

1. Introduction

This generator has two main functions: generating inverted index list using Common Crawl data set and executing query search using inverted index list. The development environment is OS X or Linux.

2. How to run this program

Pre-requisites: Java 7, mongoDB, node, maven. MongoDB should be started before running this program.

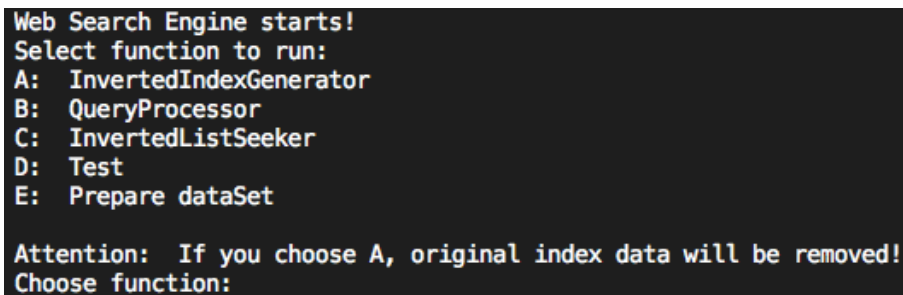
First, go to directory of the package, build the project using Maven, in terminal:

```
cd path/to/your/InvertedIndexCreator
mvn install
```

Then, prepare the dataset using shell “initialize.sh”, which will download and decompress the common crawl data automatically. In terminal:

```
mvn exec:java
```

You will see:



```
Web Search Engine starts!
Select function to run:
A: InvertedIndexGenerator
B: QueryProcessor
C: InvertedListSeeker
D: Test
E: Prepare dataSet

Attention: If you choose A, original index data will be removed!
Choose function:
```

Pic 1.1 Interface

Input E, and after it finishes, you will see the common crawl data in the “input” directory.

To start generating inverted index list, again, in terminal:

```
mvn exec:java
Type A
```

It will take some time to generate this index list. We will talk about run speed details in other section.

After it finishes, you will see three files: `lexicon.txt`, `pageUrlTable.txt` and `invertedIndexList.txt`.

Here we come to the query processing part. In terminal:

```
mvn exec:java  
Type B
```

It will take some time to load the data, and when it finishes you would see:

```
[INFO ] 2017-11-03 20:44:21.158 [main] QueryProcessOrchestrator - Loading lexicon...  
[INFO ] 2017-11-03 20:44:37.737 [main] QueryProcessOrchestrator - Lexicon loaded!  
[INFO ] 2017-11-03 20:44:37.737 [main] QueryProcessOrchestrator - Loading pageUrlTable...  
[INFO ] 2017-11-03 20:44:38.639 [main] QueryProcessOrchestrator - PageUrlTable loaded!  
The server is running
```

Pic 1.2 Server running

Now it's time to query, go to "frontEnd" directory, in terminal:

```
cd frontEnd  
npm install  
npm run dev
```

You would see:



Pic 1.3 Search interface

Type in the words you would like to search, the rule is that "&" represents conjunctive search and "@" means disjunctive search. Click search button or press enter, you would see the result in XML format:

```

▼<docId>
  196840
  <bmValue>20.54378606874033</bmValue>
  <freq>dog : 28; cat : 29; bird : 27;</freq>
  ▼<url>
    http://providence.backpage.com/CleaningServices/cranston-junk-remo
  </url>
  ▼<snippets>
    ...anup-rat feces cleanup-rodent feces cleanup-raccoon feces clean
    feces clean up service-rat feces clean up-rodent feces clean up-ra
    droppings clean up contractors dog droppings clean up squirrel dro
  </snippets>
</docId>
▼<docId>
  28579
  <bmValue>18.15781737702905</bmValue>

```

Pic 1.4 XML search result

You can also just type <http://localhost:8888/query?dog&cat&bird> in your browser to get the result directly.

3. Main components description

3.1 Inverted index generator

3.1.1 Index structure

Lexicon: word, start byte, end byte, number of documents that contain this word.

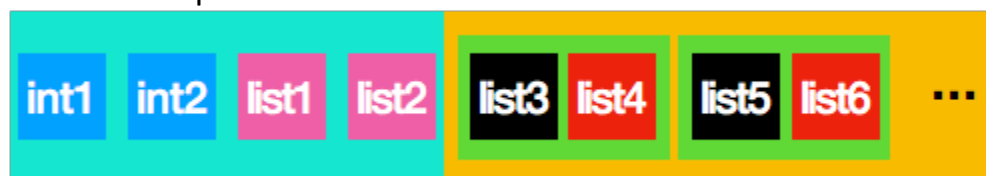
Example: happily,558908207,558913791,1385

Page url table: doc id url doc length.

