engine. Built using the **Flask** framework in Python, it handles user inputs and displays search results. The backend, written in C++, communicates with Flask via the **Boost** library over a socket connection on port 9001.

3.4.1 C++ and Python Communication

Communication between the C++ backend and Flask is managed as follows:

- The C++ backend runs a server on port 9001, listening for incoming connections from Flask.
- Flask sends the user’s query to the backend via a socket.
- The backend processes the query with the **QueryProcessor** module and returns the results.
- Flask formats and displays the results to the user.

3.4.2 Web Interface

The web interface, built with HTML and CSS, manages user interactions:

- **search.html**: The page where users input search queries.
- **result.html**: The page that displays search results.
- **Flask Server**: Manages communication with the C++ backend and runs on port 5001, accessible via a web browser.

This design ensures efficient communication, enabling fast, real-time query processing and result display.

3.5 Important Functions

The backend of the search engine system consists of two core modules, *IndexBuilder* and *QueryProcessor*, responsible for index construction and query processing. Below is a summary of the key functions and their roles within the system.

3.5.1 IndexBuilder Module

- **readData()**: Reads and processes the raw dataset, calculates word frequencies, and builds the intermediate inverted list, where the (lower-case) normalized postings are sorted using a red-black tree. It also creates a page table for document metadata and writes it to disk.
- **mergeIndex()**: Merges intermediate posting lists by comparing document IDs and frequencies, ensuring no data loss. If the intermediate files are already sorted, the merging process can be accelerated by popping the whole docID list of a posting instead of one-by-one.
- **buildLexicon()**: Constructs the final compressed index during the last stage of the indexing process. This function calls `Lexicon::build()` to encode document IDs and frequencies using Delta and VarByte encoding, and splitting postings into blocks. At the same time, the lexicon is created, recording the start/end positions, document count, and block count for each term's postings, enabling efficient data retrieval during query processing.

3.5.2 QueryProcessor Module

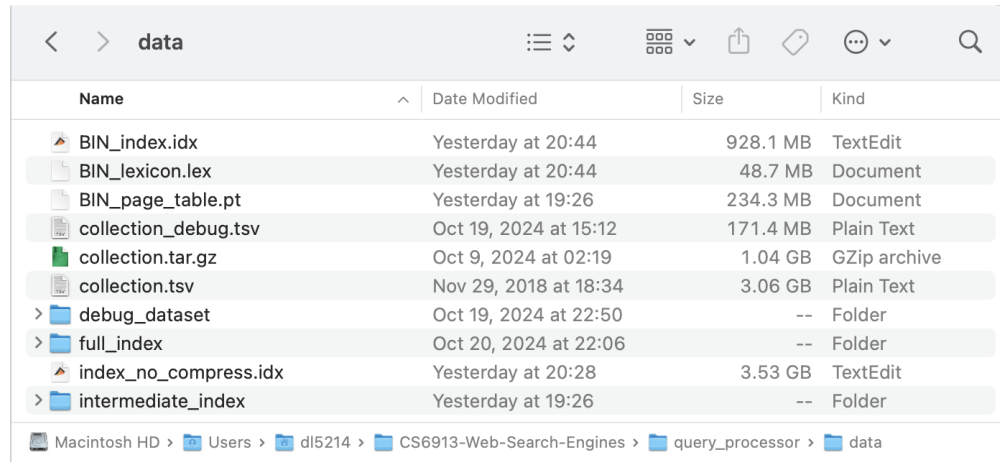
- **_queryTAAT()**: Handles Term-At-A-Time (TAAT) queries. For conjunctive (AND) queries, the function starts by finding the term with the fewest documents (`minTerm`) and building a hash map with BM25 scores by calling `_decodeBlocksToMap()`. It then calls `_updateScoreMap()` for other terms in the query to check for collisions with the initial term's docIDs. If a collision is found, it updates the score; otherwise, it removes the docID from the map. After processing all terms, `_getMapTopK()` is used to retrieve the top-k documents using a min-heap based on BM25 scores. For disjunctive (OR) queries, the function updates a score array for each term in the query by calling `_decodeBlocksToList()`, and `_getListTopK()` retrieves the top-k results. It is noticeable that in TAAT mode, scores are updated while the index is being decompressed.
- **_queryDAAT()**: Processes Document-At-A-Time (DAAT) queries. For conjunctive (AND) queries, it calls `_decodeBlocks()` to decompress the index and retrieve docID lists for each term, using binary-search-based `_nextGEQ()` to accelerate docIDs matching. For disjunctive (OR) queries, it also retrieves docID lists by calling `_decodeBlocks()`, calculates the BM25 scores for each term in the query, and therefore the max score of each term is known. Afterwards, it calls the helper function `_maxScoreTopK()` to apply The Top-K MaxScore algorithm to optimize disjunctive queries performance.

