READ ME

Introduction:

This readme document provides a comprehensive overview of my Indexer, which includes components such as an inverted index, page table, and lexicon data structure. These components are systematically created as the code proceeds and the final inverted index gets created as a compressed file in binary format.

I have used Xcode for my project.

How to Run the Program with Xcode on macOS:

To run my program on macOS using Xcode, follow these detailed steps:

1. **Xcode Installation:**
   - Ensure that you have Xcode installed on your macOS. You can download and install it from the Apple App Store or the official Apple website.

2. **Project Setup:**
   - Open Xcode and create a new C++ project. Name your project and choose a location for it.

3. **Add Source Files and maintain the directory structure:**
   - In your Xcode project, add the following files to your project:
     - ‘main.cpp’
     - 'indexer.cpp '
     - 'page_table.cpp '
     - 'text_processing.cpp '
     - 'file_operations.cpp '

Following is the directory structure:

```
inverted_indexer/
├── main.cpp
├── indexer.h / .cpp
├── page_table.h / .cpp
├── text_processing.h / .cpp
├── file_operations.h / .cpp
├── docs.trec
├── subinvertedindx/ (sub-index files)
├── merged/ (merged index files)
├── output/ (generated output files)
```

4. **Build and Run:**
   - Build your project by clicking the "Build" button in Xcode. This will compile the C++ source code and generate the executable.

5. **Run the Program:**
   - After building, you can run the program by clicking the "Run" button in Xcode.

6. **View Output:**
   - The program will generate 3 files:
      1) A compressed inverted index binary file named final_compressed_index.bin,
      2) A page table named PageTable.txt
      3) A Lexicon structure called lexicon.txt

   You can find these files in the output folder.

Program Functionality:

My indexer performs a range of functions, including:

- **Indexing:** It creates an inverted index that maps words to the documents in which they appear, along with their frequencies. Index is created by going through the .trec files in batches of 1000 and identifying documents and writing the index to intermediate text files. These files are merged and sorted and then finally compressed

- Memory limiter: As mentioned in instructions I have a memory limiter in place that restricts the memory usage till 1 GB. If while running the batches memory exceeds 1 GB it directly writes the file to the disc

- **Page Table:** It maintains a page table that associates document IDs with their respective URLs. It is a text file generated while creating the intermediate index files. It is stored in the output folder.

- **Sorting and Merging:** The program sorts and merges intermediate index files efficiently in batches of 100 and then again in batches of 2 until one file is left.

- **Compression:** It compresses the index files using a var-byte compression reaching compression up to 37%.

Internal Working:

The program follows a structured set of internal steps to achieve its functionality:

1. **Reading Data:** It reads a TREC data file, extracting text and URL information. The code reads the TREC file, which typically contains a collection of documents in a specific format, to extract and index their content. In the 'file_operations.cpp' file, the 'readTRECFile' function is responsible for this task. It initiates by opening the TREC file, and then it iterates through the file's lines. During this traversal, the code identifies and extracts specific content enclosed within tags like '<TEXT>' and '</TEXT>', which often denote the main body of the document. It also captures the document's URL information.

2. **Indexing:** For each document, it tokenizes and indexes the text data by creating an inverted index. It is primarily accomplished through the 'indexer.cpp' file. As the code reads and processes documents from the TREC file, it begins by sanitizing and tokenizing the text, breaking it down into individual words. For each word, the code converts it to lowercase to ensure uniformity. It then maintains an inverted index, represented as a hash map, associating each word with a list of document pairs. This list contains document IDs and their respective frequencies of occurrence for the given word. If the word is encountered multiple times within the same document, the code increments the frequency accordingly. If the word has not been seen before in the document, a new entry is created in the list. In essence, the code captures the semantic structure of the document collection by mapping words to the documents in which they appear, along with their frequencies.

3. **Page Table:** Simultaneously, it constructs a page table associating document IDs with URLs. The page table is responsible for mapping document IDs to their corresponding URLs, providing a convenient and efficient way to locate the actual documents associated with the indexed data. In the 'page_table.cpp' file, the 'createPageTable' function receives as input a document ID and the URL of the document. This function opens a file named "PageTable.txt" and appends the document ID and its corresponding URL, separated by a colon and space, to this file. This operation allows the program to maintain a comprehensive record of all documents in the collection, facilitating rapid access to the full content of indexed documents when needed.

