Web Search Engine Homework 2: InvertedIndex

Author: Sheng Wang

Data: 10/14/2021

1. Program Overview

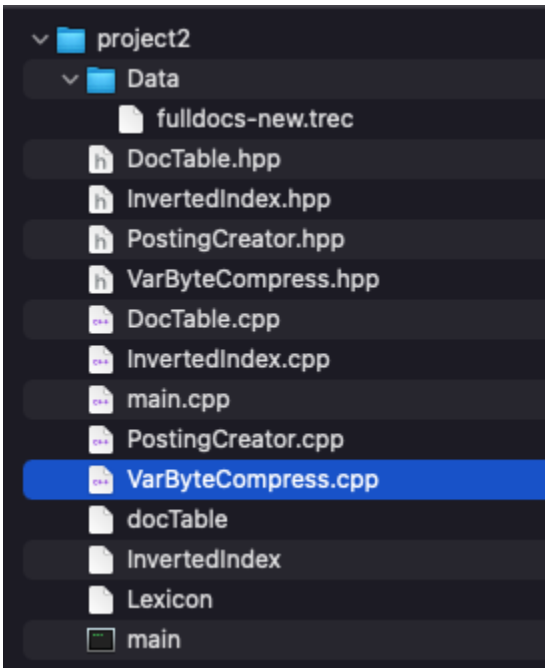
The purpose of this program is to build an inverted index structure from a provided dataset. There are three major steps the program takes to create such structure:

First, the program reads in the dataset and parses the content to create intermediate postings for each document and writing them to temporary files. Second, the program sorts each temp file and merges them into one large sorted file. Third, the program reads in the merged file and creates an inverted list for each term and writes them to disk.

There will be three files generated in the end. The first file is the docTable which records each document's id, link and size, and it is created during the dataset parsing. The second one is the lexicon and the third one is the InvertedIndex. They are both created in the last step where lexicon records the term along with its position in the inverted lists while InvertedIndex keeps the actual content of the list.

2. Compile and Run

- Below are my project structure



- To compile the program: open the terminal and go to the directory where the main project is. Type command:

```
g++ -std=c++17 -O3 ./main.cpp ./PostingCreator.cpp ./DocTable.cpp ./InvertedIndex.cpp ./VarByteCompress.cpp -o main
```

- To run the project: In the project directory, type the following command:
    - ./main -s yourSourcePath
    - For example, here is how I run the program:

```
(base) sheng:project2 shengwang$ g++ -std=c++17 -O3 ./main.cpp ./PostingCreator.cpp ./DocTable.cpp ./InvertedIndex.cpp ./VarByteCompress.cpp -o main
(base) sheng:project2 shengwang$ ./main -s ./Data/fulldocs-new.trec
The path of the source is ./Data/fulldocs-new.trec
```

3. Program Design and Description

The program is designed to have four components where each component handles different jobs. The main.cpp is used to parse user input and call each component.

- **PostingCreator.cpp & PostingCreator.hpp**
    - **What it does:**

      Read in dataset, create intermediate posting for each document, write postings to temporary files.
    - **How it works:**

      In the function createPosting, it opens the dataset and reads it line by line in a while loop. The function uses a vector of posting struct (term, docid, frequency) to keep track the postings in one document. Since we do not know where one document starts and ends, we will check for 3 conditions and use a stack to keep track of the documents.

      In the loop, we check if the current line is tag <TEXT>, if so, we push 1 to the stack, update document Id and start counting documents size. We also check if the current line is </TEXT>, if so, we pop the stack, sort the vector of postings by term, and merge postings with the same term.

      After that, we write the postings to a large sized buffer, and if the buffer is full, we call the function flushBuffer to write the buffer to a temp file. At last, we reset the vector and other variables for the next document. If the current is neither <TEXT> or </TEXT>, and the stack is not empty, it means we are paring the text of the document. So for each line, we use

another loop to find the proper word character by character. For each character, we first check if it is one of the delimiters which is a predefined constant, if not, we check if the character is either a number or an alphabet, if so, we push the char into a word string. If a character fails the check, it means the end of a word, then we will lowercase the word and create a posting of the word and push to the vector.
After the execution of createPosing, there will be several temp files of the buffer size. Then the program will call SortAndMerge, in which we will sort each temp file first by term, then by docId. Then we will merge these temp files into one large file. The program uses unix command sort to achieve that.