Example: 77 http://1057thehawk.com/this-is-great-revenge-on-a-double-parked-jerk/ 4976

Inverted index list:

Each word owns a tuple like below:



Pic 3.1 Inverted index list structure

The yellow block represents data source part. Green represents each chunk. Within a chunk, list3, for example, is the list of some doc id and list4 is the list of frequency of this word appears in that doc.

Example: list3 is {1,2,3} and list4 is {5,7,2}. We can tell that the word appears 5 times in doc 1.

The list1 is the list of those last doc ids.

Example: in yellow block, we have list3: {1,2,3} list4: {5,7,2} list5: {4,5,6} list6: {9,1,3}. We have list1 = {3,6}

The list2 is the list of size number of each chunk (green)

Int1 is the byte length of list1 and int2 is the byte length of list2.

We call the light blue part meta data.

To seek an inverted index list for a word, we locate the word in lexicon, get its start byte. With the start byte, we can locate the start position of its inverted index list. First, we read 4 bytes, get int1(which is the length of list1), then read 4 more bytes, get int2(which is the length of list2); read int1 bytes, get list1, read int2 bytes, get list2. And now we get the meta data of the inverted index list, then we can decompress only the part we want based on the meta data.

3.1.2 General workflow

App is the entrance class, **invertedIndexOrchestrator.class** contains the general workflow.

WetReader will read all *.wet files in “input” directory in a loop and each time when it finishes parsing a page, it will take data parsed and write (append) temp postings to a temp Postings file. WetReader will keep updating a “page_url” table at the same time.

The content of each page will be stored in mongoDB, whose key is their doc id, for further query usage.

This part is using multi threads, each wet file will generate a temporary posting file.

UnixSort will sort each posting file, merge those sorted posting files and sort the merged posting file and store sorted postings to “sortedPostings.txt”.

IndexGenerator will build the inverted index list using data in sortedPostings.txt and generate the lexicon while building the list.

3.1.3 WetReader

WetReader can recognize each page when going through the text lines. For each page within one *.wet file, its headlines will be stored in a list of strings. And the same for the content lines of each page.

After getting content and header data for a page. It will:

- a. Get unique items and their frequency within a page, store the information in a map and add it to a list of map called “wordCountMapList” (it is actually store the information of the intermediate postings of this wet file). There is a filter here to keep only ASCII terms.
- b. Get the page’s url and update the page_url table.

After it finishes processing data for a wet file, intermediate postings will be generated and append to the existing tempPostings.txt (it will create one if not exist).

3.1.4 UnixSort

It is implemented in a shell. It sorts the intermediate postings by term, for those with the same term, sort them by docId. The code is quite simple:

```
#!/bin/bash

cd output/

cat Postings_*.txt > sortedPostings.txt
sort -S 2G -t$'\t' -k 1,1 -k 2n,2 -o sortedPostings.txt
< sortedPostings.txt
rm *_*.txt
```

Pic 3.2 Unix sort shell commands

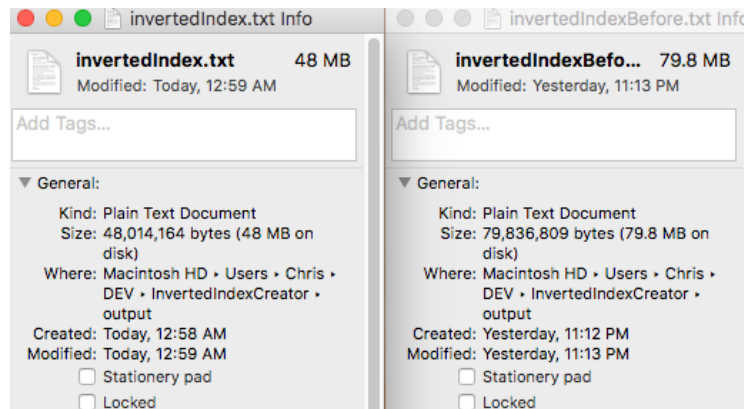
This part is a bottleneck of the whole process. For a 5GB intermediate postings, it takes about 10 minutes to finish sorting. For 10 GB postings, it takes more than 40 minutes (which is wired but I haven't figured out why).

3.1.5 IndexGenerator

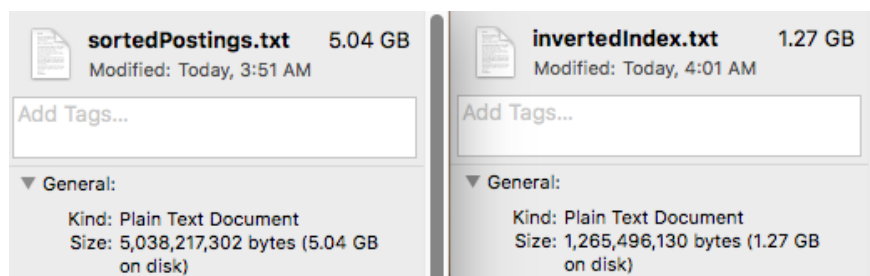
IndexGenerator will open a `FileInputStream` to read sorted postings and build the inverted index lists. The detailed structure is already discussed in section 3.1.1. All data are compressed and written in binary format. Start byte and end byte for each term's inverted index list are stored in a `lexionMap`, whose key is term and value is (startByte, endByte, numOfDocContainsThisTerm).

