Vanilla Search Engine

Boyan Xu, *New York University*

***Abstract*— This paper presents a comprehensive description of a search engine program designed to efficiently process and respond to user queries using an inverted index structure. The program, developed as part of an assignment for a Computer Science and Engineering course at NYU Tandon School of Engineering, leverages the inverted index built in a previous homework assignment and introduces a sophisticated query processor as its core enhancement. The modular design of the program is encapsulated in several Rust source files, each contributing to different aspects of the search engine's functionality.**

***Index terms*—Information Retrieval, Inverted Index, Search Engine**

I. Introduction

In the realm of information retrieval and search engine technology, the efficient processing and management of large volumes of data is a fundamental challenge. This paper describes the design and implementation of a search engine program developed as part of an academic exercise for the Computer Science and Engineering course at NYU Tandon School of Engineering. The program is a continuation and enhancement of a previous assignment, where an inverted index was created. The current iteration extends this by introducing a robust query processor, capable of efficiently handling and responding to user queries.

A. Program overview

The search engine is built using Rust, a language known for its performance and safety features, making it an ideal choice for handling the complexities and demands of search engine architecture.

The program is structured into several modules, each responsible for a specific aspect of the search engine's functionality. These modules include:

1. `main.rs` as the program's entry point,
2. `parser.rs` for document parsing,
3. `indexer.rs` and `bin_indexer.rs` for indexing
4. `term_query_processor.rs` for query processing
5. `utils.rs` for utility functions
6. `external_sorter.rs` for IO-efficient external sorting
7. `disk_io.rs` for disk input/output operations.

We will look into implementation details of each component in the paper.

B. Tokenizer

The index is designed to support UTF-8 encoding. In `parser.rs` we normalizes the UTF-8 characters in documents with NFKC (Normalization Form KC: Compatibility Composition) and splits the normalized text into words (tokens) using the Rust crate `unicode-segmentation` [1].

We also replace certain characters (like ".", "_", "-", and quotation marks) with whitespace to treat them as delimiters, converts tokens to lowercase, filters out stop words, and omits numeric tokens below a certain threshold (less than 10).

C. Index structure

The index structure of the search engine program exhibits a three-level design. This hierarchical structure consists of the Directory File, Lexicon File, Index File, each playing a distinct yet interconnected role.

The Directory File acts as a streamlined lookup table, significantly accelerating access to term metadata in large datasets.

The Lexicon File serves as a comprehensive metadata repository, facilitating efficient term lookups and data decompression.

The Index File forms the core of the inverted index, containing compressed posting lists and term frequencies essential for impact score calculations.

More details will be discussed in the next section.

D. Query processor

The query processor adapts "Term-at-a-Time" (TAAT) strategy. In this model, the system processes each query term independently, retrieving and scoring all documents containing that term before moving on to the next term.

Our query processor supports both conjunctive and disjunctive queries. A conjunctive query (AND query) retrieves documents that contain all the specified terms, ensuring that each result is highly relevant to the entire query. If the user inputs "quantum AND mechanics AND applications" as a conjunctive query, the search engine will return documents that contain all three terms - "quantum," "mechanics," and "applications." This ensures the results are specifically focused on the applications of quantum mechanics, providing precise and targeted information.

On the other hand, a disjunctive query (OR query) broadens the search by retrieving documents that contain any of the specified terms. If the user inputs "quantum OR mechanics OR applications" as a disjunctive query, the search engine will return a broader range of documents. These documents could contain any one of the terms or a combination of them. The results might include general information on quantum theory, mechanics separately, or various types of applications, offering a wider array of information related to any of the terms.

Our query processor implements rudimentary caching feature, so repeated search

E. Web user interface

The program comes with a web query interface. The backend is written with `actix-web`[2]. The frontend is written with `tailwindcss`[3] and `htmx`[4].

II. Index layout

This section will discuss the binary layout of the inverted index in details.

A. Level 1 - Directory File

The first level is embodied by the Directory File, which functions as an efficient lookup table within the lexicon file.

This file contains pointers for every Nth term into the Lexicon File, enabling rapid access to term metadata without the need to scan the entire Lexicon. This directory structure is particularly advantageous for large datasets, where quick term access is paramount.

It essentially acts as an index of the lexicon, providing a shortcut to relevant term information and significantly speeding up the search process.

```
+---------------------------------+
|   Total Directory Entries (4B)  |
+---------------------------------+
|   Term 1 for Entry              |
|   (Variable-length)             |
+---------------------------------+
|   Pointer to Term 1 in Lexicon  |
|             (4/8B)              |
+---------------------------------+
|               ...               |
|   (Repeats for every Nth term)  |
+---------------------------------+
```

- **Total Directory Entries**: Total number of terms stored in this directory.
- **Pointer to Term 1 in Lexicon**: offset to the start of the term's metadata in the Lexicon File

B. Level 2 - Lexicon File

At the second level is the Lexicon File, which serves as a comprehensive metadata repository for every term in the index. This file is crucial for efficient term lookups and decompression of posting lists.

It stores a variety of metadata, including the term's length, its unique identifier (`TermID`), and statistical data like document frequency and total term frequency.

