Program Overview

The C++ Information Retrieval Program under examination is a comprehensive system designed to perform document processing, indexing, merging, and compression. In this section, we will provide an in-depth overview of the program's primary functions, its architecture, and its overarching goals.

1 Functions and Processes

Document Parsing and Indexing

At its core, this program parses a collection of text documents, extracting and indexing words. This indexing process generates an inverted index, where each word is associated with a list of pairs. Each pair contains a document ID and the frequency of that word within the document. This initial step is pivotal for enabling efficient search and retrieval.

Batch Processing and Intermediate Indexing

To manage the indexing process efficiently, the program employs a batch processing approach. The documents are divided into manageable batches, and subindexes are created for each batch. The size of these batches is determined by a configurable parameter, with a typical choice being 100 documents per batch or till it hits the maximum memory usage limit. Batch processing reduces memory overhead and allows for incremental indexing of a large document collection.

Merge Sort Operations

After subindexes are created, the program initiates merge sort operations to combine them into larger intermediate indexes. The merge process is iterative, with smaller indexes being combined into progressively larger ones. This systematic merging ultimately results in a single, consolidated inverted index.

VByte Compression

In addition to creating the inverted index, the program compresses the data to enhance storage efficiency and retrieval speed. It employs VByte compression, a technique for encoding integers. This compression process converts document IDs and their corresponding word frequencies into a more compact format. The compressed data is stored in a binary file, "compressed_inverted_index.bin."

2 Program Architecture

The program's architecture is characterized by its modular approach to document processing and indexing. Key components and processes include:

- **Document Reader:** Responsible for reading documents from an input file, in this case, "msmarco-docs/fulldocs-new.trec."

- **Inverted Index:** A central data structure that holds the word-document associations.

- **Batch Processing:** Divides the document collection into manageable chunks for intermediate indexing.

- **Merge Sort:** Combines subindexes to create intermediate indexes.

- **VByte Compression:** Converts data to a more compact format for storage.

- **Binary Output:** The compressed inverted index is saved in "compressed_inverted_index.bin."

- **Lexicon Index:** An accompanying index, "lexicon_index.txt," provides metadata about each word's position in the compressed index.

How to Run the Program

1 Prerequisites

Before running the program, make sure you have the following prerequisites in place:

- C++ Compiler: Ensure that you have a C++ compiler installed on your system. Common choices include GCC (GNU Compiler Collection) and Clang.

- Input Data: The program expects a collection of text documents as input. In this example, the input file is "msmarco-docs/fulldocs-new.trec." Make sure you have the document collection in the specified format or adjust the program to work with your data.

- Dependencies: Review the program's code and documentation for any specific library or tool dependencies that need to be installed.

2 Compilation

To compile the program, follow these steps:

1. Open a terminal or command prompt.

2. Navigate to the directory containing the program's source code.

3. Use the C++ compiler to compile the program.

4. If compilation is successful, you will have an executable binary ready for use.

3 Execution

After compiling the program, you can run it using the following steps:

1. In the same terminal or command prompt, navigate to the directory where the compiled binary is located.

2. The program will start executing, processing the input documents, creating the inverted index, performing merge sort operations, and generating the compressed inverted index.

4 Configuration

Depending on the program's design, you may have the option to configure parameters such as batch size and input/output file paths. Review the source code and to understand how to customize the program's behavior according to your specific requirements.

5 Output

Once the program completes its execution, it will generate several output files, including the compressed inverted index and possibly a lexicon index. These files are typically created in the program's working directory, so make sure to check for the presence of these files and their location.

Internal Workings

In this section, we will delve into the internal processes of the C++ Information Retrieval Program. Understanding how the program performs document parsing, indexing, merging, and compression is crucial to appreciate its functionality fully.

1 Document Parsing and Indexing

The heart of the program lies in its ability to parse documents and create an inverted index. Here's a breakdown of how this process works:

Document Reading
- The program reads documents from the specified input file, in this case, "msmarco-docs/fulldocs-new.trec."

Inverted Index Creation
- As each document is processed, the program extracts words and transforms them into lowercase to ensure consistent indexing.
- It maintains an inverted index structure, represented as an unordered map. Words serve as keys, and their corresponding values are lists of pairs.
- Each pair consists of a document ID and the frequency of the word in that document.
- The program checks whether a word already exists in the index. If it does, it updates the frequency for the respective document. If not, a new entry is created.

2 Intermediate Indexing

Batch processing and intermediate indexing are key techniques employed to manage memory and facilitate incremental processing of a large document collection. Here's how these processes work:

Batch Processing
- The program splits the document collection into manageable batches. The batch size is often a configurable parameter, such as 100 documents per batch or till it hits max memory usage limit.
- Each batch is processed separately, and subindexes are created for each batch.

Merge Sort Operations
- After subindexes are created, the program initiates merge sort operations. These operations involve merging smaller subindexes into progressively larger intermediate indexes.
- The merging process is iterative and continues until a single, consolidated inverted index is generated.

