University of Pisa
Department of Information Engineering
Artificial Intelligence and Data Engineering
Multimedia Information Retrieval & Computer Vision course

Search Engine

By

Sultan Mahmud

1. Introduction

This report documents the implementation of a Multimedia Information Retrieval System, which processes and indexes large-scale text datasets using an Inverted Index and facilitates efficient query processing. The system follows the requirements outlined in the MIRCV Project Description and incorporates several advanced indexing and retrieval techniques.

The core objectives of this project are:

1. Indexing: Building an efficient Inverted Index to support scalable text search.
2. Query Processing: Executing ranked TF-IDF-based queries with conjunctive and disjunctive search.
3. Optimization: Improving index storage efficiency, query retrieval speed, and memory management.
4. Evaluation: Analyzing indexing performance, query execution times, and system efficiency.

This project was developed in C++ and follows best practices such as modular programming, efficient data structures, and multi-stage indexing.

2. Project Structure & Main Modules

The system is structured into distinct modules to separate concerns and maintain scalability:

1. Document Processing (DocumentParser.cpp & DocumentParser.h)
    - Parses the raw dataset.
    - Cleans, tokenizes, and normalizes text.
    - Supports stopword removal and stemming.
2. Indexing (InvertedIndex.cpp & InvertedIndex.h)
    - Implements Single-Pass In-Memory Indexing (SPIMI).
    - Builds partial index files and merges them into a final inverted index.
    - Stores document length, term lexicon, and mappings.
3. Query Processing (QueryProcessor.cpp & QueryProcessor.h)
    - Processes user queries through a command-line interface.
    - Supports conjunctive & disjunctive search.
    - Retrieves results using TF-IDF scoring.
4. Utility Functions (utils.cpp & utils.h)
    - Provides text preprocessing functions such as normalization, lowercasing, and stemming.
    - Supports UTF-8 to Unicode conversion.
5. Main Entry (main.cpp)
    - Initializes the system, builds the index, and executes queries.
    - Ensures smooth integration of all components.

3. Indexing Process

The indexing pipeline consists of several steps to transform raw documents into an efficiently searchable inverted index.

3.1 Document Preprocessing

This stage ensures that the raw text data is cleaned, tokenized, and normalized before indexing. The following preprocessing steps are applied systematically to every document.

Each document undergoes the following preprocessing steps:

3.1.1 Text Cleaning

Text cleaning is the first and most essential step in preprocessing, as it ensures that the data is in a structured format and free from unnecessary noise. The system performs the following cleaning operations:

1. Unicode Handling

- The dataset may contain text from multiple languages using different character encodings.
- The program ensures that all text is processed using UTF-8 encoding to support non-ASCII characters.
- Invalid Unicode sequences are detected and skipped to avoid processing errors.

2. Removing Special Symbols & Numbers

- Punctuation marks (e.g., .,!?;:()[]{}) are removed to keep only meaningful words.
- Special characters such as currency symbols, mathematical symbols, and control characters are eliminated.
- Numerical values are generally discarded, as they do not contribute to meaningful textual search.

3. Handling Malformed Lines & Empty Pages

- Some documents may contain missing content, corrupted characters, or unusual line breaks.
- The system identifies and removes empty or malformed documents to avoid indexing garbage data.
- A whitespace normalization step ensures that excessive spaces or line breaks are removed.

3.1.2 Tokenization & Normalization

Once the text is cleaned, it is broken down into tokens (words) and further normalized.

1. Tokenization

Tokenization is the process of splitting a sentence into individual words (tokens):

- The system uses whitespace and punctuation as delimiters to separate words.

2. Lowercasing

- All words are converted to lowercase to ensure case-insensitive search.
- This helps to avoid duplicates, e.g., *"Apple"* and *"apple"* are treated as the same term.

3. Handling Word Variations

- The system removes extra spaces and trims leading/trailing whitespace.
- If a word contains special annotations (e.g., “term/NN” or “(example)”), these characters are stripped out.

3.1.3 Stopword Removal & Stemming (Optional)

This step further refines the dataset to reduce noise and improve search efficiency.

1. Stopword Removal

- Common words that do not add meaningful search value (e.g., "the", "is", "and", "of") are removed.

2. Stemming (Porter Algorithm)

- Stemming reduces words to their root form, ensuring that word variations are treated as the same term.

3.2 Index Construction (SPIMI)

The Single-Pass In-Memory Indexing (SPIMI) algorithm is a highly efficient method used to construct an inverted index from a large collection of documents. Unlike traditional indexing methods that maintain a global data structure in memory, SPIMI processes documents in chunks, allowing the system to handle large-scale datasets while minimizing memory usage. The key idea behind SPIMI is to process and store smaller portions of the index incrementally, instead of attempting to build the entire index in memory at once.

