Program ReadMe

Table of Contents

ALGORITHM ➖

1. **Create an Unsorted File:**
   - Start with an unsorted file containing a large collection of documents. These documents may be in various formats, such as plain text or structured data.

2. **Split the File into N Files:**
   - Divide the unsorted file into N smaller files, where N is chosen so that each of these smaller files can fit into memory for efficient processing. This is done to break down the problem into manageable chunks.

3. **Within Each File:**
   - For each of the N smaller files, follow these steps:

   a. **Sort the File In-Memory:**

- Load the entire contents of the smaller file into memory (RAM).
- Perform an in-memory sort operation on the document collection within this file. This step helps arrange the documents within the file in a specific order, such as alphabetical order based on document IDs.

b. **Generate (TermID, DocID) Pairs and Create a Posting List:**
- While sorting the documents, generate (TermID, DocID) pairs, where TermID is a unique identifier for each term (word) and DocID is a unique identifier for each document.
- Maintain an inverted index or posting list that stores which documents each term is present in. This list allows for efficient searching and retrieval.

c. **Store Positional Information and Document-Term Frequency:**
- Optionally, store additional information about the term's position within the document and its frequency within each document. This information can be useful for more advanced search operations.

d. **Write to Disk:**
- Write the sorted and indexed documents back to disk as a new file. This new file contains the documents in a sorted order and the associated inverted index information.

4. **Merge the Sorted Runs:**
- After processing all N smaller files, you will have N sorted and indexed files. Now, merge these files to create a single consolidated index.

a. **Merge Runs:**
- Take M files at a time (the number of files that can fit in memory without exceeding available memory constraints).
- For each group of M files, read the first line from each file, and choose the lowest string based on an ASCII comparison.
- Write this chosen line to a new file, effectively merging the runs.
- Continue this process until all lines from all files have been processed. Delete the original files after they are merged since they are no longer needed (this helps in memory efficiency).

b. **Continue Merging:**
- Continue the above merging process iteratively until you are left with just one final file, which contains the complete sorted and indexed collection.

5. **Final Output:**
- The final output is a single file that contains the entire document collection sorted by some criteria (e.g., document IDs). Additionally, you have an inverted index or posting list that helps in efficient searching and retrieval.

6. **Time Complexity:**

- The time complexity of this algorithm is primarily determined by the sorting and merging steps. In theory, sorting dominates the time complexity. However, in practice, steps 1 and 4 (merge) are often the most time-consuming, especially when dealing with large document collections. The time complexity is typically O(N log N), where N is the number of words (terms) present in the entire collection.

1. Introduction

This ReadMe document provides a comprehensive explanation of the program. The program is designed to create an index from a large collection of documents efficiently. This document will cover the program's functionality, how to run it, its internal workings, execution time, index size, design decisions, limitations, and a conclusion.

2. Program Overview

The program encompasses various steps and components to create an inverted index efficiently:

2.1 Decompression

Decompression:
- The program opens the input file using the `gzFile` data type, a part of the zlib library.
- It decompresses the data line by line using zlib's `gzgets` function, which is efficient for processing large compressed datasets.You should adjust the buffer size according to your needs. After decompression, you can continue with the parsing and processing logic.This approach allows you to handle compressed data efficiently by decompressing small chunks at a time while parsing. i create separate files to store the lexicon and pagetable .after uncompressing the line we extract document information line by line and perform stemming .

2.2 Stemming

Stemming:
- After decompression, the program performs stemming on the text.
- The Porter2 stemmer is employed, which processes a vector of strings and writes the stemmed words along with document IDs to an output file.
- The output file likely contains data in the format "stemmed_word doc_id."

2.3 Page Table Information

Page Table Information:
- The program extracts additional document information, such as URLs and document sizes.
- This information is stored in a separate file, known as the "page table."
- The page table is a crucial component of the index, providing metadata about the documents in the collection.

2.4 Sorted Runs Creation

Sorted Runs Creation:
- To ensure efficient memory usage, the data is split into multiple files of nearly equal size, such that each file can be processed entirely in memory.
- Within each of these files, the data is sorted in-memory.
- During the sorting process, the program generates (termID, docID) pairs.
- Additionally, the program maintains information about the documents in which each term is present.
- Once the in-memory sorting and processing are complete, the data is written to disk as a sorted run.
- These sorted runs serve as temporary storage units before they are merged into a final index.

2.5 Merging Sorted Runs

Merging Sorted Runs:
- The merging step is crucial in creating the final index from the sorted runs.
- The merging process effectively combines the data from multiple sorted runs into a single, sorted index.
- The program reads and processes data from the sorted runs, efficiently managing the merging of posting lists for the same terms.
- Additionally, term metadata, including start and end positions and document count, is updated during this process.
- This final merged index represents the complete index of the document collection.

