

Web Search Engine

Homework 3

Yiqiu Wu/yw3101

Index

1.How to run this program.....	2
2.How it works internally.....	4
3.The result.....	8
4.Limitations.....	9
5. Major functions and modules.....	10

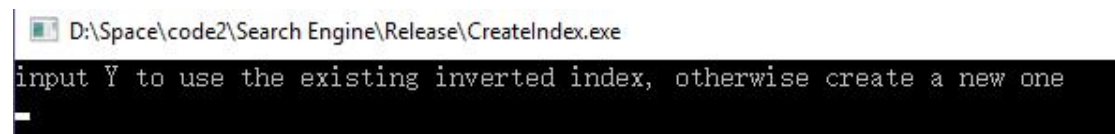
1.How to run this program

This program is written in C++ on Visual Studio 2017 on Windows 10. It is recommended to use Visual Studio to open this whole project.

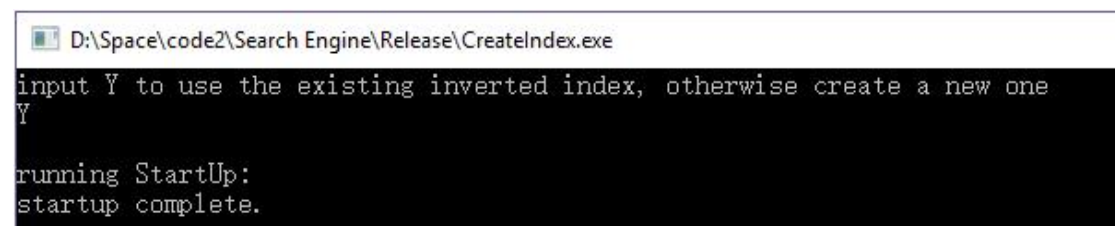
Before running this program, please check if the program is running in Release Mode, which will greatly improve the running speed of this program. Also please make sure there is enough stack for this project(The stack reserve size is already set to be a big number in project setting).

Original crawling files must be uncompressed WET files from "http://commoncrawl.org". There is no limit on the number and the name of those files. To create new index, need to put those files into "CreateIndex/pages/" directory. And most importantly, make sure there is no files in all directories with name like "CreateIndex/postingsX/", especially after your last use.

When running the program, follow the screen guide.
First, you will be asked whether to use existing index.

A screenshot of a Windows command prompt window titled "D:\Space\code2\Search Engine\Release\CreateIndex.exe". The prompt displays the text "input Y to use the existing inverted index, otherwise create a new one" followed by a cursor on a new line.

1. If you input "Y", then the program will read files in "CreateIndex/index/" directory, and put lexicon table and document table into memory. This StartUp process would take a few seconds.

A screenshot of the same command prompt window. It shows the prompt "input Y to use the existing inverted index, otherwise create a new one" with the letter 'Y' entered. Below this, the program outputs "running StartUp:" and "startup complete." on separate lines.

2. If you input other word, then the program will start creating inverted index based on the files you put into the "CreateIndex/pages/" directory. Processing sequence is first creating postings files, then merging files and creating final inverted index files. The whole process takes a lot of time, be careful.

After 1 or 2 selection, you can start your query.

First, you need to choose search mode between conjunctive search and disjunctive search.

A screenshot of the command prompt showing the prompt "please input search mode("and" conjunctive "or" disjunctive otherwise quit)." with the word "and" entered on the line below.

Then, you can input a list of search terms which should be parsed by space. Pay

attention that max number of search terms is 20.

```
please input search terms.    //use Q to quit
computer science
```