Step 1: Chunk-wise Indexing

SPIMI operates by processing documents in manageable chunks, ensuring that the indexing process does not consume excessive system memory. When a document batch is read, the words are extracted, normalized, and inserted into a temporary in-memory data structure. Each term is mapped to a list of postings, which contain document IDs and term frequencies.

To improve efficiency, each batch of documents is indexed separately and stored as a partial inverted index on disk. This means that for every chunk processed, the system creates a temporary index file that stores term-posting pairs. When the allocated memory for a chunk reaches a predefined limit, the partial index is flushed to disk, and a new in-memory structure is initialized for the next chunk. This approach ensures that even large datasets can be processed without memory overflow.

For example, if the system is processing 1 million documents, instead of loading and indexing all of them at once, it divides the dataset into smaller subsets (e.g., 100,000 documents per chunk). Each subset is indexed separately and saved as a partial index file, which is later merged in the next step.

Step 2: Merging Partial Indexes

After all document chunks have been processed, the system is left with multiple partial inverted index files stored on disk. These partial indexes need to be merged to create a final, unified inverted index. The merging process ensures that the postings for each term from different index files are combined in lexicographic order, maintaining the structure necessary for efficient retrieval.

During merging, terms from different partial indexes are compared and consolidated, ensuring that the final index maintains an ordered and compressed list of postings for each term. The merging process follows a multi-way merge algorithm, which reads multiple partial index files simultaneously, picking the smallest (lexicographically first) term, and writing it to the final index. Each term's postings are also merged and sorted based on document IDs, allowing for efficient retrieval operations.

To further optimize this process, buffering and priority queues are often used to speed up merging. Instead of loading all partial index files into memory at once, only a portion of each file is read at a time, ensuring that disk I/O operations are minimized. This step significantly improves the efficiency of the index construction process.

Step 3: Lexicon & Metadata Storage

Once the final merged inverted index is built, additional metadata structures are stored separately to facilitate fast retrieval. These structures include:

- Term Lexicon: A mapping of terms to their corresponding inverted list locations in the final index file. This allows queries to quickly locate the relevant term's posting list without scanning the entire index.
- Document Length Table: Stores the length of each document, which is useful for computing TF-IDF and ranking scores during query execution.
- Document ID Mapping: Maintains a mapping between internal document IDs (used in indexing) and the original document identifiers. This ensures that search results can be linked back to the correct document.

By maintaining these structures separately, the retrieval process becomes much more efficient, as queries only need to access the relevant portion of the index rather than scanning the entire dataset. The lexicon allows quick lookups of term positions, while the document metadata enables fast ranking computations.

4. Query Processing

The query processing module is responsible for retrieving the most relevant documents based on a user query. The system supports ranked retrieval using the TF-IDF scoring method, which ensures that documents most relevant to the user's query appear at the top of the search results.

The system enables two types of query modes:

1. Conjunctive Search (AND) – Retrieves only documents that contain all the query terms.
2. Disjunctive Search (OR) – Retrieves documents that contain at least one of the query terms.

Efficient query execution is achieved through preprocessing, optimized retrieval operations, and ranking mechanisms, ensuring fast and accurate search results.

4.1 Query Execution

The query execution process consists of four main steps:

1. User Input Handling
2. Processing Query Terms
3. Ranking & Retrieval
4. Returning Results

Step 1: User Input Handling

When the system is initialized, it prompts the user to enter a search query. Queries can be single-word or multi-word search terms. The user is also asked to specify the search mode:

- Conjunctive Search (AND Mode): The system retrieves only those documents that contain all the words in the query. This ensures higher precision but may return fewer results.
- Disjunctive Search (OR Mode): The system retrieves any document that contains at least one of the query terms. This increases recall, ensuring more results are displayed.

Step 2: Processing Query Terms

Before searching for relevant documents, the system preprocesses the query to ensure consistency with the indexed terms. The query undergoes the following transformations:

1. Text Normalization
    - Converts all words to lowercase (e.g., "Machine" → "machine").
    - Removes special characters and punctuation (e.g., "data-science" → "data science").
2. Tokenization
    - Splits the input into individual words for processing.
3. Stopword Removal & Stemming (*Optional*)
    - Removes common stopwords (e.g., "the", "and", "in", "of").
    - Applies stemming to reduce words to their root form (e.g., "running" → "run").
4. Retrieving Posting Lists
    - For each preprocessed query term, the system fetches its posting list from the inverted index.
    - The posting list contains a list of documents that contain the term, along with term frequencies (TF).

Step 3: Ranking & Retrieval

Once the relevant posting lists have been retrieved, the system computes TF-IDF scores for each document. This ensures that documents containing frequent and important query terms appear higher in the search results.

