**Introduction:**

This document outlines a program designed to create and query an **inverted index**, a key structure in information retrieval that maps terms to the documents containing them. This efficient search structure is built by processing large text data, organizing it with optimized sorting, and storing it for quick access.

The program includes a query module that ranks documents using the **BM25 algorithm**, scoring documents based on term frequency and relevance to the search query. This report covers the program’s structure, setup, performance, and limitations, with insights into potential improvements, including advanced ranking and parallel processing for faster performance.

**High-Level Program Overview:**

The program comprises three primary components:

**Index Formation**:

* Constructs an inverted index from input text data, merging intermediate files and storing the final, compressed index for efficient storage and retrieval.

**Efficient Sorting and compression**:

* Optimizes data organization with memory-efficient sorting (using Heap Sort) and compresses the merged index with VarByte encoding. The compressed index is then stored in a binary format to save space.

**Query Handling**:

* Processes queries with the BM25 algorithm to rank documents by relevance, accessing compressed data directly from the binary file.

**User Interface**:

* Allows users to enter query requests through a browser interface (localhost) and displays the top-K ranked documents by relevance.

**Requirements and Installation Instructions**

**Overview**

This document provides a step-by-step guide for setting up, installing, and running a simple search engine program. The program consists of three main modules written in C++ that work together to create an efficient and compressed inverted index for processing search queries.

**Key Components:**

1. **Inverted Index Creation:** The program first parses document data and forms an intermediate inverted index.

2. **Efficient Merging and Compression:** Intermediate data is merged and compressed using variable-byte encoding.

3. **Query Processing:** Queries are processed using the BM25 ranking model to retrieve and rank relevant documents.

**Requirements**

1. **Operating System**: Linux or Windows with a compatible C++ compiler.

2. **C++ Compiler**: g++ (GCC version 7.5 or higher recommended).

3. **Libraries**:

• Standard C++ library

• If required, install a multithreading library for optimized merging (optional but recommended for large datasets).

4. **Dataset**: A directory of text files representing the document corpus for indexing.

5. **File Permissions**: Ensure that read and write permissions are granted for the directories used for input files, intermediate files, and output.

**Installation and Setup**

**Step 1: Preparing Input Files**

1. Create or ensure the presence of a directory containing text files for indexing. These files should contain the document text you wish to index.

2. Verify each text file’s readability and format compatibility (ASCII text recommended for optimal parsing).

**Step 2: Compiling the Program Files**

Navigate to the search-engine directory and compile the three C++ files using g++:

cd search-engine

g++ -o form\_inverted form\_inverted.cpp

g++ -o io\_efficient\_merge\_sort io\_efficient\_merge\_sort.cpp

g++ -o query query.cpp

This will produce three executables:

• form\_inverted: Parses and creates the intermediate inverted index.

• io\_efficient\_merge\_sort: Merges and compresses the inverted index.

• query: Processes queries against the compressed index.

**Program Execution Steps**

**Step 1: Run form\_inverted to Create Intermediate Inverted Index**

1. Execute the form\_inverted program:

./form\_inverted

2. This program will parse each document, extract terms, and create an ASCII file called document\_index.txt, containing:

• Document IDs and their respective lengths.

• The average document length for reference in scoring.

3. It also generates intermediate inverted index files.

**Step 2: Run io\_efficient\_merge\_sort for Merging and Compression**

1. Execute the io\_efficient\_merge\_sort program:

./io\_efficient\_merge\_sort

2. This program merges the intermediate inverted index and applies variable-byte encoding to compress the list.

3. Output files generated:

• Lexicon.txt: An ASCII file containing metadata on terms and their locations in the index.

• data.bin: A binary file holding the compressed inverted index in binary format.

**Step 3: Run query for Query Processing**

1. Execute the query program, providing it with the query terms to search within the indexed documents:

./query

2. The program reads from the compressed data.bin file and uses the Lexicon.txt file to map terms to document IDs.

3. It ranks and returns document IDs based on the BM25 scoring model.

**Step 4: Setting Up the User Interface for Query Processing (Optional)**

To provide a browser-based interface for submitting and viewing query results:

1. **Compile the Query Program**

First, ensure the query.cpp file is compiled to create an executable:

g++ -std=c++20 query.cpp -o query

2. **Start the Local PHP Server**

Run the following command to start a PHP server on localhost at port 8000:

php -S localhost:8000

3. **Integrate PHP and HTML Files**

The process.php script will call the compiled query executable to process search terms submitted via the HTML interface and display results.