3 VByte Compression

To optimize storage and retrieval efficiency, the program employs VByte compression for encoding document IDs and their corresponding word frequencies. Here's a detailed look at this compression process:

VByte Encoding
- VByte encoding is a method used to compress integers. The program encodes the frequency of words and document IDs using this technique.
- For each integer, the program extracts 7 bits at a time, setting the most significant bit (MSB) if additional bytes are required.
- These compressed integers are stored in the final inverted index.

Binary Output
- The compressed inverted index is saved as a binary file named "compressed_inverted_index.bin." This binary format is optimized for efficient storage and retrieval.

Lexicon Index
- Alongside the compressed index, the program generates a lexicon index in a text file named "lexicon_index.txt." This file provides metadata about each word, including its position in the compressed index.

4 Program Components

The program's architecture consists of several essential components:

- Document Reader: Responsible for reading and parsing input documents.
- Inverted Index: The central data structure used for indexing words.
- Batch Processing: Divides documents into manageable batches.
- Merge Sort:** Combines subindexes to create larger indexes.
- VByte Compression: Converts data into a more compact format.
- Binary Output: The final compressed inverted index is saved in binary format.
- Lexicon Index: Contains metadata about each word in the index.

5 Significance of Information Retrieval

Understanding the internal workings of the program is crucial, as information retrieval and indexing play a fundamental role in various applications. From web search engines to data analysis, the efficient organization and retrieval of information are critical components of modern technology.

Major Functions and Modules

To gain a deeper understanding of the C++ Information Retrieval Program, it's essential to examine its major functions and modules. In this section, we will provide detailed insights into the key components of the program, shedding light on their roles and functionalities.

1 `createIndex`

The `createIndex` function is responsible for building the inverted index, a critical data structure in information retrieval. Here's a closer look at this function:

- Functionality: The `createIndex` function takes as input an unordered map, `Index`, which represents the inverted index. It also receives a vector of words and a document ID. For each word in the vector, the function processes it, converts it to lowercase, and checks whether it already exists in the index. If the word is found, it updates the frequency count for the current document. If the word is not present, a new entry is created with the document ID and a frequency of 1.

- Role: This function is central to the document parsing and indexing process, as it populates the inverted index with word-document associations. It ensures that words are consistently processed in lowercase and tracks their frequencies within documents.

2 `writeIndex`

The `writeIndex` function handles the writing of subindexes to text files. Here's an overview of its functionality:

- Functionality: The `writeIndex` function takes an unordered map, `Index`, as well as a name for the subindex file. It opens the file and iterates through the index. For each word in the index, it writes the word, followed by a colon, and then the list of document ID-frequency pairs for that word. The function appends this information to the file.

- Role: This function is responsible for persisting the intermediate subindexes on disk. These subindexes are later merged to form the final inverted index. The function ensures that subindex data is stored in a structured and organized manner.

3 `writeDocID`

The `writeDocID` function manages the creation of a document ID file. Here's an overview of its functionality:

- Functionality: `writeDocID` is responsible for writing document IDs and their corresponding values (e.g., URLs) to a file. It takes a map of document IDs and their values, as well as the current document ID and value to be written. The function opens the file and appends the document ID and value in the specified format.

- Role: This function maintains a record of document IDs and their associated values. This information is used for document retrieval and identification, forming a critical part of the information retrieval system.

4 `mergeFiles`

The `mergeFiles` function is pivotal for merging subindexes and creating intermediate indexes. Here's an overview of its functionality:

- Functionality: The `mergeFiles` function takes a vector of filenames representing subindexes and an output filename for the intermediate index. It combines the subindexes by grouping word-document associations based on words. The merged data is then written to the output file.

- Role: This function is instrumental in the merging process, which consolidates smaller subindexes into larger intermediate indexes. It ensures that the data from various subindexes is merged systematically and efficiently.

5 `mergeSort` and `mergeSortloop`

The `mergeSort` function, along with its counterpart `mergeSortloop`, handles the sorting and merging of intermediate indexes. Here's an overview of their functionality:

- Functionality: The `mergeSort` function takes two filenames representing intermediate indexes, sorts their data, and writes the merged data to an output file. It employs a merge operation to combine data from the two indexes.

- ‘mergeSortloop` Functionality: The `mergeSortloop` function orchestrates the iterative merge process. It takes batch size and total file count parameters, initiating multiple rounds of merging. It facilitates the progressive creation of larger intermediate indexes.

- Role: These functions are integral to the merging phase of the program. They ensure that subindexes and intermediate indexes are combined systematically and efficiently, resulting in a consolidated inverted index.

Output

Time Required to Create the Final Index = 3.30 hours.
Time Required to Compress the Final Index = 16 minutes.

Size of Final Index = 12.7 GB.
Size of COmpressed Index = 7.94 GB.