- **DocTable.cpp & DocTable.hpp**
    - **What it does:**
      Create an entry for each document, and write entries to the disk.
    - **How it works:**
      For a DocTable object, it has a vector to hold each doc entry. For each entry, it records the document id, the link of the document, and the size of the text body. At the beginning of the parsing in PostingCreator, the program will call the constructor of DocTable. At the end of parsing, the program will call the function writeTable in PostingCreator to write all the entries to the disk.

- **VarByteCompress.cpp & VarByteCompress.hpp**
    - **What is does:**
      Using varbyte compression to compress an integer array, and decode an compressed array.
    - **How it works:**
      This file consists of 4 functions, encode, encodeNum, and writeByte are used to compress integer arrays. And decode is used to uncompress a

compressed array. The algorithm is from the lecture notes. For compressing an integer array, we first pass the reference of the array to function encode which returns a vector of unsigned char. In function encode, we loop through theint array, and encode each number using encodeNum and then concatenate the result. writeByte is used to write number that is smaller than 256, and it uses uint8_t to make sure using only 8 bits. The function decode uses the same logistics from the one in the slide, it reads in the unsigned char array, and loops through the array to decode each number.

The functions in VarByteCompress.cpp are called in InvertedIndex.cpp after we create one block in the inverted list for one term. Compression is applied on the docid array and the frequency array, and also applied on arrays in metadata.

- **InvertedIndex.cpp & InvertedIndex.hpp**
  - **What it does:**

    Read in merged posting and build inverted list for term, build lexicon, write both invertlists and lexicon to disk
  - **How it works:**

    This class is used to create inverted indexes and the lexicon. For the lexicon structure, the class uses a vector to store the info of each term. For each term, the program will records the term string, the start position in invertedindex, the end position in invertedindex, and the number of postings the list contains.

    The function buildInvertedIndex is the main function to create inverted lists. It starts by reading in the mergedPostings file while opening a oftream to write. The function extracts the term, docid and frequency of the first line as the TERM to compare. Then it continues to read the next line and compares that line's term to TERM, if the terms match, it pushes

the docid and frequency to their corresponding vector; if they do not match, it means all the docIds and frequency for that TERM are parsed, so the program goes into a loop and start construct the inverted list.

The algorithm to construct an inverted list for one term is following:
We have two vectors of the same size that contain the docid and frequency. And for one block we will include 64 docids and 64 frequencies.

First, we set up two unsigned char vectors: one for the final list, one for concatenating blocks. Second , we set up two int vectors for metadata: one for block size, one for last id. Third, we loop through the docids and frequency array, for every 64 docids and frequencies, we use functions in VarByteCompress.cpp to encode the ids and frequency, then we push back the results to the block array, and update the metadata: to add the size of the current block, and to add the last id of the current block. Finally, after every id and frequency is processed, we compress the metadata, and we concatenate the metadata and the blocks to form the list. Then we write the list to the output file, get its current stream position, and update the lexicon. After we parse the mergedPostings for every term, the program will call writeLexcion to write the entries to the disk.

4. Test and Performance

I usually test on different parts. For PostingCreator and docTable, I first test if the program can read through the whole data set: I look at the number of temp files it creates and multiply the buffer size to see if it is close to the original dataset. Then I looked at the last docId in docTable and matched it with the last docId in the last temp file and see if it is close to the number on the dataset website. To test the correctness of postings, I read in only one or two documents and print each word it added to see if it is proper. To test the varbyte compress, I wrote a test to encode an integer array, and then decode the result to see if the decoded array matches the original array. To test if the buildInvertedlist

