**Inverted Index Compression using VarByte**

* Shreyas Kilingaru Nadumanae (sk10696)

**Contents**

The zipped file contains the following files:

1. breakTrecFile.cpp
2. createPostingsForChunks.cpp
3. createSubPostingsFiles.cpp
4. updateSubPostingsFile.cpp
5. mergeSubpostings.cpp
6. compression.cpp
7. readme.pdf

**How to run the program**

To run the program, follow these steps:

* The program is a C++ application and can be compiled and executed on a C++ compiler (e.g., g++).
* The path to the input .trec file and the desired number of documents per output file should be specified within the code of the breakTrecFile.cpp file. The file execution flow is mentioned below.

**Brief summary**

The programming language used for this assignment is C++. There are a total of 6 .cpp files present and each performs its own tasks. A detailed explanation of these files is given in the next section.

The execution flow for the files should be in following order:

**breakTrecFile.cpp:** Reads the .trec file and breaks it into 2142 files of 1500 documents each. These files are stored in the ‘data’ folder. Naming convention followed: output\_0, output\_1….. Each file is approximately 10-11MB.

**createPostingsForChunks.cpp:** Fetches all the files from the ‘data’ folder. It creates a Postings file for each of the files in the folder. All these Postings files are stored in the ‘unfilteredPostings’ folder. The Postings files follow the same naming conventions as the chunks. Eg: output\_0 file will have a postings\_0 file, output\_1 will have a postings\_1 file and so on… Each file is about 8-9 MB.

**createSubPostingsFiles.cpp:** Fetches all the files from the ‘unfilteredPostings’ folder. Performs merges for the all the Postings files and writes each index to its respective Postings\_Alphabet file. Program stores all these Postings\_Alphabet files in the ‘sortedPostingsAlphabetically’ folder. For example, all words starting with 'a' will be present in postings\_a file and words starting with 'b' will be present in postings\_b file and so on.

**updateSubPostingsFile.cpp:** Fetches files from the ‘sortedPostingsAlphabetically’ folder. It merges all the same words into 1 posting and stores the data back into those respective files. The file size here varies from 200MB (postings\_u.txt) to 1.5GB (postings\_s.txt).

**mergeSubpostings.cpp:** Fetches files from the ‘sortedPostingsAlphabetically’ folder. It merges all the postings\_alphabet files to one file postings\_inverted.txt and saves this in the ‘mergePostings’ folder. This file has the all the inverted indexes of all the words. It also creates a hashtable from the postings\_inverted file in the ‘mergePostings’ folder and saves it in the ‘hashtable’ folder in a hashtable.txt file. The postings\_inverted file present here is about 16.7GB. The hashtable formed is just over 1GB in size.

**compression.cpp:**  Fetches the postings\_inverted.txt file from the ‘mergePostings’ folder. It performs varByte encoding on all the numbers (docID and Frequencies) and stores this in .bin file in the compressed format. The compressed file is 8.53GB.

**Detailed explanation**

**1) breakTrecFile.cpp**

This program processes a large .trec file containing documents and splits these documents into smaller text files. It reads the input file line by line. When it encounters "<DOC>" and "</DOC>" tags, it identifies the start and end of a document.

**What the program can do:**

1. Reading a large .trec file.
2. Dividing the documents into smaller text files, with 1500 documents in each output file.
3. Creating a "data" folder to store the output files, in case it doesn't already exist.
4. Writing each batch of documents to a new text file within the "data" folder.

**How it works internally:**

1. It maintains a vector called **documents** to temporarily store lines belonging to a document.
2. It tracks whether it is inside a document with the **isInsideDocument** boolean variable.
3. When it encounters "<DOC>", it sets **isInsideDocument** to true and starts adding lines to the current document.
4. When it encounters "</DOC>", it increments **documentCount** and checks if the specified number of documents per output file (**documentsPerFile**) is reached.
5. If the count is reached, it calls the **writeDocumentsToFile** function to write the documents to a new text file, resets the temporary storage, and increments the **fileIndex** for naming the output files.
6. After reaching the end of the file, it checks if there are any remaining documents in the **documents** vector. If so, it writes them to a file.
7. Finally, the program closes the input file and returns.