The Lexicon File also holds pointers to the term's posting list in the Index File, along with additional metadata necessary for data retrieval and decompression, such as the number of compressed blocks, the size of the final block, and the highest docID for each term.

```
+----------------------------------+
|          Total Terms (4B)        |
+----------------------------------+
|   Term 1 Length (2B)             |
+----------------------------------+
|   Term 1 Literal                 |
|   (Variable-length)              |
+----------------------------------+
|   TermID for Term 1 (4B)         |
+----------------------------------+
|   Document Frequency for T1 (4B) |
+----------------------------------+
|   Total Term Frequency for T1 (4B)|
+----------------------------------+
|   Pointer to Compressed List for |
|     T1 in Index File (4/8B)      |
+----------------------------------+
|   Number of Blocks for T1 (4B)   |
+----------------------------------+
|   Size of Last Block for T1 (4B) |
+----------------------------------+
|   Last docID for T1 (4B)         |
+----------------------------------+
|   Total Bytes for Compressed List |
|         for T1 (4/8B)            |
+----------------------------------+
|   Block Offsets for T1           |
|  (4/8B each x Number of Blocks)  |
+----------------------------------+
|   Block Maxima List for T1       |
|   (4B each x Number of Blocks)   |
+----------------------------------+
|                ...               |
|   (Repeats for subsequent terms) |
+----------------------------------+
```

- **Term Literal**: the actual string representation of the term.
- **Total Term Frequency**: Total occurrences of the term across all documents.
- **Pointer to Compressed List**: Byte offset to the start of the compressed postings in the Index File.
- **Number of Blocks**: Number of compressed blocks for the term's postings.
- **Size of Last Block**: Number of postings in the smaller final block.
- **Last docID**: The highest docID where the term appears (used for block skipping).

- **Total Bytes for Compressed List**: Total bytes consumed by compressed postings of the term in the Index File.
- **Block Offsets List**: Byte offsets for the start of each compressed block in the Index File.
- **Block Maxima List**: The highest docID in each block (used for block skipping).

C. Level 3 - Index File

The third level of the index structure is the Index File, which is fundamental to the search engine's capability to retrieve posting lists.

It stores the core elements of the inverted index – the compressed posting lists. These lists consist of sorted document IDs (docIDs) encoded in delta form (delta from previous one), associated with each indexed term and are segmented into blocks using Stream VByte compression [5]. This method significantly reduces the storage space required while maintaining quick access to the data.

Additionally, the Index File contains uncompressed frequencies for each term, detailing how often a term appears in each document, which will be used to calculate impact score (BM25) during query processing.

```
+--------------------------------+
|  Compressed Block for Term 1... |
|         ...                     |
|         ...                     |
+--------------------------------+
|  Smaller Last Block for Term 1  |
+--------------------------------+
|  Uncompressed Frequencies for T1 |
|         ...                     |
|         ...                     |
+--------------------------------+
|         ...                     |
|  (Repeats for subsequent terms.) |
+--------------------------------+
```

- The block size is 64, but the final block for each term can be smaller than 64.
- The frequency values corresponding to the docIDs above

III. Design Decision

In this section, we will introduce the problem background of a few of our design decisions.

A. I/O efficient

Our program is designed with a focus on I/O efficiency, a crucial aspect for handling large-scale data processing tasks typical in search engine operations. I/O efficiency primarily revolves around optimizing the way the program reads from and writes to disk, as these operations can be significant bottlenecks, especially when dealing with vast amounts of data. Here are some key features of our I/O efficient design:

1. **Buffered I/O Operations**: By using buffered reads and writes (`BufReader` and `BufWriter`), the program minimizes the number of direct disk access operations. This buffering allows for larger chunks of data to be read or written at once, reducing the overhead associated with frequent disk accesses.
2. **External Sorting Mechanisms**: We implement an external mrege-sort in `external_sorter.rs` that enable the program to process data that exceeds the available memory.
   - `MergingIterator`: This struct facilitates sequential reading of sorted data from files. It employs a buffered reader to efficiently manage file I/O and includes a function `next()` to retrieve the next element in the sorted file.
   - `merge_sorted_files()`: This function creates a BinaryHeap (as a min-heap) to efficiently find the next smallest item across all input files, ensuring a correctly sorted order in the output file.
3. **Sequential Disk Access**: Where possible, the program favors sequential disk access patterns over random access. Sequential access is typically faster as it reduces the time spent on disk seek operations, which is beneficial for operations like indexing and merging sorted data.

B. Data compression

In our search engine, we have implemented data compression techniques, specifically stream-vbyte encoding and delta encoding, to optimize storage and improve query processing efficiency.

1. **Stream-VByte Encoding**: Stream-VByte [5] is a variable-byte encoding technique used to compress the posting lists, where document IDs (docIDs) are stored. This method is particularly effective for compressing integer sequences, making it ideal for compressing the lists of docIDs in our inverted index. We use Marshall Pierce's implementation in the crate `stream-vbyte` [6].
2. **Delta Encoding**: Alongside stream-vbyte, we employ delta encoding, a method where we store differences between successive docIDs instead of the docIDs themselves. This technique takes advantage of the fact that docIDs in the posting lists are often close to each other, leading to smaller differences and, consequently, more efficient compression. Delta encoding not only reduces the size of the stored data but also

speeds up the process of reading and decompressing the data during query execution.

The implementation of data compression techniques, specifically stream-vbyte and delta encoding, in our search engine offers several significant benefits:

1. **Reduced Storage Space**: One of the most immediate advantages of compression is the reduction in storage requirements. By compacting the posting lists, stream-vbyte and delta encoding decrease the size of the indexed data, leading to more efficient utilization of disk space. This is particularly beneficial when dealing with large-scale data sets typical in search engine operations.
2. **Improved I/O Efficiency**: Compressed data requires less disk I/O to read and write, which can be a major performance bottleneck, especially for large files. By reducing the volume of data that needs to be transferred between disk and memory, compression significantly enhances the overall I/O efficiency of the system.
3. **Balancing CPU Load**: While compression and decompression add some computational overhead, the reduction in data volume that needs to be processed can balance out this extra CPU load. Efficiently compressed data can lead to overall lower CPU cycles for processing the same amount of information.

However, it's important to note that, currently, our implementation does not apply compression to the frequencies associated with each docID. The frequencies are stored in an uncompressed format, which may result in increased storage requirements and potentially slower access times compared to a fully compressed index. In future iterations of our search engine, incorporating frequency compression could further optimize storage and query processing, contributing to a more efficient search system.