4. **Access the UI in Your Browser**

Open your browser and navigate to the following URL:

http://localhost:8000

This interface allows you to enter query terms and view the top-ranked documents based on BM25 scoring.

5. **Clear Cache for New Queries**

After viewing the results, it’s recommended to reload the page to reset the query box and clear any cached data.

**Additional Notes**

• **Data Format**: Ensure that document text files are plain ASCII for optimal parsing.

• **BM25 Model**: The query processing module (query.cpp) uses the BM25 algorithm to calculate relevance scores, providing a ranked list of document IDs.

• **Performance**: For large document corpora, ensure adequate memory and CPU resources are available, as the merging and compression step can be resource-intensive.

• **Error Handling**: If a file is missing or incorrectly formatted, the program will output specific error messages.

**Example Usage**

After successful execution of the above steps, you can run the following command to test query processing:

./query "sample search terms"

Expected Output:

• A list of document IDs ranked by relevance based on the query terms provided, using BM25 scoring.

**Internal Mechanics**

**Module 1: form\_inverted.cpp**

This module is responsible for parsing and indexing the document corpus to create an intermediate inverted index. Its main functions include term parsing, frequency counting, and partial index generation.

1. **Document Parsing**:

• The program scans through each document in the corpus.

• Each term encountered in a document is stored along with its document ID and frequency count.

2. **Intermediate Index Creation**:

• For every 10,000 documents, an intermediate inverted index file is generated to manage memory usage efficiently.

• This intermediate index holds terms, document IDs, and term frequencies in each document.

3. **Document Statistics**:

• Alongside the inverted index, form\_inverted.cpp generates an ASCII file named document\_index.txt.

• This file contains:

• Document IDs, their lengths, and the average document length across the corpus.

• These statistics are essential for ranking documents using the BM25 scoring algorithm, as they provide normalization factors for term frequency and document length.

**Module 2: io\_efficient\_merge\_sort.cpp**

This module merges and compresses intermediate inverted index files, converting them into a final compressed binary format with the required metadata. The merging process leverages a buffer-based heap sort mechanism, optimized for memory efficiency.

1. **Heap-Based Merging**:

• The program merges the intermediate index files generated by form\_inverted.cpp using a heap sort algorithm.

• A buffer size of 1,000 inverted index entries is used for merging, though this can be adjusted based on the system’s memory capacity.

• During each merge operation, details of each term are populated in the lexicon file (Lexicon.txt).

2. **Lexicon File Population**:

• For each term, an entry is added to Lexicon.txt with the following details:

• **Term Name**: The text of the term.

• **Term ID**: A unique identifier for the term.

• **Number of Documents Containing the Term**: The document frequency of the term.

• **Binary Position of the Term in the Inverted Index**: The position within the binary file after compression, allowing quick access.

3. **Binary Conversion of the Inverted Index**:

• The merged inverted index is converted into a binary format (data.bin) for efficient storage and traversal.

**Representation of the Metadata of Inverted Index for a Term**

| **Field** | **Description** |
| --- | --- |
| TS | Total size of the term |
| LastDocIDSize | Number of last document IDs |
| LastDocIDList | List of last document IDs |
| TCS | Total number of chunks |
| CSL | Chunk size list (bytes) |

Example for a specific term in compressed binary format:

***[TS] [LastDocIDSize] [LastDocIDsList] [TCS] [CSL] [128 DocIds] [128 Frequencies] [128 DocIds] [128 Frequencies] ...***

• The binary format includes metadata for each term, organized as follows:

• **TS**: Total size of the term block.

• **LastDocIDSize**: Number of elements in the last document ID list.

• **LastDocIDList**: Stores the last document IDs from each batch of 128 document IDs.

• **TCS**: Total number of chunks in the list.

• **CSL**: List of chunk sizes in bytes for each 128-document batch.