4 Sample Results

This section presents sample results from the system, including screenshots that demonstrate the running time, query results through the command line interface, and query results through the web browser interface.

4.1 File Size and Running Time

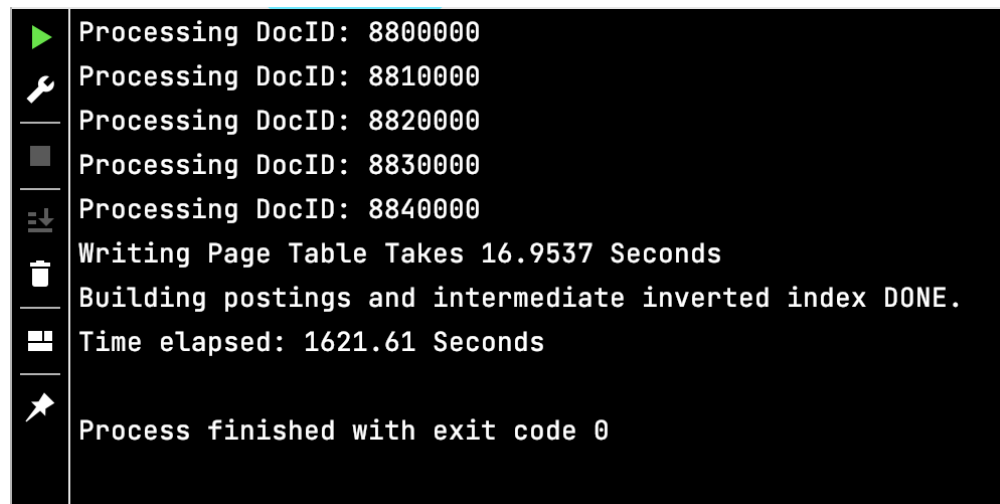
Figure 2 shows the size of the inverted index, lexicon and page table.



Name	Date Modified	Size	Kind
BIN_index.idx	Yesterday at 20:44	928.1 MB	TextEdit
BIN_lexicon.lex	Yesterday at 20:44	48.7 MB	Document
BIN_page_table.pt	Yesterday at 19:26	234.3 MB	Document
collection_debug.tsv	Oct 19, 2024 at 15:12	171.4 MB	Plain Text
collection.tar.gz	Oct 9, 2024 at 02:19	1.04 GB	GZip archive
collection.tsv	Nov 29, 2018 at 18:34	3.06 GB	Plain Text
debug_dataset	Oct 19, 2024 at 22:50	--	Folder
full_index	Oct 20, 2024 at 22:06	--	Folder
index_no_compress.idx	Yesterday at 20:28	3.53 GB	TextEdit
intermediate_index	Yesterday at 19:26	--	Folder

Figure 2: Sizes of compressed Inverted Index, Lexicon and Page Table.

Figures 3 to 6 show the system's running time during various stages of execution, including index building and query processing.



```
Processing DocID: 8800000
Processing DocID: 8810000
Processing DocID: 8820000
Processing DocID: 8830000
Processing DocID: 8840000
Writing Page Table Takes 16.9537 Seconds
Building postings and intermediate inverted index DONE.
Time elapsed: 1621.61 Seconds
Process finished with exit code 0
```

Figure 3: Running time for parsing and page table writing.

```
▶ /Users/d15214/CS6913-Web-Search-Engines/query_processor/cmake-build-debug/Main
Merging inverted index. Timing started...
Number of intermediate index files: 178
Merging inverted index DONE.
Time elapsed: 1707.48 Seconds
Process finished with exit code 0
```

Figure 4: Running time for merging intermediate index.

```
▶ you 918932614 923296611 1956263 67
young 923305971 923468029 62233 3
your 923552926 926821461 1465254 50
yourself 926854832 926971313 43046 2
zip 927677293 927751816 28001 2
zone 927845001 927932303 32749 2
Building Lexicon and Final Index DONE.
Time elapsed: 731.31 Seconds
Process finished with exit code 0
```

Figure 5: Running time for lexicon building and compression.

```
read lexiconItem: x 915077213 915286146 82969 4
read lexiconItem: y 915651692 915742536 33523 2
read lexiconItem: z 927217139 927259697 15087 1
There are 1469232 words in Lexicon Structure
Loading PageTable and Lexicon Done.
Time elapsed: 10.85 Seconds
Console mode: please use query loop in console...
Welcome to CS6913 Web Search Engine!
input 'exit' to exit.
Please input a query, e.g. 'hello world'.
query>>
```

Figure 6: Running time for loading page table and lexicon.