C. *Block skipping*

In our search engine, we have implemented block-skipping to enhance the efficiency of conjunctive queries. Block-skipping is a technique that accelerates query processing by bypassing certain blocks of data that are irrelevant to the current search query.

In conjunctive queries, where the search result must contain all query terms, scanning through every block of postings lists for each term can be time-consuming, especially in large datasets. This is where block-skipping becomes advantageous. By skipping over blocks of postings that do not contain the document IDs being searched for, we significantly reduce the amount of data processed, speeding up the query.

Our block-skipping approach is evident in the `query_term_postings_after_doc_k` function. Here's how it works:

1. **Metadata Utilization**: The function begins by fetching the metadata for the term in question, which includes `block_maxima` - the maximum document ID in each block of postings for the term.
2. **Iterative Block Assessment**: We iterate through the blocks of postings. For each block, we compare its maximum document ID (`max_docid`) with a specified document ID threshold (`k`). If `max_docid` is less than `k`, it indicates that the entire block contains document IDs that are not relevant to the query, allowing us to skip this block entirely.
3. **Selective Processing**: Once we encounter a block where `max_docid` is greater than or equal to `k`, we start processing from this block. We seek to the block's starting position in the index file and begin reading and decompressing the document IDs.
4. **Delta Decoding and Frequency Reading**: After adjusting the first document ID in the block (if necessary), we perform delta decoding to retrieve the actual document IDs. We also read the corresponding frequencies for these document IDs.
5. **Postings Compilation**: We compile the postings (document ID and frequency pairs) for the current term, filtering out those document IDs that are below the threshold `k`.

By implementing block-skipping, our search engine becomes more adept at handling conjunctive queries efficiently. This method significantly reduces the I/O and computational load, especially in cases where the queried terms have a wide distribution across the document set.

D. *Impact score*

Our search engine employs the BM25 algorithm [7] for computing impact scores, a decision motivated by BM25′s effectiveness in information retrieval. BM25 is a widely-accepted ranking function that calculates the relevance of documents to a given search query, based on the terms present in each document.

The BM25 scoring formula is a widely-used ranking function in information retrieval systems, including search engines. It is based on the probabilistic information retrieval model and calculates the relevance of a document to a given search query. The formula for BM25 is as follows:

$$\text{Score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{(f(q_i,D)\cdot(k_1+1))}{\left(f(q_i,D)+k_1\cdot\left(1-b+b\cdot\frac{|D|}{\text{avgdl}}\right)\right)} \tag{1}$$

where:

- $\text{Score}(D,Q)$ is the BM25 score of a document D given a query Q,
- n is the number of query terms,
- $\text{IDF}(q_i)$ is the inverse document frequency of the query term q_i,
- $f(q_i,D)$ is the frequency of the query term q_i in the document D,
- $|D|$ is the length of the document D in words,
- avgdl is the average document length in the text collection from which documents are drawn,
- k_1 and b are free parameters, chosen as $k_1 = 1.2$ and $b = 0.75$ in our implementation.

The $\text{IDF}(q_i)$ component is calculated as:

$$\text{IDF}(q_i) = \ln\left(\frac{(N-n(q_i)+0.5)}{(n(q_i)+0.5)}+1\right) \tag{2}$$

where:

- N is the total number of documents, and
- $n(q_i)$ is the number of documents containing the query term q_i.

Our implementation of BM25 features:

1. **Dynamic Score Calculation**: In our implementation, during a conjunctive query, BM25 scores are calculated in real-time for each document that contains the query terms. The function `bm25` computes the score based on term frequency (`tf`), document frequency (`df`), and document length, along with the total number of documents and the average document length in the corpus.
2. **Use of Metadata**: The index stores essential metadata, such as document frequency and document length, which are used in the BM25 calculation. This metadata facilitates the dynamic computation of relevance scores by providing the necessary data for each document and term in the query.

A notable limitation in our current implementation is the real-time computation of BM25 scores. Precomputing and storing certain components of the BM25 calculation (such as the Inverse Document Frequency (IDF) component) could potentially improve query performance.

However, this would require a trade-off between the freshness of the data and the speed of query execution, a consideration that would need to be balanced based on the specific requirements and usage patterns of the search engine.

E. Term-at-a-Time (TAAT)

Opting for the Term-at-a-Time (TAAT) approach over the Document-at-a-Time (DAAT) in our search engine was a strategic decision, influenced by our index structure and performance optimization goals.

1) Why TAAT:

1. **Index Structure Compatibility**: Our index is optimized for quick term access and efficient term-level operations. TAAT aligns perfectly with this structure, allowing us to process each term's postings list independently and utilize our index's capabilities, such as block-skipping and compressed data handling, to their fullest.
2. **Simplified Query Logic**: TAAT simplifies the query processing logic, especially when dealing with complex queries involving multiple terms. It offers a more straightforward approach to intersecting postings lists for conjunctive queries, as each list is processed and filtered independently before the intersection.

2) Why Not DAAT:

1. **Complexity with Compressed Data**: Our index's use of compression techniques would add complexity to a DAAT approach. In DAAT, accessing and decompressing specific parts of each term's postings list for every document can be computationally more intensive and less efficient.
2. **Inefficient Block-Skipping**: DAAT would not leverage our index's block-skipping feature effectively. This feature is more aligned with a term-centric approach where entire blocks can be skipped if they are irrelevant to the query term.

In summary, the choice of TAAT over DAAT in our search engine is rooted in the specific characteristics of our indexing strategy and the need for efficient query processing. TAAT offers a more compatible, efficient, and resource-effective approach given our engine's design and operational context.

F. Numeric Token

In our search engine, we have made a deliberate design decision to include numbers greater than 10 as valid tokens during the parsing phase. This choice is rooted in a balance between the relevance and specificity of numerical data in search queries.