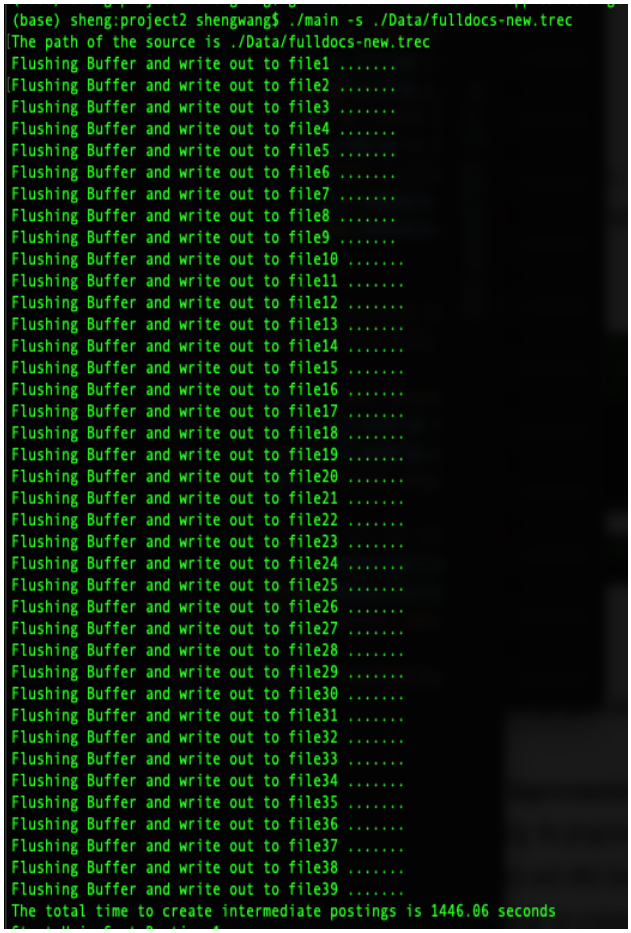
compresses the correct number of blocks, I print the posting number and the number of blocks to see if ceiling(numPosting/blocknum) = 64.

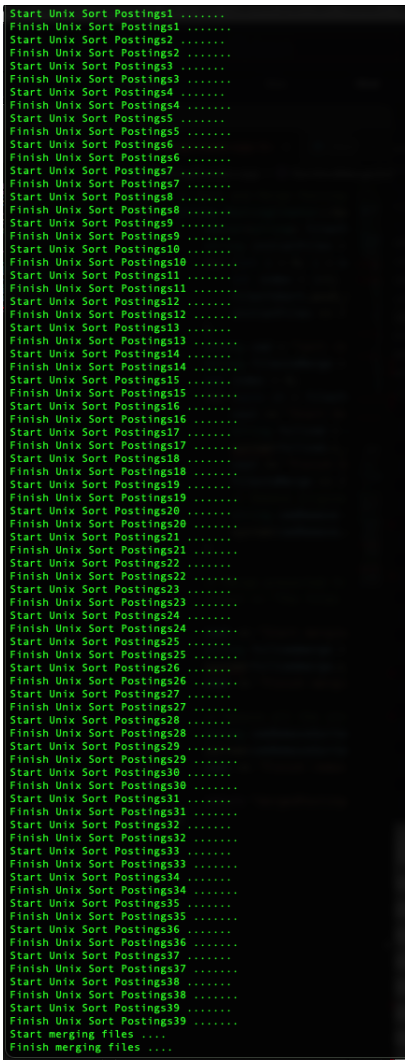
Run Time for 10000 documents:

```
(base) sheng:project2 shengwang$ ./main
Flushing Buffer and write out to file1 .......
The total time to create intermediate postings is 4.78407 seconds
Start Unix Sort Postings1 .......
Finish Unix Sort Postings1 .......
Start merging files ....
Finish merging files ....
Finish removing Sorted Postings .........
```

```
The total time to create invertedindex and lexicon 5.962 seconds
The total time is 10.7476 seconds
```

Run Time for the Whole dataset:





```
The total time to create invertedindex and lexicon 2590.63 seconds
The total time is 4036.73 seconds
```

- And the size of the file get generated are the following:

5. Limitations and Possible Improvements

- For document parsing: the program only keeps numbers and english characters. For special characters and other languages, it skips over. The program can be improved using unicode for content parsing.
- The unix sort and merge runs quite slowly as the intermediate file size increases with limited disk space and ram on my computer.
- The parsing really relies on the structure of the input the data. For example, since every doc url is in the first line of text, I simply extract the first without parsing. But if the format of input data changes, this parser will not work.