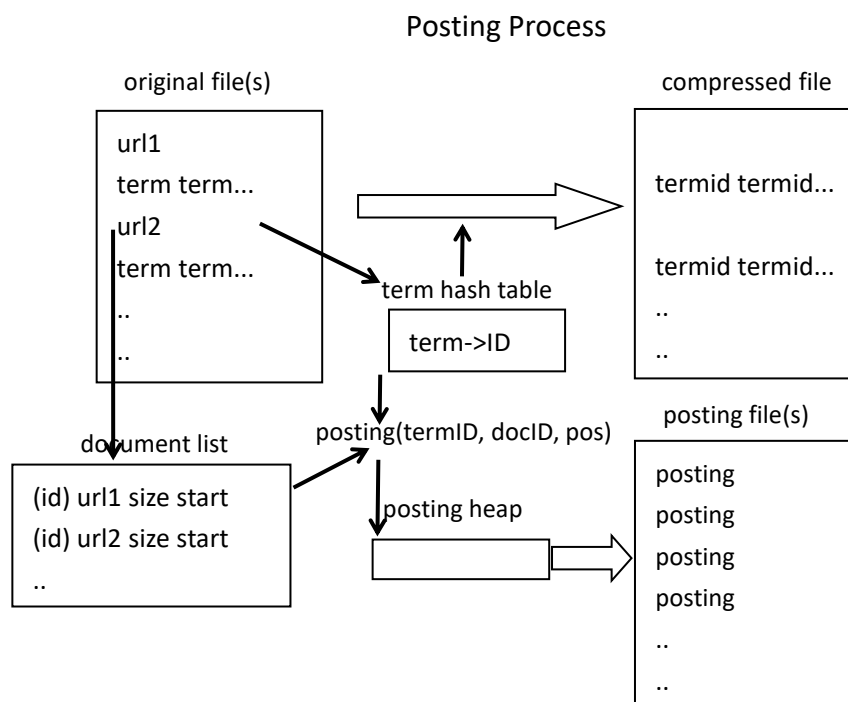
Top 10 high score results will show their url, frequency and score on screen with their snippet text with length of about 10 words.

```
please input search terms.    //use Q to quit
computer science
1.http://onyhoveb.keep.pl/computer-science-research-titles.php totalFreq: 88 score:13.2074
Snippet text: computer science research titles bc boat
Snippet text: computer science research titles bc boat in
2.http://essays.essayon.xyz/fox/computer-thesis/ totalFreq: 86 score:13.0154
Snippet text: computer thesis esl essays respect to
Snippet text: sciencin computer science some theses involve building systems
3.https://duq.edu/academics/schools/liberal-arts/departments-and-programs/math-and-computer-science/computer-science-ba
/bs totalFreq: 39 score:13.0051
Snippet text: major in computer science duquesne university play twitter
Snippet text: in computer science duquesne university play twitter facebook
4.http://cs.highpoint.edu/programs.php?f=programs&month=4&year=2021&day=3 totalFreq: 30 score:12.997
Snippet text: programs mathematics computer science high point university high
Snippet text: mathematics computer science high point university high point
5.https://www.gradschools.com/doctorate/computer-science/on-campus?countries=canada totalFreq: 70 score:12.973
Snippet text: computer science doctorate campus degrees graduate
Snippet text: computer science doctorate campus degrees graduate programs
6.https://cims.nyu.edu/people/profiles/KOSKINEN_Eric.html totalFreq: 22 score:12.8836
Snippet text: institute mathematics computer science center for atmosphere ocean
Snippet text: mathematics computer science center for atmosphere ocean science
7.http://cs.highpoint.edu/programs.php?f=ba_csreq&month=6&year=2020&day=3 totalFreq: 17 score:12.8407
Snippet text: computer science mathematics computer science high
Snippet text: computer science mathematics computer science high point
8.http://cs.highpoint.edu/programs.php?f=ba_csreq&month=8&year=2019&day=7 totalFreq: 17 score:12.8391
Snippet text: computer science mathematics computer science high
Snippet text: computer science mathematics computer science high point
9.http://csinquiry.org/apcsp/postlist.html totalFreq: 11 score:12.8311
Snippet text: post list computer science academy galileo toggle navigation
Snippet text: list computer science academy galileo toggle navigation computer
10.http://cscproject.org/e-newsletter totalFreq: 36 score:12.8141
Snippet text: the computer science collaboration project login register
Snippet text: the computer science collaboration project login register cscproject
```

After that you can input other search terms, or return to choose search mode.