4.2 Query through Command Line Interface

Figures 7 to 10 display query results obtained via the command line interface.

```
query>>dog cat
conjunctive (0) or disjunctive (1)>>0
search using 0.10s
Here are the Top 15 results:
1: 18.93 7649252
7649252 Let your dog, if you have one, out at the same time as the cat. Your dog is probably bonded with your cat, but not with the bully
ch larger than cats, seeing a dog around should prove intimidating to the bu

2: 18.86 422786
422786 Veterinarians; Services; Know Your Pet. Cat Breeds; Cat Care; Dog Breeds; Dog Care; Con

3: 18.79 2033837
2033837 The same type of DNA markers, single nucleotide polymorphisms (SNPs) are used for the cat test as for the dog tests, however the c
cific to the cat and will not cross-react with the dog. The Cat Ancestry will not work on the dog, the dog test will not work for the cat.
```

Figure 7: Query 'dog cat' in TAAT CONJ mode via command line.

```
query>> dog cat
conjunctive (0) or disjunctive (1)>>0
search using 0.07s
Here are the Top 15 results:
1: 17.75 422786
422786 Veterinarians; Services; Know Your Pet. Cat Breeds; Cat Care; Dog Breeds; Dog Care; Con

2: 17.52 8743849
8743849 Mon animal de compagnie. But French usually say what the pet is. My dog - Mon chien My cat, mon c

3: 17.52 6854171
6854171 Dog Water Dispenser,Gravity Feeder,Automatic Dog Waterer,Pet Waterer Feeder Set for Cat Dog By OL
```

Figure 8: Query 'dog cat' in DAAT CONJ mode via command line.

```
query>>dog cat
conjunctive (0) or disjunctive (1)>>1
search using 1.90s
Here are the Top 15 results:
1: 18.93 7649252
7649252 Let your dog, if you have one, out at the same time as the cat. Your dog is probably bonded with your cat, bu
ch larger than cats, seeing a dog around should prove intimidating to the bu

2: 18.86 422786
422786 Veterinarians; Services; Know Your Pet. Cat Breeds; Cat Care; Dog Breeds; Dog Care; Con

3: 18.79 2033837
2033837 The same type of DNA markers, single nucleotide polymorphisms (SNPs) are used for the cat test as for the dog
cific to the cat and will not cross-react with the dog. The Cat Ancestry will not work on the dog, the dog test will
```

Figure 9: Query 'dog cat' in TAAT DISJ mode via command line.

```

query>>dog cat
conjunctive (0) or disjunctive (1)>>1
search using 0.22s
Here are the Top 15 results:
1: 18.93 7649252
7649252 Let your dog, if you have one, out at the same time as the cat. Your dog is probably bonded with your cat, but not w
ch larger than cats, seeing a dog around should prove intimidating to the bu

2: 18.86 422786
422786 Veterinarians; Services; Know Your Pet. Cat Breeds; Cat Care; Dog Breeds; Dog Care; Con

3: 18.79 2033837
2033837 The same type of DNA markers, single nucleotide polymorphisms (SNPs) are used for the cat test as for the dog tests,

```

Figure 10: Query 'dog cat' in DAAT DISJ mode via command line.

4.3 Query through Web Browser Interface

Figure 11 is the home page of the web-based interface.

Search Engine

Enter your query here

☐ Conjunctive (AND)
 ☒ Disjunctive (OR)

Search

Figure 11: Web Interface for the Query System.

Figures 12 and 13 show query results using the web-based interface.

Search Results
Your query: dog day afternoon
Mode: Conjunctive (AND)

Document ID	BM25 Score	Content
2210371	18.3989	Content: 2210371 This Site Might Help You. RE: What Happened to Al Pacino's Voice? For those who notice it Do you notice that somewhere between the first Godfather movies Dog Day Afternoon And the later movies Scarface Scent of a Woman that there's a change in Al Pacino's voice? It became rougher..What's the story behi
102528	18.417	Content: 102528 Describes how the film impacted the gangster genre and the trend of heavy violence in such films. Trends and Films that have influenced the creation and development of Scarface would be Taxi Driver The God Father Part I and II Mean Streets Dog Day Afternoon Bonnie and Clyde and the gangster movie
7089630	18.4352	Content: 7089630 Sidney Lumetâ€™s first film was 1957â€™s 12 Angry Men. He made 20 more between that and Dog Day Afternoon (and 22 more afterward) and by his own account he never used improv. â€œI donâ€™t like actors to improvise to use their own language â€œ he said in the Dog Day Afternoon DVD com

Back to search

Figure 12: Query 'dog day afternoon' in DAAT CONJ mode via web interface.