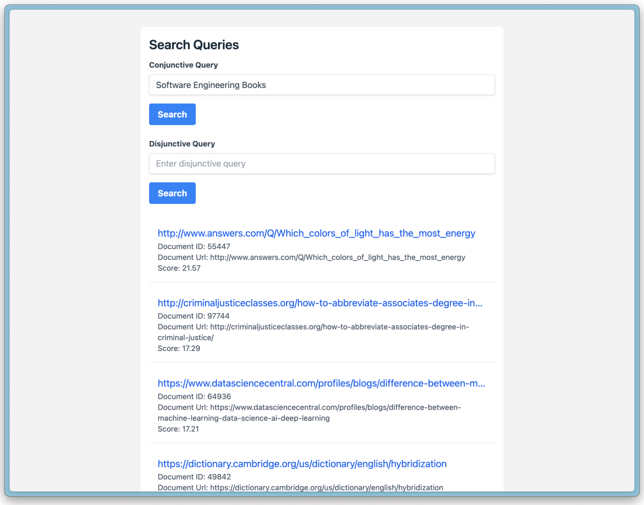
1. **Relevance of Larger Numbers**: Numbers larger than 10 often carry significant information content.

For instance, years (like 2021), quantities, measurements, and other numerical data above this threshold are likely to be relevant in the context of search queries. Including these numbers ensures that the search engine captures important, specific information that might be crucial for users' search intents.

2. **Filtering Out Less Relevant Numbers**: Numbers less than 10 are often considered less specific and can appear frequently in texts without carrying substantial individual significance (such as in counts, lists, or common phrases). By filtering these out, we reduce the noise in our data, focusing instead on more distinctive numerical tokens that are likely to contribute more meaningfully to a search query's context.
3. **Optimizing Index Size and Search Precision**: Including only numbers greater than 10 helps optimize the index size by preventing it from being bloated with common, less informative numerical values. This optimization can lead to more precise search results, as the indexed numbers are more likely to be part of relevant and specific search queries.

IV. Web Query Interface

Our search engine program comes equipped with a user-friendly web query interface, blending a powerful backend written in `actix-web` with a stylish and responsive frontend designed using `tailwindcss` and `htmx`. This combination offers a seamless user experience and efficient search functionality.



Software Engineering Books

http://www.answers.com/Q/Which_colors_of_light_has_the_most_energy
Document ID: 55447
Document Url: http://www.answers.com/Q/Which_colors_of_light_has_the_most_energy
Score: 21.57

http://criminaljusticeclasses.org/how-to-abbreviate-associates-degree-in...
Document ID: 97744
Document Url: http://criminaljusticeclasses.org/how-to-abbreviate-associates-degree-in-criminal-justice/
Score: 17.29

https://www.datasciencecentral.com/profiles/blogs/difference-between-m...
Document ID: 64936
Document Url: https://www.datasciencecentral.com/profiles/blogs/difference-between-machine-learning-data-science-ai-deep-learning
Score: 17.21

https://dictionary.cambridge.org/us/dictionary/english/hybridization
Document ID: 49842
Document Url: https://dictionary.cambridge.org/us/dictionary/english/hybridization

Figure 1: The web user interface for searching.

A. Backend with `actix-web`:

- **Robust and Scalable**: The backend of the web interface is powered by `actix-web` [2], a high-performance web framework for Rust. Known for its speed and scalability, `actix-web` provides a robust foundation for handling web requests and managing the search engine's query processing.
- **Asynchronous Query Handling**: It efficiently manages asynchronous web requests, ensuring that the search engine can handle multiple queries simultaneously without performance bottlenecks.

B. Frontend with `tailwindcss` and `htmx`:

- **Stylish and Responsive Design**: The frontend leverages `tailwindcss` [3], a utility-first CSS framework, to create a visually appealing and responsive design. This ensures that the interface is accessible and user-friendly across various devices and screen sizes.
- **Dynamic and Interactive**: `htmx` [4] is utilized to enhance the interactivity of the web interface. It allows for dynamic content updates without full page reloads, making the search experience smoother and more responsive. With `htmx`, search results are dynamically loaded and displayed, offering a seamless user interaction.

The interface supports both conjunctive and disjunctive query, and presents the search results with relevant details like document ID, URL, and BM25 scores.

V. Program Setup

Our search engine program is developed in Rust, a language known for its performance and safety, and is compiled into a single binary, ensuring ease of deployment and execution. Here's a guide on how to run the program in different modes:

A. Setup Rust

Before running our search engine program, you'll need to set up the Rust environment and install necessary dependencies. Here's a step-by-step guide:

1. **Install Rust**:
   - Rust can be installed using `rustup`, which is the official Rust toolchain installer.
   - To install `rustup`, open your terminal and run:
     ```
     curl --proto '=https' --tlsv1.2 -sSf
     https://sh.rustup.rs | sh
     ```
   - This command will download a script and start the installation. Follow the on-screen instructions to complete the setup.
   - After installation, you can manage Rust versions and associated tools with `rustup`.
2. **Update Rust**:

- To ensure you have the latest version, you can update the Rust toolchain using:

  ```
  rustup update
  ```

3. **Check Rust Installation**:
   - To verify that Rust is installed correctly, run:

     ```
     rustc --version
     ```
   - This command will display the version of Rust compiler (`rustc`) installed on your system.

4. **Clone the Program Repository**:

   ```
   git clone https://github.com/BoyanXu/Inverted-Index
   ```

5. **Install Dependencies**:
   - Navigate to the project directory (where the `Cargo.toml` file is located) and run:

     ```
     cargo build
     ```
   - This command compiles the project and downloads and installs all the necessary dependencies specified in `Cargo.toml`.