2.How it works internally

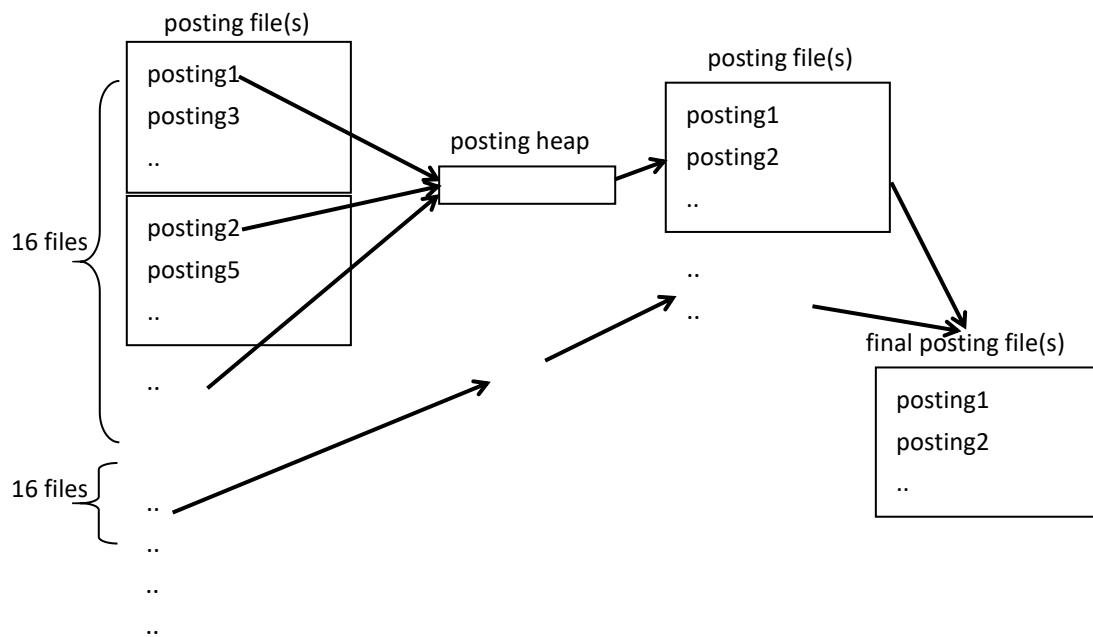
If you choose to first create inverted index based on original_page files, the program will get a filename list of them. The program works on those files one by one. The current file is bound with a reader which has a 5 MB buffer. The program reads each file through buffer, and parses terms out. English terms are hashed so that the program can get whether the term is met before. Also, documents information such as url and length is stored in memory. termID and documentID both depend on their appearing order. Parsed terms are transformed into posting(termID, docID, position) and pushed into a posting heap. The heap sorts those postings by their termID and docID. Each time the heap size reaches a certain number, it will be output to a intermediate posting file. The file writer has a 20MB buffer. Besides, all the postings are compressed by calculating difference and vbyte. Those intermediate files are in binary format. At the same time, original_page files are changed into termID version and output to be a compressed_page file which is used for showing snippet text in query process. At the end of this posting creating, there will be several posting files in "CreateIndex/postings/", a compressed_pages file in "CreateIndex/index/". In memory, there is a document table with doc info and doc position of compressed_pages file, a lexicon table with term string and termID.



When it comes to merging posting files, if the number of files is smaller than 16, the

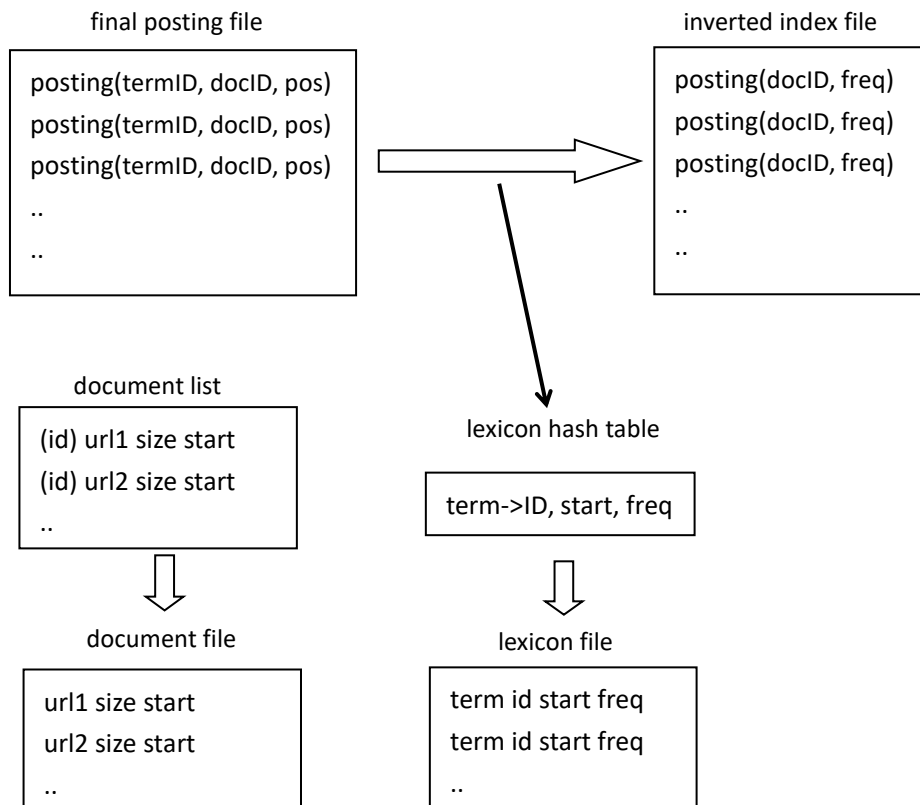
merging process will take one pass to merge them all, otherwise, it need more passes. Each time the program merges 16 files, it will open 16 file readers with each file and a file writer. First the program uncompressed the first postings of each file and push them into a temporary heap. Heap's smallest posting will be popped out and the next posting from the same file will be pushed in. When the heap is empty, this turn of merging is over. Finally, we can get a complete sorted posting file. Of course, those files are also be compressed and in binary format.