4. **Merging:** The program performs batched merge sorting on intermediate index files to manage memory efficiently. In this code, the sorting and merging operations are carried out in the 'file_operations.cpp' file. First, during the batched merge sort, the program segregates the indexed documents into batches, where each batch represents a subset of the entire document collection. These batches are then merged, creating intermediate merged files that contain the sorted and merged data from the individual batches. This step optimizes the process of organizing the data efficiently. Next, in the 'mergeSort' function, two intermediate files are merged at a time, ensuring that the data within them is sorted based on the keys, which are the terms in this case. The resulting merged file contains the indexed information in sorted order. These sorting and merging steps are critical in producing an inverted index that enables fast and efficient retrieval of documents.

5. **Compression:** It compresses the final index data using a variable-byte encoding scheme for space optimization. The compression is achieved through the 'compress' function in the 'file_operations.cpp' file. After the inverted index has been built and saved, the code reads the index from the file and processes it line by line. For each line representing a word and its associated document frequencies, the code tokenizes and extracts the document IDs and their corresponding frequencies. It then encodes these values using a variable byte encoding scheme, a method commonly used for compressing integer values. The compressed bytes for document IDs and frequencies are then written to an output file. The resulting compressed data occupies significantly less space while preserving the essential information needed for efficient retrieval. Additionally, a lexicon file is created to store information about the terms in the index.

Performance:

The program is designed to efficiently process data and create the index files in a reasonable amount of time. In this case it took *4 hours* to completely create all files and compressed binary file. The performance may vary depending on the dataset size and system specifications, but it's optimized for use on macOS.

Index File Size:

Inverted index without compression: 13.74 GB
Inverted index after compression: 8.56 GB

Design Decisions:

The code exhibits several design decisions aimed at achieving a well-structured and functional inverted index, Lexicon and Page table. One key design choice is modularity, as the code is organized into separate header and source files, each responsible for distinct tasks. This modularity enhances code maintainability and readability. Another design decision is error handling; the code provides error messages and checks for file operations, ensuring robustness and fault tolerance. Moreover, the code employs data structures effectively, utilizing hash maps for the inverted index, facilitating efficient word-document mapping. The use of variable byte encoding for compression is another notable design choice, optimizing storage space and retrieval performance. The code also reflects a clear separation of concerns, with functions specializing in distinct operations like indexing, page table creation, and merging. Finally, the code employs hard-coded file paths, which, while pragmatic for smaller-scale projects, might not be the most scalable approach. In summary, the design of this code is characterized by modularity, robust error handling, effective data structures, and a clear separation of concerns, with some scope for improvements in file path management for larger-scale applications.

Limitations:

- The program may not be optimized for extremely large datasets and may require additional memory resources.
- Error handling in the code is minimal, and it may not gracefully handle all possible input scenarios.

Major Functions and Modules:

Here are the major functions and modules in my code:

- **main.cpp:** The main program entry point.
- **indexer.h / indexer.cpp:** Indexing functions and data structures.
- **page_table.h / page_table.cpp:** Page table creation and management.
- **text_processing.h / text_processing.cpp:** Text processing and cleaning.
- **file_operations.h / file_operations.cpp:** File I/O, sorting, merging, and compression functions.

1. **Main Function (main.cpp):** The 'main' function serves as the entry point for the program. It orchestrates the overall workflow, including reading the TREC file, batching, merging, sorting, and compression. It also coordinates the various modules and functions.

2. **File Operations** (file_operations.h and file_operations.cpp): This module contains functions related to file operations. The key functions include 'readTRECFile' for parsing TREC documents, 'mergeFiles' for merging intermediate index files, 'mergeSort' for sorting and merging pairs of files, and 'batchedMergeSort' for batched sorting and merging. Additionally, the module includes the 'compress' function for compressing the inverted index.

3. **Indexer** (indexer.h and indexer.cpp): The Indexer module is responsible for building the inverted index. The 'indexer' function takes a list of words and creates an index that maps each word to the documents where it appears along with the term frequencies. The 'saveIndex' function writes the index data to an output file.

4. **Page Table** (page_table.h and page_table.cpp): The Page Table module handles the creation of a page table, associating document IDs with their corresponding URLs. The 'createPageTable' function appends these mappings to a file named "PageTable.txt."

5. **Text Processing** (text_processing.h and text_processing.cpp): This module is responsible for text preprocessing. The 'sanitizeText' function removes non-alphanumeric characters and ensures uniformity in text by converting it to lowercase.

6. **Compression:** The compression of the inverted index is done in the 'compress' function. It encodes document IDs and frequencies using variable byte encoding, producing a more space-efficient representation.

These modules work in tandem to read, process, index, merge, sort, and compress documents, ultimately creating an efficient inverted index for information retrieval purposes. The modularity and division of responsibilities in the code enhance readability, maintainability, and the ease of future updates or extensions to the system.