**Major Functions and Modules:**

1. **writeDocumentsToFile**: This function writes a batch of documents to a new text file in the "data" folder. It handles file creation and writing operations.
2. **main**: This is the main program logic. It reads the .trec file, accumulates documents, and calls the **writeDocumentsToFile** function when the specified batch size is reached or at the end of the file.

**2) createPostingsForChunks.cpp**

This program is written for processing a collection of documents from the ‘data’ folder and generating posting lists.

**What the program can do:**

1. **Document Processing:** The program reads a collection of text files, and extracts information from these files. It collects URLs, processes the document text, and builds posting lists for terms found within the documents.
2. **Posting List Generation:** It generates posting lists for each term encountered in the documents. A posting list records which documents contain a particular term and how many times the term appears in each document.
3. Each file present in the ‘data’ folder creates a new Postings files. For example, output\_0.txt (holds one chunk of 1500 documents) stores it’s postings in postings\_0.txt file. Similarly, output\_1.txt will have its own postings\_1.txt file.

**How it works internally:**

1. The program parses text files by iterating through each line. It extracts the URLs and document content from within **<TEXT>** and **</TEXT>** tags.
2. It processes the text content by splitting it into words based on the spaces, removing unwanted characters, and then adding the words to posting lists, associating them with the respective document IDs. Document IDs are numbered from 0 to n depending on when the document was parsed. It also provides a function, **removeCharactersFromBothEnds**, which removes specified characters from the beginning and end of a string. This function is used to clean up words for processing.
3. The program organizes the posting lists in memory and writes them to text files within an "unfilteredPostings" directory. This directory has separate Postings files for each of the chunks that were previously formed. It also maintains a mapping of document IDs to URLs and writes it to "docIDToUrlMapping.txt".

**Major Functions and Modules:**

1. **removeCharactersFromBothEnds**: Removes specified characters from the beginning and end of a string. It used for cleaning the words.
2. **addToPostingList**: Adds an entry to the posting list for a term and document ID.
3. **writePostingListsToFile**: Writes the posting lists to text files.
4. **outputDocIDToUrlMapToFile**: Writes the document ID to URL mappings to a file.
5. **skipWord**: Determines whether a word should be skipped based on specific condition.
6. **parseTxtFile**: Parses a .txt file, extracting URLs and processing document content to create posting lists.
7. **extractNumber**: Extracts a number from a directory entry's filename so that a corresponding postings file can be generated.
8. Top of Form

**3) createSubPostingsFiles.cpp**

This program performs an external merge sort on the collection of the postings files. It aims to sort the posting lists alphabetically and distribute them into separate output files based on the starting letter of the terms.

**What the program can do:**

1. **External Merge Sort:** The program is designed to sort and merge posting lists. It performs an external merge sort. It makes use of the concept of min heap. A line from each of the files is pushed to the heap, and then a minHeap is performed. The resultant min value is then placed into its respective file.
2. **Distribute Posting Lists:** The sorted posting lists are distributed into separate output files based on the starting letter of the terms. For example, if a word starts with a, then it is placed into a postings\_a.txt file and so on.

**How it works internally:**

1. The program defines a custom data structure, **LineWithIndex**, which holds a line of text and the index of the input file it belongs to.
2. It also defines a custom comparison function, **lineWithIndexCompare**, used for sorting lines in a priority queue. The priority queue is used to perform the external merge sort.
3. The main function, **externalMergeSort**, is used for sorting and merging the posting lists. It uses a priority queue (min-heap) to merge lines from multiple input files. The sorted lines are written to the corresponding output files based on the starting letter of the terms.
4. The program maintains a list of input files and initializes the priority queue with the first line from each input file.
5. The program determines the starting letter of each line, assigns it to the appropriate output file, and appends the line to the output file.
6. The program continues this process until all lines from the input files are processed and merged into the output files.

**Major Functions and Modules:**

1. **externalMergeSort**: The main function responsible for external merge sorting and distributing the posting lists into output files.
2. **LineWithIndex** and **lineWithIndexCompare**: Custom data structure and comparison function for efficient sorting of lines during the external merge.

**4) updateSubPostingsFile.cpp**

This program is used to update sorted posting list files. It cleans and organizes the posting list data, removing specific characters, and then rewrites the cleaned data back to the original files.