Merge Process



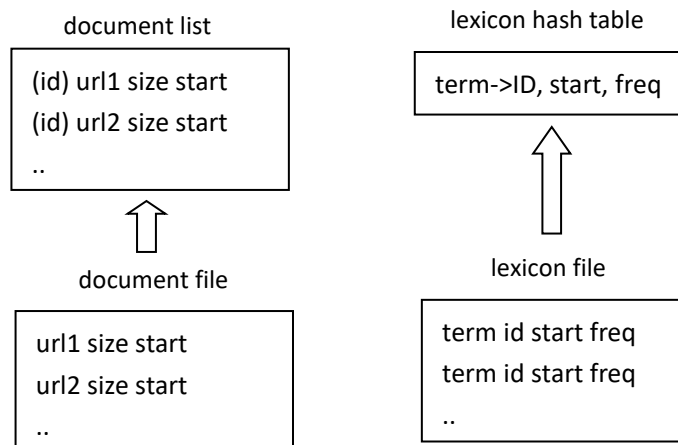
Now it's time for creating index. The program reads the postings(termID, docID, position) from the complete sorted posting file, and count the frequency of term in each doc. New generated postings are in format like (docID, freq). Each Term's startpoint and total frequency is stored into lexicon table. New postings are compressed and saved into "CreateIndex/index/inverted_index" in binary format. Document table(url, size, start) are saved into "CreateIndex/index/documents". Lexicon table(term, termID, start, freq) are saved into "CreateIndex/index/lexicon".

Creating index Process



If you choose to use existing index when the program begins, the files used are exactly the upper generated different kinds of index files. Lexicon table and document table will be read into memory. Compressed_pages file and inverted index file will be used in the following query process.

Start Up Process



When processing query, The program first splits the query sentence into terms. Depending on whether query mode is conjunctive or disjunctive, the program will choose different algorithm to process the query. The main idea is to use DAAT(Document-At-A-Time). The program will open a inverted list for each term. If query mode is AND, the program finds the smallest docID in all the lists, computes its doc score by BM25 and put it into a heap of size 10. If query mode is OR, the program finds the smallest docID in either one of all the lists. Then program will find next smallest docID until all the lists are empty. Top 10 results will be printed with their url, freq and score. Snippet text are found by looking for search terms inside that doc part in compressed_pages file.

3.The result

Original files: 50 WET files with total size of nearly 15 GB.

Processing time: 53 minutes in total.

CreatePosting takes 13.5 minutes.

MergePosting takes 27 minutes, due to the program using 2 passes to merge the files.

CreateIndex takes 12.5 minutes.

StartUp takes about 8 seconds.

Every query takes about 1 second to show top 10 result urls and their snippet text.

Result files Size:

Intermediate posting files are about 6.67 GB large each pass.

Documents file is 147 MB, containing about 1.6 million pages.

Lexicon file is 214 MB, containing about 6.5 million lexicons.

Inverted index file is 842 MB.

Compressed pages file is 1.81 GB.