4. **Compression**:

• Document IDs and term frequencies are encoded using variable-byte encoding to reduce file size.

• The compressed format ensures fast traversal and reduced storage overhead while retaining the necessary data for score calculation and traversal logic.

The primary operations in this module include:

• Merging the intermediate inverted indexes into a final index for each term.

• Converting the final inverted index into a binary format with metadata.

• Performing variable-byte encoding on document IDs and term frequencies.

• Populating the lexicon file with necessary information for efficient term lookup and scoring.

**Module 3: query.cpp**

The query.cpp module takes user queries, searches the compressed inverted index in data.bin, and retrieves ranked results based on the BM25 score.

1. **Query Processing**:

• The program first accepts the query from the user.

• Query terms are sorted based on the number of documents they appear in, with the term present in the fewest documents designated as the *primary term*.

2. **Document-at-a-Time (DAAT) Operation**:

• The program uses a conjunctive (AND) approach to process terms, reducing search time by focusing on documents containing all query terms.

• Starting with the primary term, each document ID is traversed, and for each document ID, the program retrieves matching IDs for the other terms in the query.

• The **LastDocIDList** is used to jump directly to relevant blocks, bypassing unnecessary data and reducing decompression overhead.

3. **BM25 Scoring**:

• If a document contains all query terms, the BM25 relevance score is computed using:

• Information from Lexicon.txt (term metadata),

• document\_index.txt (document lengths and average length),

• data.bin (compressed binary inverted index).

• The score represents the document’s relevance to the query.

4. **Ranking and Output**:

• The scores are stored in a max-heap, allowing efficient retrieval of the top-ranked documents.

• The program outputs the top K results in time, where is the number of top results requested, and is the total number of scored documents.

This approach ensures efficient and accurate retrieval of relevant documents, optimized for performance through selective traversal and compressed indexing.

**Performance Metrics**

The program’s performance varies based on dataset size and the complexity of queries. Key metrics include:

• **Runtime Analysis**:

• **Index Formation**: Measures the time to process and merge terms from input documents into an inverted index. Factors influencing runtime include document size and frequency of terms across documents.

• **Sorting**: Sorting efficiency is crucial for managing large datasets; it leverages batch processing to enhance I/O performance, especially during merges.

• **Querying**: Query runtime depends on the number of terms, document frequencies within the index, and is optimized through BM25 scoring, which enables precise ranking based on term occurrence.

• **Index File Sizes**: Index file sizes depend on dataset volume and term frequency. Initial tests on sample data suggest estimated file sizes can provide a basis for resource planning with larger datasets.

For MS MARCO passage-ranking dataset (**8841823 Documents**)

Size of Single Intermediate File (10000) Documents : **1 sec**

Total Time taken for generating entire Intermediate inverted index files : **19 min**

Total time taken to merge and compress the Intermediate inverted Index files : **4 min**

Total size of the lexicon.txt file : **26.2 MB**

Total size of the document\_index.txt file: **100.7 MB**

Total size of the data.bin (Final inverted index compressed binary file) : **285.3 MB**

Total time taken to process an Single word : **2 sec**

Average words per document **: 54**

**Limitations**

While the program is designed for efficiency, several constraints may arise with larger datasets:

• **Memory Constraints**: For very large datasets, memory usage may become significant, potentially affecting program performance and causing delays.

• **Index File Size**: When dealing with extensive document collections, the index file size can grow substantially, impacting storage and potentially slowing retrieval.

• **Character Constraints**: Terms containing semicolons are neglected when reading documents. This design choice prevents conflicts in generating the intermediate inverted index, as semicolons are used as separators in the output format.

**Conclusion and Future Directions**

In summary, this program provides a robust inverted index with a BM25-based query processor. It handles efficient indexing, sorting, and querying, establishing a solid foundation for further development.

Future improvements may include:

• **Enhanced Ranking**: Additional ranking techniques could include the context of the query using deep learning, potentially improving result relevance.

• **Parallelization**: Many subprograms could benefit from parallel execution, allowing the program to leverage system configurations for faster processing.

• **Optimized Compression**: Further data compression methods could be incorporated to reduce index file sizes without sacrificing access speed.