B. Download dataset

For our search engine to function properly, it needs access to a dataset. We'll be using the MSMARCO dataset for this purpose. Here's how you can set up the data folder and download the dataset:

1. **Create a Data Directory**:
   - First, create a directory in the project folder to store the dataset. You can do this manually or via the command line. For command line, navigate to the project's root directory and run:

     ```
     mkdir data
     ```
   - This command creates a new directory named `data` in your project folder.

2. **Download the MSMARCO Dataset**:
   - The MSMARCO dataset can be downloaded from the official source. Use the following command to download the dataset directly into the `data` directory:

     ```
     curl https://msmarco.z22.web.core.windows.net/msmarcoranking/msmarco-docs.trec.gz -o data/msmarco-docs.trec.gz
     ```
   - This command uses `curl` to download the dataset and saves it as `msmarco-docs.trec.gz` in the `data` directory.

3. **Verify the Download**:
   - Ensure that the file has been downloaded correctly by checking its presence in the `data` directory. You can list the contents of the directory using:

     ```
     ls data
     ```
   - This should show `msmarco-docs.trec.gz` listed among the contents.

By following these steps, you'll have the necessary data directory set up and the MSMARCO dataset downloaded, allowing the search engine program to process and index the data for query processing.

C. Compile, build, run

There

1. **Using Serde JSON for Serialization (Debug Mode)**:
   - In this mode, the program uses `serde_json` for serialization, which is useful for debugging as it outputs human-readable JSON files.
   - To run the program in this mode, use the command:

     ```
     cargo run --package Inverted-Index --features debug_unicode --bin Inverted-Index
     ```
   - This command compiles and runs the program with the `debug_unicode` feature enabled, allowing you to trace and inspect data in a more understandable format.

2. **Using Bincode for Serialization (Production Mode)**:
   - For production use, `bincode` is used for serialization. It provides a more compact and efficient binary format, suitable for handling large datasets.
   - Run the program with this serialization method using:

     ```
     cargo run --package Inverted-Index --bin Inverted-Index
     ```
   - This command runs the program in its default mode, which is optimized for performance and storage efficiency.

3. **Building a Release Version**:
   - For deploying the search engine, it's recommended to build a release version, which optimizes the program for performance.
   - Compile a release build using:

     ```
     cargo build --release
     ```
   - This command creates an optimized binary in the `./target/release` directory.

4. **Running the Backend Server**:
   - Once the program is built, you can run the backend server to interact with the search engine through its web interface.
   - Start the server with:

```
RUST_LOG=debug ./target/release/Inverted-
Index
```

- This command runs the server in debug logging mode, enabling detailed logs for troubleshooting or monitoring server activity. Now you can visit the web query interface at: `127.0.0.1:8080`.

By following these instructions, you can compile and run our search engine in various modes depending on your needs, whether it's for development, debugging, or deployment in a production environment.

VI. Program Structure

In this section, we visit each module of our source code.

A. *Overview of `main.rs`*

The `main.rs` file serves as the entry point of our search engine program. It integrates various modules and sets up the core functionality, including the web interface and the indexing process. Here's a detailed overview of its key components:

1. **Module Imports**:
   - Imports several modules like `parser`, `indexer`, `utils`, `disk_io`, `external_sorter`, `bin_indexer`, and `term_query_processor`, each handling specific functionalities within the search engine.
2. **Utility Functions**:
   - `cleanup_postings_data_folder`: A function to clean up the directory used for storing intermediate postings data. This ensures a clean start for index building.
3. **Index Building Function (`build_index`)**:
   - Orchestrates the process of building the inverted index. It involves processing the raw data file, applying external merge sort on the batches, and building the binary inverted index for efficient query processing.
4. **AppState Structure**:
   - Defines the `AppState` struct, encapsulating the `TermQueryProcessor`. This shared state is crucial for handling queries in the web interface.
5. **Query Handling Functions**:
   - `handle_conjunctive_query` and `handle_disjunctive_query`: These asynchronous functions handle web requests for conjunctive and disjunctive queries, respectively. They process the queries using the `TermQueryProcessor` and return the results in JSON format.
6. **Web Server Setup (`main` function)**:
   - Initializes the `TermQueryProcessor` and sets up the Actix web server.
   - Defines routes for conjunctive and disjunctive query processing and serves static files for the web interface.
   - The server listens on `127.0.0.1:8080`, establishing the local endpoint for web interactions.
7. **Actix Web Framework**:
   - Utilizes `actix_web` to define and manage web routes, request handling, and HTTP responses. The use of `actix_files` serves static files, essential for the frontend interface.

This `main.rs` file is pivotal in tying together various components of the search engine, providing a cohesive structure that supports both the core functionalities of indexing and query processing, and the user interface for web-based interactions.

B. *Overview of `indexer.rs`*

The `indexer.rs` module in our search engine plays a crucial role in indexing documents and managing the data necessary for efficient query processing. Here's a breakdown of its key components and functionalities:

1. **Indexer Structure**: `Indexer` is the main struct that encapsulates the indexing logic. Contains three primary data structures:
   - `postings`: A `HashMap` that holds the temporary postings lists. Each list maps a token ID to another `HashMap` of document IDs (`docID`) and their corresponding frequency.
   - `doc_metadata`: A `HashMap` that stores metadata about the documents, such as their URL and the number of terms in each document.
   - `term_id_map`: A `BiMap` (bidirectional map) for efficient mapping between terms and their unique IDs. It supports both term-to-ID and ID-to-term lookups.
2. **Key Methods in `Indexer`**:
   - `new`: Constructs a new `Indexer` instance with initialized data structures.
   - `process_document`: Parses a document to extract its docID, URL, and tokens. It updates `doc_metadata` and constructs the postings lists by aggregating term frequencies.
   - `dump_postings_to_disk`: Writes the current postings lists to disk and clears them from memory. This method is crucial for managing memory usage during indexing.