**What the program can do:**

1. **Data Cleaning:** The program cleans the posting list data by removing specific characters from terms.
2. **Merging of Postings:** It clubs the same words into the same lines and appends their document frequency pairs. Before this code is run, each of the postings\_a.txt, postings\_b.txt and so on, can have multiple lines with the same word. After this function, all those words would be club into one posting. For example, let’s take the word ‘armadillo’. This could have a postings that looks like this:
   1. **armadillo –** (0,5) (1,10) //From file 1
   2. **armadillo –** (245, 10) (444, 11) // From file 2

The code in this file, merges the above and forms a: armadillo – (0,5) (1,10) (245, 10) (444, 11)

1. **File Updating:** After cleaning and organizing the data, the program updates the original sorted posting list files with the cleaned data.

**How it works internally:**

1. It defines a function, **updateEntries**, which processes each line of the posting list files. It extracts the word, cleans it, and adds the document frequency pairs to the data map.
2. The program then iterates through the files specified in **categoryToFileName**. For each file, it reads each line and uses the **updateEntries** function to update the data map.
3. After processing the entire file, it opens the file for writing, and the cleaned data is written back to the file.

**Major Functions and Modules:**

1. **cleanWord**: A function that cleans a word by removing specific characters.
2. **updateEntries**: A function that updates the data map with cleaned data from each line in the posting list file.

**5) mergeSubpostings.cpp**

This program performs operations to organize and process data from the sorted posting list files. It involves merging sorted posting lists, creating a hashtable from the data, and saving the hashtable to a file.

**What the program can do:**

1. **External Merge Sort:** Like before, the external merge sort is used again here. It performs an external merge sort on a set of sorted posting list files (postings\_a.txt, postings\_b.txt and so on), merging them into a single output file (postings\_inverted.txt).
2. **Creates a Hashtable:** The program also creates a hashtable from the merged posting list data. The hashtable associates words with pairs containing their frequency and line number within the posting lists.
3. **Save Hashtable to File:** The hashtable is then saved to a text file for future retrieval and analysis.

**How it works internally:**

1. **External Merge Sort:**
   * The program starts by defining a custom structure, **LineWithIndex**, to hold lines and their corresponding file index.
   * It sets up a priority queue (min-heap) to facilitate external merge sorting.
   * The input files are opened and initialized, and their first lines are added to the min-heap.
   * The program proceeds to merge the input files by repeatedly taking the top line from the min-heap, determining the output file index based on the starting letter of the line, and appending the line to the corresponding output file.
   * This process continues until the min-heap is empty.
2. **Create Hashtable:**
   * After the external merge sort, the program reads the merged data from the output file.
   * It processes each line to extract words and their frequency and line number. This data is stored in a hashtable.
3. **Save Hashtable to File:**
   * The hashtable data is then sorted alphabetically.
   * It is then written to an output file, "hashtable.txt," where each line contains a word along with its frequency and line number.

**Major Functions and Modules:**

1. **externalMergeSort**: Performs external merge sort on the input posting list files and merges all the files into a postings\_inverted.txt file.
2. **createAndSaveHashtable**: Creates a hashtable from the merged posting list data and saves it to a text file.

**6) compression.cpp**

This is the final program that is run on the dataset. It performs VarByte encoding on posting list data (document IDs and frequencies).

**What the program can do:**

1. **VarByte Encoding:** The program reads posting list data from the postings\_inverted.txt file and encodes the integers (document IDs and frequencies) using the VarByte variable-length encoding scheme. It saves the encoded data in a binary file.

**How it works internally:**

1. **VarByte Encoding:**
   * The program uses the VarByte encoding scheme to compress integers.
   * It iterates through each line of the input file, which contains words, document IDs, and frequencies in a specific format (e.g., "word: (docID, frequency)").
   * For each line, it extracts the word and the associated (docID, frequency) pairs.
   * The program processes each (docID, frequency) pair, encoding them using the VarByte encoding scheme.
   * Encoded docIDs and frequencies are appended to a binary output file, with delimiters between different parts.
   * The program repeats this process for each line in the input file, encoding the entire dataset.
   * The encoded data is saved in binary format, with each word and its encoded (docID, frequency) pairs on separate lines.

**Major Functions and Modules:**

1. **varByteEncode**: Performs VarByte encoding of an integer and returns its binary representation.
2. **main**: Reads the posting list data from a text file, parsing it to extract words, document IDs, and frequencies. Encodes the document IDs and frequencies using VarByte. Writes the encoded data to a binary output file, preserving the structure of the original data.