Search Results
Your query: dog day afternoon
Mode: Disjunctive (OR)

Document ID	BM25 Score	Content
6443245	16.6916	Content: 6443245 Full Definition of AFTERNOON. 1. : the part of day between noon and sunset. 2. â€œ afternoon adjective. See afternoon defined for English-language learners. See afternoon defined for kids. ADVERTISEMENT.ull Definition of AFTERNOON. 1. : the part of day between noon and sunset. 2. â€œ afternoon adjective. See afternoon defined for English-language learners. See afternoon defined for kids. ADVERT
6443247	16.763	Content: 6443247 Definition of AFTERNOON for Kids. : the part of the day between noon and evening.ull Definition of AFTERNOON. 1. : the part of day between noon and sunset. 2. â€œ afternoon adjective. See afternoon defined for English-language learners. See afternoon defined for kids. ADVERT
7089632	16.8517	Content: 7089632 YouTube. 1 In 1972 a Brooklyn bank robbery intended to fund a sex-change operation turned into a day-long standoff. Three years later Sidney Lumet turned that strange story into Dog Day Afternoon a lively intense and surprisingly funny crime film featuring one of Al Pacinoâ€™s best perfo

Back to search

Figure 13: Query 'dog day afternoon' in DAAT DISJ mode via web interface.

5 Discussion

5.1 Advantages and Fancy Features

The results from both TAAT and DAAT modes are consistent in terms of ranking, docID, and calculated scores, which confirms the index structure is correctly implemented and the query process is functioning efficiently. This ensures that no critical components are missing from the system.

Additionally, beyond the core requirements, I have implemented several enhancements:

- **Web interface:** This provides a better user experience by allowing queries to be made through a browser.
- **Top-K max score algorithm:** Improves the performance of DAAT disjunctive queries by early termination.
- **Support for both TAAT and DAAT:** Both query processing methods are implemented, and their consistency has been verified.

5.2 Limitations

There are some limitations observed during testing. During the demo, the TA noted that, for the same query, the score returned in disjunctive mode was lower than that in conjunctive mode, which is not expected. Although I confirmed consistency between TAAT and DAAT modes, this suggests there may be an issue in the scoring logic, specifically within the `QueryProcessor::getBM25()` function.

Additionally, during the demo time, I only displayed docIDs and scores, with the original dataset open for manual verification. While the assignment stated that snippet generation was optional, relying solely on docIDs does not meet typical user expectations for a search system, where users expect to see relevant content alongside their query terms. In this report, I have revised the page table structure by introducing a `docPos` feature, now enabling the system to display the original content along with the docID, which improves the user experience.

6 Conclusion

The developed search engine demonstrates robust performance in processing both TAAT and DAAT queries, with consistent results that verify the correctness of the index structure and query logic. The system excels in handling large datasets, efficiently executing queries, and offering an intuitive user experience through the web interface. The incorporation of the top-K MaxScore algorithm enhances performance, particularly for disjunctive queries, allowing for faster retrieval of top results.

Key strengths include the effective implementation of compressed index structures, the ability to efficiently merge and process large volumes of data, and the system's overall flexibility. Additionally, the enhancements to content display improve the user experience by

providing not just docIDs but also relevant content snippets, which align with typical user expectations.

While the current system uses BM25 for document ranking, real-world search engines often factor in additional elements such as user behavior, query context, and personalization to provide more refined results. This system provides a strong foundation for further advancements in ranking models and search functionalities.

References

- [1] Howard Turtle and James Flood. “Query evaluation: Strategies and optimizations”. In: *Information Processing Management* 31.6 (1995), pp. 831–850. ISSN: 0306-4573. DOI: [https://doi.org/10.1016/0306-4573\(95\)00020-H](https://doi.org/10.1016/0306-4573(95)00020-H). URL: <https://www.sciencedirect.com/science/article/pii/030645739500020H>.
- [2] Shuai Ding and Torsten Suel. “Faster top-k document retrieval using block-max indexes”. In: *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR ’11. Beijing, China: Association for Computing Machinery, 2011, pp. 993–1002. ISBN: 9781450307574. DOI: 10.1145/2009916.2010048. URL: <https://doi.org/10.1145/2009916.2010048>.