- `dump_lexicon_to_disk`: Converts the `term_id_map` to a standard `HashMap` and writes it to disk. This lexicon is vital for query processing, as it allows the system to map terms to their unique IDs.
- `dump_doc_metadata_to_disk`: Writes the `doc_metadata` to disk, which is essential for storing information about the documents in the corpus.

3. **Indexing Process**:
    - The `Indexer` is responsible for creating the inverted index. It processes each document, updates postings lists and metadata, and ensures these are stored both in memory and on disk.
    - The use of `HashMap` and `BiMap` allows for efficient data handling and quick lookups, which are critical for both indexing and subsequent query processing.
4. **Memory Management**:
    - Regularly dumping postings to disk helps manage the memory footprint of the indexer, ensuring that the system remains efficient even when processing large datasets.

In summary, the `indexer.rs` module is central to the functionality of the search engine, handling the critical task of indexing documents. It balances in-memory data management with persistent storage, ensuring efficient and effective indexing that lays the foundation for the search engine's query processing capabilities.

C. *Overview of `parser.rs`*

The `parser.rs` module in our search engine plays a critical role in parsing and preprocessing text data from documents. This module prepares the raw text for the indexing process by extracting and normalizing relevant information. Here's an overview of its key components and functionalities:

1. **Document ID Handling**:
    - `DOCID_COUNTER`: An atomic counter used to assign unique identifiers to documents.
    - `get_doc_id`: A function that increments and returns the next document ID. This ensures each document processed by the indexer is uniquely identified.
2. **Core Parsing Functions**:
    - `parse_document`: The primary function that orchestrates the parsing process. It extracts the document ID, URL, and tokenizes the text content of a document.
    - `extract_url`: Extracts the URL from the document's text content. This typically represents the first line of the text.
    - `extract_text_content`: Uses a regular expression to extract the main textual content of a document enclosed within specific tags (e.g., `<TEXT>`).
3. **Text Normalization and Tokenization**:
    - `parse_line`: Performs the normalization of text to Unicode NFKC (Compatibility Composition) and replaces various punctuation marks with whitespace to treat them as delimiters. It then tokenizes the text into words using Unicode segmentation.
    - Filters out stop words and numbers less than 10 to focus on more meaningful and specific terms in the text.
4. **Efficiency and Relevance**:
    - The use of `lazy_static` for caching stop words and atomic operations for document IDs are key for efficiency.
    - The filtering criteria in `parse_line` are designed to balance the inclusion of relevant terms and exclusion of common, less informative elements, enhancing the search engine's focus on relevant content.

In summary, the `parser.rs` module is integral to the preprocessing phase of the search engine, ensuring that documents are parsed, normalized, and tokenized effectively. This preparation is crucial for building a robust and efficient inverted index, laying the foundation for accurate and relevant search query processing.

D. *Overview of `disk_io.rs`*

The `disk_io.rs` module in our search engine is crucial for handling disk input/output operations, particularly focusing on data compression, decompression, and serialization. This module ensures efficient data storage and retrieval, which is essential for the search engine's performance. Below is an overview of its key functionalities:

1. **Key Functions and Structures**:
    - `decompress_gzip_file`: Decompresses gzip files using `GzDecoder`, returning a `BufRead` for efficient line-by-line reading.
    - `process_gzip_file`: Main function for processing compressed document files. It initializes logging, reads documents, and processes them using the `Indexer` module. It also manages batch processing for postings and dumping data to disk.

2. **Data Dumping Functions**:
   - `write_posting_to_disk`: Serializes and writes postings data to disk. It handles both debug (JSON format) and production (binary format) modes.
   - `write_lexicon_to_disk`: Serializes the lexicon (term-to-ID mappings) and writes it to disk, supporting both text and binary formats.
   - `write_doc_metadata_to_disk`: Serializes and saves document metadata (like document URL and length) to disk.
3. **Data Processing and Merging**:
   - `merge_sorted_postings`: Merges multiple sorted postings files into a single file using the `merge_sorted_files` function from the `external_sorter` module.
   - `load_doc_metadata`: Loads document metadata from a file into a `HashMap`, supporting both debug and production modes.
4. **Configuration and Error Handling**:
   - Uses conditional compilation (`cfg` attributes) to handle different modes (debug vs. production).
   - Comprehensive error handling to ensure robustness during I/O operations.

In summary, `disk_io.rs` is pivotal for managing the large volumes of data our search engine processes. It efficiently handles the reading and writing of compressed and serialized data, ensuring that the search engine's performance is optimized for both the indexing and query processing stages.

E. *Overview of `external_sorter.rs`*

The `external_sorter.rs` module is a critical component in our search engine that handles the external sorting of postings lists. This module is essential for managing large data sets that exceed the memory capacity, ensuring efficient sorting and merging of postings. Here's an overview of its key functionalities:

1. **`MergingIterator` Structure**:
   - Designed to facilitate the merging of sorted postings lists from multiple files.
   - `reader`: A `BufReader` associated with each file to be merged, allowing efficient reading of data.
   - `new`: A constructor that initializes a `MergingIterator` with a given file.
   - `next`: Retrieves the next term and its corresponding postings list from the file. This function handles both debug (JSON) and production (binary) data formats.
2. **`ReverseOrdered` Structure**:
   - A utility structure used in conjunction with a binary heap to manage the merging process.
   - Implements ordering traits (`PartialEq`, `Eq`, `Ord`, and `PartialOrd`) to ensure that the terms are correctly ordered in the heap, facilitating the merge operation.
3. **External Merging Functionality**:
   - `merge_sorted_files`: The primary function responsible for merging multiple sorted files into a single file.
   - Uses a `BinaryHeap` (min-heap) to efficiently find and merge the smallest (next in order) term and its postings from multiple files.
   - Ensures that the terms are merged in the correct order, producing a single, comprehensively sorted postings list.
4. **Postings Writing Function**:
   - `write_posting`: A helper function that writes a term and its postings list to the output file. It handles both debug and production modes, serializing the data into either JSON or binary format.