1. Term Frequency (TF) Calculation
    - Measures how often a term appears in a document.

$$TF = \frac{\text{Number of times term appears in the document}}{\text{Total number of terms in the document}}$$

2. Inverse Document Frequency (IDF) Calculation
    - Reduces the importance of common terms and increases the importance of rare terms:

$$IDF = \log\left(\frac{\text{Total number of documents}}{\text{Number of documents containing the term}}\right)$$

3. TF-IDF Score Calculation
    - The TF-IDF score is computed for each document as:

$$TF - IDF = TF \times IDF$$

    - Documents with higher TF-IDF scores are ranked higher in the search results.
4. Sorting Results
    - The retrieved documents are sorted in descending order based on their TF-IDF score.

Step 4: Returning Results

After ranking, the system displays the top 20 most relevant documents based on their TF-IDF scores. The result format includes:

- Document ID
- Term Frequency
- TF-IDF Score

5. Performance Evaluation

This section presents the performance results for document parsing, indexing, and query execution. The system was tested on the MSMARCO Passage Ranking Dataset, containing 8,841,823 documents, with the following key evaluation metrics:

1. Indexing Time – The time taken to parse, preprocess, and construct the inverted index.
2. Index Size – The sizes of key indexing components: docIDs, lexicon, and document lengths.
3. Query Response Time – The efficiency of TF-IDF-based ranked retrieval.
4. Scalability & Efficiency – Evaluation of how well the system performs under large-scale data.

5.1 Indexing Performance

The indexing process was conducted using the SPIMI algorithm in C++, with an estimated chunk size of 1,000,000 documents per batch. The system processed 8,841,823 documents, splitting them into 9 partial index chunks, which were later merged into a final index.

The system was tested on **8,841,823 documents** with two different configurations:

1. **Without Stemming & Stopword Removal** (Baseline)
2. **With Stemming & Stopword Removal** (Optimized)

Indexing Performance Comparison

Configuration	Parsing & Preprocessing Time	Indexing Time	Total Time
Without Stemming & Stopwords	603.52 seconds	7,111.35 seconds (~1.97 hours)	7,714.87 seconds (~2.14 hours)
With Stemming & Stopwords	1,066.22 seconds	6,547.34 seconds (~1.82 hours)	7,613.56 seconds (~2.11 hours)

Index Size Comparison

Configuration	Lexicon Size	DocLengths Size	DocID Size
Without Stemming & Stopwords	1,665 MB	1,310 MB	1,210 MB
With Stemming & Stopwords	1,469 MB	1,300 MB	1,200 MB

Preprocessing Time Increased (+76%) with Stemming & Stopwords Removal

- The additional text processing (stemming and stopword removal) resulted in a higher parsing time (from 603.52s to 1,066.22s).

Indexing Time Decreased (~8%)

- With stemming and stopword removal, less redundant data was stored, leading to faster index creation.

Lexicon Size Reduction (~11.8%)

- The lexicon size dropped from 1,665 MB to 1,469 MB when stemming and stopword removal were enabled.
- This suggests that removing redundant words (stopwords) and reducing word variations (stemming) made indexing more compact.

5.2 Query Performance

The system supports ranked retrieval using TF-IDF and allows users to choose between conjunctive (AND) and disjunctive (OR) search modes.

Query Processing Performance

Configuration	Estimated Query Response Time
Without Stemming & Stopwords	29 - 38 ms
With Stemming & Stopwords	25 - 30 ms

Query Processing Speed Improved (~13-20% faster)

- Without stemming & stopwords, query response time was 29 - 38 ms.
- With stemming & stopwords, query response time improved to 25 - 30 ms.
- This speedup is due to smaller lexicon size and more compact index structures, which reduce disk I/O and memory lookup time.

5.3 Scalability & Efficiency

The system was compiled with the following optimizations:

- Enabled Stemming ⟶ Improved query performance by reducing word variations.
- Enabled Stopword Removal ⟶ Reduced index size by eliminating high-frequency common words.
- C++17 with UTF-8 Support ⟶ Ensured compatibility with multi-language text processing.

The incremental indexing approach (SPIMI) ensures that large datasets are handled efficiently, without exceeding memory constraints.

7. Limitations & Future Improvements

Although the system meets most of the project requirements, some improvements can be made:

1. Index Compression
    - Implement variable byte encoding to reduce index size.
2. Query Optimization

    - Add dynamic pruning (MaxScore, WAND) for faster retrieval.
3. Scoring Enhancements
    - Implement BM25 scoring for better ranking.
4. Memory Optimization
    - Reduce index memory footprint for handling larger datasets.

8. Conclusion

This project successfully implements an efficient multimedia information retrieval system based on inverted indexing and TF-IDF search ranking. The system can handle large datasets and provides fast query responses using optimized search algorithms. Future work will focus on further improving efficiency through index compression, better ranking algorithms, and query optimizations.