2.6 Binary Files

Binary Files:
- The program stores certain data structures, such as the page table and lexicon, in binary files.
- Storing data in binary format is more space-efficient than plain text, reducing disk space overhead.

2.7 Compression

Compression:
- The program applies basic delta encoding compression to the docCount values in the binary index file.
- This compression method helps further reduce the size of the index files while maintaining data integrity.

3. How to Run the Program

To run the program, follow these steps:

1. Compile the code using the following command, adjusting paths as needed:
   ```
   g++ -I "path-to-boost-headers" "path-to-zlib-headers" final_index.cpp porter2_stemmer.cpp -o source
   ```

2. Execute the compiled program:
   ```
   ./source
   ```

4. Internal Workings

Let's delve into the internal workings of each program component:

4.1 Decompression

Decompression:
- The program opens the input file using `gzFile` and decompresses data line by line using zlib's `gzgets` function.

4.2 Stemming

Stemming:
- After decompression, the program performs stemming on the text using the Porter2 stemmer.
- The stemming process takes a vector of strings, stems each string, and writes the results to an output file.

4.3 Page Table Information

Page Table Information:
- The program extracts URL and size information for each document.
- This information is stored in the page table file.
- The structure of the page table file typically includes document ID, URL, and size information.

4.4 Sorted Runs Creation

Sorted Runs Creation:
- The program creates sorted runs as an intermediate step in the indexing process.
- The data is split into multiple files of nearly equal size to allow in-memory processing.
- Each of these files is sorted in-memory, and (termID, docID) pairs are generated.
- These pairs are temporarily stored in memory and flushed to disk as a sorted run when memory is full.
- The sorting process ensures that the terms within each run are in lexicographic order.

4.5 Merging Sorted Runs

Merging Sorted Runs:
- The program manages the merging of sorted runs to create the final index.
- It opens and initializes a set of input files, one for each sorted run.
- The program reads the first line of each input run to extract the initial term and its associated posting list.
- Data structures are maintained to keep track of the terms and their associated posting lists in each run.
- The program efficiently merges the sorted runs, combining posting lists for the same term and updating term metadata.
- Word metadata, such as start and end positions and document counts, is managed during the merging process.
- The merged data is written to an output file, which represents the final index.
- Term metadata is also written to a lexicon file in text and binary formats.

4.6 Binary Files

Binary Files:
- The program stores page table and lexicon information in binary files.
- These binary files are space-efficient and provide faster data access compared to plain text files.
- The binary format is used to write data structures with document information and term metadata.

4.7 Compression

Compression:
- The program applies basic delta encoding compression to docCount values in the binary index file.
- Delta encoding involves storing the difference between values rather than the actual values.
- This method helps reduce the size of the index files.

5. Execution Time and Index Size

- The execution time of the program

 depends on the size and complexity of the input dataset.
- For the provided dataset, the program took approximately 28,898 seconds (around 8 hours) to create the final index.
- The size of the resulting index files may vary based on the dataset size, but they are designed to be memory-efficient, minimizing disk space usage.

6. Design Decisions

Let's discuss the key design decisions made in the program:

- **Merge-Based Indexing**:
  - The program utilizes a merge-based indexing approach, which is efficient in terms of memory usage and can handle collections of various sizes.
  - This approach ensures that the program remains scalable and practical even for large datasets.

- **Compression**:
  - The program applies basic delta encoding compression to further reduce index file sizes.
  - This design decision aims to strike a balance between minimizing disk space usage and maintaining the integrity of the index.

- **Binary Files**:
  - Storing certain data structures, such as the page table and lexicon, in binary format minimizes disk space usage.
  - This design choice optimizes the storage of metadata and other information.

7. Limitations

It's important to be aware of the limitations of the program:

- **Memory Usage**:
  - While the program is memory-efficient, extremely large collections may still pose memory constraints.
  - The program needs to fit the entire vocabulary in memory, limiting the volume of data that can be indexed on a single machine.

- **Rigidity in Sorting**:
  - The sorting method used in the program may become less efficient for very large datasets.
  - For extremely large datasets, more advanced sorting algorithms might be required.

- **Compression Improvement**:
  - The program applies basic delta encoding compression, which can be further optimized for better compression ratios.
  - Improving compression methods could result in even smaller index file sizes.

- **Block Skipping**:
  - The program does not implement metadata for block skipping, which is a technique used for efficient querying.
  - This feature could enhance the query performance of the final index.

8. Conclusion

In conclusion, the merge-based indexing program is a robust solution for efficiently creating an index from a collection of documents. It follows a merge-based approach that balances memory efficiency with the need to handle collections of varying sizes. The program optimizes disk space usage through compression and binary file storage.

While it has certain limitations for extremely large datasets and has room for further compression optimization, the program is a practical solution for creating indexes that can be queried efficiently. I intended to make further improvements in the next assignment.