In summary, the `external_sorter.rs` module is pivotal in the indexing process, handling one of the most resource-intensive tasks: sorting large postings lists. Its implementation of external sorting and merging ensures that the search engine can efficiently process and index large datasets, a critical aspect of its scalability and performance.

F. *Overview of `bin_indexer.rs`*

The `bin_indexer.rs` module in our search engine is responsible for building the binary inverted index, a critical component for efficient query processing. This module focuses on organizing and storing the indexed data in a binary format, optimizing both space and access speed. Here's an overview of its key components and functionalities:

1. **`TermMetadata` Structure**:
   - `TermMetadata` struct holds metadata for each term, such as term ID, document frequency, total term frequency, and pointers to postings list data in the index file.
   - Includes information about the structure of the postings list, such as the number of blocks, size of each block, and max docID in each block, crucial for block-skipping and efficient data access.
2. **Building the Binary Index**:
   - `build_bin_index`: The main function that orchestrates the building of the binary index. It opens the postings file, creates buffer writers

for the index, lexicon, and directory files, and processes each term's postings list.
- Handles both debug and production modes, with different data serialization formats (JSON for debugging and binary format for production).

3. **Index Postings Function**:
   - `index_postings`: Processes each term's postings list, encoding docIDs using `stream_vbyte` and writing the encoded data and frequencies to the index file.
   - Manages metadata for each term, including tracking offsets and maxima for block-skipping.
   - Writes term metadata to the lexicon file and updates directory entries as necessary.
4. **Binary Data Handling**:
   - Utilizes `stream_vbyte` for efficient encoding of postings lists, reducing index size and enhancing access speed.
   - Employs `byteorder` and `bincode` for handling binary data serialization and deserialization.
5. **Directory File Management**:
   - Manages directory entries for efficient term lookups, crucial for query processing. This involves writing term lengths, terms themselves, and their positions in the lexicon file.

In summary, the `bin_indexer.rs` module is fundamental in transforming the raw postings data into a structured binary inverted index. Its efficient handling of data encoding, metadata management, and binary serialization lays the foundation for the search engine's performance in terms of both storage efficiency and query processing speed.

G. *Overview of `term_query_processor.rs`*

The `term_query_processor.rs` module is a crucial part of our search engine, managing the query processing logic. It interfaces with the indexed data to execute search queries efficiently. Here's an overview of its key components and functionalities:

1. **Structures and Data Handling**:
   - `TermQueryProcessor`: The main struct responsible for query processing. It maintains readers for directory, lexicon, and index files, as well as caches for directory and term metadata.
   - `SearchResult` and `QueryResponse`: Structures used to serialize query results into JSON format, containing fields like document ID, URL, and score.
2. **Query Processing Functions**:
   - `new`: Initializes the `TermQueryProcessor` with paths to the index, lexicon, and directory files. Loads document metadata and calculates the average document length.
   - `query_term_directory`: Retrieves the position of a term in the lexicon file, utilizing caching for efficiency.
   - `query_term_metadata`: Fetches metadata for a term, such as document frequency and term frequency, and caches this information.
   - `query_term_all_postings`: Retrieves all postings for a term, decompressing and delta decoding the docIDs.
   - `query_term_postings_after_doc_k`: Fetches postings for a term starting from a specific document ID, useful for query optimization.
3. **Query Types**:
   - `conjunctive_query`: Handles conjunctive (AND) queries by finding the intersection of postings lists of the query terms. Utilizes the BM25 scoring function for ranking results.
   - `disjunctive_query`: Processes disjunctive (OR) queries by combining postings lists of the query terms and scoring each document.
4. **Scoring and Ranking**:
   - `bm25`: Implements the BM25 scoring function, which is used to rank documents based on their relevance to the query terms.
5. **Utility Functions**:
   - `doc_url`: Retrieves the URL of a document given its ID.
   - `delta_decoding`: Decodes a list of delta-encoded docIDs into their original values.
6. **Efficiency and Performance**:
   - The module is designed for efficient query processing, leveraging caching and optimized data structures. It interacts with the binary index to quickly access and process the required data for query execution.

In summary, the `term_query_processor.rs` module is central to the search engine's capability to process and respond to user queries. It combines advanced data handling techniques, efficient caching, and effective scoring algorithms to deliver fast and relevant search results.

H. *Overview of `utils.rs`*

The `utils.rs` module in our search engine serves as a central configuration and constants repository. This module defines various parameters and constants that are used throughout the program, ensuring consistency and ease of configuration. Here's a breakdown of its key components:

1. **Debugging Constants**:
   - `DEBUG_MODE`: A boolean flag to toggle debugging mode on or off. This can affect how data is processed, stored, or logged throughout the application.
   - `DEBUG_DOC_LIMIT`: Sets the maximum number of documents to process when in debug mode, allowing for controlled testing and debugging with a limited dataset.
   - `BATCH_SIZE`: Determines the size of each batch of documents to be processed before dumping postings to disk. This is set relative to the `DEBUG_DOC_LIMIT` to ensure manageable data handling during debugging.
2. **Indexing Constants**:
   - `BLOCK_SIZE`: Specifies the size of each block in the postings lists. This is crucial for block encoding and impacts both the indexing and query processing.
   - `DIRECTORY_NTH_TERM`: Determines the interval at which terms are stored in the directory for the binary index. This parameter is vital for optimizing term lookups during query processing.
3. **BM25 Scoring Constants**:
   - `BM25_K1` and `BM25_B`: These constants are parameters for the BM25 scoring function, widely used in information retrieval to rank documents based on their relevance to a query. The values of these constants are chosen based on common practices and can be adjusted for tuning the scoring behavior.