4.Limitations

need to download and uncompress files on your own

only accept WET files

need to make sure that “postingsX” directories are empty before using program to create index

no pipe between creating final posting file and creating index

no chunkwise compression

no caches (query processing is fast enough)

The program is composed of several main classes.

5. Major functions and modules

CreateIndex.cpp

Main function is in "CreateIndex.cpp". It contains some simple codes to make use of the main manager class MyPageReader. It calls major step functions in MyPageReader in sequence to realize the program.

class MyPageReader

It is responsible for all the major step functions in program.

StartPosting():

It uses **Getfiles()** to put all the original file names into a file list. It uses **LoadFile()** iterates the list, and creates a **MyBufferReader** for file reading. It also creates two **MyFileWriter**, one for compressed_pages file, one for intermediate postings file. The reader reads through the file so that the file parser can extract all the documents information and terms. DocIDs and termIDs are both assigned in the appearing order. Those documents information are stored in a list in **InvertedIndexManager**. All the terms are hashed into lexicon table in **InvertedIndexManager**, be transformed into **RawPosting** format and be put into a postingHeap which is a priority_queue type structure. Each time postingHeap is full, it calls **PostingHeapout()** to output the content of the heap into posting file writer. As for compressed_pages file writer, it gets termID version of all the original files.

StartMerge():

Each time it opens 16 **MyFileReader** to read 16 posting files and opens a **MyFileWriter** to write a intermediate posting file. The very first postings of each files are pushed into a postingHeap. It will pop out the smallest posting and push a new posting until all the readers are empty. It also controls how many passes to merge.

StartCreateIndex():

it opens the final posting file with a **MyFileReader**, creates a **MyFileWriter** to output the final index and creates two **MyBufferWriter** to output the lexicon table and document table. When doing the reading, all the postings are transformed from **RawPosting** to **FinalPosting**. Each lexicon's start point and frequency are stored in **InvertedIndexManager**. When it comes to write lexicon file and document file , lexicon hash table and document list in **InvertedIndexManager** will be output.

StartUp():

it opens the lexicon file and document file with two **MyFileReader**, and put them into InvertedIndexManager.

StartQuery():

It splits search terms and puts them into a string list. For each term in list, it uses **openList()** to create a inverted list pointer list. It uses **nextGEQ(lp,k)** in order for each list to find the next posting in list lp with docID $\geq k$ and return its docID. When it finds the target document, it uses **getFreq()** to get the frequency of the document, and uses **ComputeBM25()** to get the doc score. docIDs and their scores are pushed into a scoreHeap which will pop out one element if size reaches 10. Finally we can

get 10 docIDs and their scores. Based on document information in **InvertedIndexManager**, it gets the document content from compressed_pages file and find snippet text around the search terms.

class MyFileReader: A binary file reader which can read certain position of a file. It has a 5 MB buffer. Other class can fetch content from its buffer. Those content will be uncompressed through VAR-BYTE. It is used to read postings files, inverted index file, compressed_pages file.

RefreshBuffer(): It reads content from its file until the file is empty.

GetStruct(): outputs a structure to other classes. if the buffer is empty, calls **RefreshBuffer()**.

class MyFileWriter: a binary file writer. It has a 20 MB buffer. Those content will be compressed through VAR-BYTE. It is used to write postings files, inverted index file, compressed_pages file.

RefreshBuffer(): writes the buffer into its file and clear the buffer.

SetStruct(): sets a structure into the buffer. If the buffer is full, calls **RefreshBuffer()**.

class MyBufferReader: similar to MyFileReader, but it can only handle string type. It is used to read the original_pages file, lexicon file and document file.

class MyBufferWriter: similar to MyFileWriter, but it can only handle string type. It is used to write lexicon file and document file.

class MyHashTable: A lexicon table created in memory when creating inverted index.

Set() & Get() functions: used to manage the lexicon table.

HasLexicon(): to check if a lexicon is in the table.

addTerm(): to add a term and give it a ID by appearing order.

class MyInvertedList: stores inverted list from **openList()**.

NextPosting(): to help nextGEQ() to get next posting.

“VarByte.h”: contains some functions about VarByte compression and uncompression.

“sturctures.h”: contains basic structures such as **Document**(url,size,start), **RawPosting**(termid,pageid,position), **FinalPosting**(pageid,frequency), **Lexicon**(termid,start,frequency)

other code details are in comments.