The `utils.rs` module is a simple yet essential part of the search engine, providing a centralized location for configuring various aspects of the system. By adjusting these constants, developers can easily tweak the performance, debugging level, and behavior of the search engine to suit different requirements or testing scenarios.

VII. Key Metrics

A. Index Size

The search engine program, as analyzed from the provided source code, produces several key files as part of its indexing process. Each file serves a distinct purpose in the construction and utilization of the inverted index, reflecting the efficiency and complexity of the system. The sizes of these files are as follows:

1) *`bin_directory.data`:* 5.3 MB

This file contains lookup table for lexicon bin_lexicon, facilitating quick access and retrieval.

2) *`bin_lexicon.data`:* 1.8 GB

This file probably contains a lexicon or dictionary for the indexed terms, which is crucial for query processing and term lookup.

3) *`bin_index.data`:* 4.3 GB

The main component of the inverted index, this file stores the associations between terms and documents, forming the backbone of the search engine's retrieval capabilities.

4) *`doc_metadata.data`:* 231 MB

Contains metadata about the documents, such as URLs, document lengths, or other pertinent information that aids in search relevance and ranking.

B. Runtime

The runtime for various stages of the indexing process, based on the processing of 3 million documents, is as follows:

1. **Process Document to Postings**:
   - Duration: *1 hour, 6 minutes, and 8 seconds*.
   - This phase involves parsing documents and generating initial postings, which are the first step in creating the inverted index.
2. **External Merge Postings**:
   - Duration: *4 hours, 57 minutes, and 57 seconds*.
   - During this stage, the postings are sorted and merged, a crucial step for large datasets that do not fit entirely in memory.
3. **Index Building**:
   - Duration: *2 minutes and 47 seconds*.
   - This final stage compiles the sorted postings into the final index structure, optimizing it for efficient query processing.

Additionally, the runtime for performing a conjunctive query for `software engineering books` is *3 minutes and 47 seconds*. This duration reflects the time taken to process the query against the indexed data, including term lookup, document retrieval, and ranking.

VIII. Discussion

Our search engine, while robust in its current form, does have areas where improvements can be made. Addressing these limitations will enhance its performance, scalability, and usability. Here's a discussion on potential future enhancements:

1. **SIMD Support for External Merge Sort**: Implementing Single Instruction, Multiple Data (SIMD) operations can significantly speed up the external

merge sort process. SIMD allows parallel processing of data, which can drastically reduce the time taken to sort large datasets.

2. **Memory Mapped Files (mmap)**: Utilizing memory-mapped files for reading and writing large datasets can improve the I/O efficiency. `mmap` provides faster access by mapping a file directly into the process's virtual memory space, reducing the overhead involved in traditional file I/O operations.
3. **Alternative Impact Scoring Mechanisms**: Exploring other scoring algorithms beyond BM25, such as TF-IDF or newer neural models, could provide more accurate or contextually relevant search results, enhancing the engine's effectiveness.
4. **Pre-computing BM25 Scores**: Pre-computing and storing BM25 scores for frequently searched terms can reduce query processing time. This approach, however, requires a trade-off between storage space and computational efficiency.
5. **Frequency Data Compression**: Implementing compression techniques for frequency data in postings lists can significantly reduce the index size, making the search engine more scalable and memory-efficient.
6. **Improved Crate File Organization**: Refactoring the codebase for better modularity and separation of concerns can enhance maintainability and readability. Organizing functionalities into more intuitive crates and modules will make the system more developer-friendly.
7. **Configurable Buffer Sizes**: Allowing dynamic configuration of buffer sizes for reading and writing operations can optimize memory usage. This is particularly useful for adapting the system to different hardware configurations.
8. **Implementing Advanced Data Structures**: Employing more advanced data structures, such as trie or B-tree for term storage and retrieval, could provide performance benefits in certain aspects of the system.
9. **Web Interface Enhancements**: Improving the user interface of the web front end, incorporating features like auto-complete, query suggestions, and advanced search options, can significantly enhance user experience.
10. **Scalability and Distributed Processing**: To handle larger datasets, implementing distributed processing and index sharding could be explored. This would involve partitioning the dataset and processing it in parallel across multiple nodes.

Each of these improvements comes with its own set of challenges and considerations. Some may require significant architectural changes, while others could be implemented with minor tweaks. The key is to prioritize these enhancements based on the impact they have on the performance and user experience of the search engine.

References

[1] Unicode-Rs, "GitHub - unicode-rs/unicode-segmentation: Grapheme Cluster and Word boundaries according to UAX#29 rules". [Online]. Available: https://github.com/unicode-rs/unicode-segmentation

[2] Actix, "GitHub - actix/actix-web: Actix Web is a powerful, pragmatic, and extremely fast web framework for Rust.". [Online]. Available: https://github.com/actix/actix-web

[3] tailwindlabs, "GitHub - tailwindlabs/tailwindcss: A utility-first CSS framework for rapid UI development.". [Online]. Available: https://github.com/tailwindlabs/tailwindcss

[4] bigskysoftware, "GitHub - bigskysoftware/htmx: </> htmx - high power tools for HTML". [Online]. Available: https://github.com/bigskysoftware/htmx

[5] D. Lemire, N. Kurz, and C. Rupp, "Stream VByte: Faster byte-oriented integer compression", *Information Processing Letters*, p. 1, 2018, doi: 10.1016/j.ipl.2017.09.011.

[6] "stream-vbyte 0.4.1 - Docs.rs". [Online]. Available: https://docs.rs/crate/stream-vbyte/latest

[7] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond", *Found. Trends Inf. Retr.*, no. 4, p. 333, Apr. 2009, doi: 10.1561/1500000019.