3.1.6 VarByteCompressor

Inverted indexes are all compressed to binary format before written out. In this homework, we use varByte algorithm to compress. I found a Java implementation of varByte compression from github: <https://gist.github.com/zhaoyao/1239611> Based on that I create a class `VarByteCompressor` which can encode Integer List to byte array and decode byte array to Integer list. It can compress inverted index list from 5.04GB to 1.27GB. Below is a test shortcut for the compressor.



Pic 3.3 Compressor test



Pic 3.4 Postings and compressed inverted index list

3.2 Query Processing

3.2.1 General workflow

Pre-loading data (lexicon and page url table) from files generated -> extract keywords from query and identify if the query is conjunctive or disjunctive -> **QueryProcessor** handles the keywords searching and return results -> generate snippet -> return results in XML format.

QueryOrchestrator is the main module that carries the workflow.

3.2.2 QueryProcessor

Implemented Document-At-A-Time query processing.

If the incoming query is a conjunctive query, with the help of nextGEQ() function, we can find out documents that contains all keywords in a smart way, calculate their BM25 values and put into a priority queue.

*nextGEQ(k): find the next posting in an inverted index list with docID $\geq k$ and return its docID. Return value $> \text{MAXDID}$ if none exists.

If the incoming query is a disjunctive query, we scan the inverted index list of each keyword, calculate BM25 value for each doc and put it into a hash map to avoid duplicate. After the scan finishes, we use a priority queue to get the top 10 documents from the map.

3.2.3 Generate Snippet

With the QueryProcessor, we get a list of document IDs. During the process of generating inverted index list, we store the content of each page in mongoDB, now we can retrieve the content data of those documents we have.

The content of a page is a string, we use a fixed-size sliding window to traverse the string. The size for now is 200 chars. If the content is shorter than that, we just return the content itself as the snippet. The window moves 200 chars ahead for every loop and each time we can get a substring of the content, we call them snippet candidates. We run an auction among those candidates. The basic idea is a candidate with a better diversity of keywords would win the auction. For example, "...cat...dog...bird..." has a higher score than "...cat...cat...cat...cat...dog".

3.2.4 Backend Server

The server parts are in the "BackendSever" package. Basically, it listens from "localhost:8888" to get incoming query, parses query to query processor and pack the results into XML format then send back.

4 Results

4.1 Inverted index generator

For 4 million pages, it takes about 2 hours and 30 minutes. The speed is about 450 pages/second.

The inverted index list is 4.96 GB, lexicon is 645.6MB and page url table is 343.8MB.

4.2 Query processor

Example result:

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<?xml version="1.0"?>
<root>
  <docId>
    1242060
    <bmValue>7.39661650835809</bmValue>
    <freq>to : 3; be : 9; or : 9; not : 6; to : 2; be : 2;</freq>
    <url>
      https://secure.actblue.com/contribute/page/commonwealthclub?recurring=12
    </url>
    <snippet>
      ...59fbede41aff9e1271fa9bf7, 1242060, I, ActBlue - Contribute Now to Commonwealth Club, Hello, Your account, Secured by ActBlue, Making a contribution on ActBlue requires JavaScript to be enabled.. Plea...
      ..date or candidate's committee., Contributions or gifts to ActBlue are not deductible as charitable contributions for Federal income tax purposes., , j... ..being provided by any other person or entity.,
      I am a U.S. citizen or lawfully admitted permanent resident (i.e., green card holder),, Paid for by ActBlue (actblue.com) and not authorized by any candi...
    </snippet>
  </docId>
  <docId>
    1491002
    <bmValue>7.16966132590805</bmValue>
    <freq>to : 6; be : 21; or : 21; not : 4; to : 8; be : 8;</freq>
    <url>
      http://www.carsurvey.org/addcomment.php?pageid=136166&type=reviews&parent=335978
    </url>
    <snippet>
      ...ome > Add a Comment, Warnings:, Insulting or offensive comments will be removed, Adverts are not allowed, Comments MUST be written in English, To keep things manageable, frequent visitors must not post...
      ... more than 3 comments per day, Defamation and Libel:, Very important - it is not acceptable to use this website to post defamatory or libellous comments about individuals or organisations. This means ...
      ...e will be passed on to the appropriate parties in cases of defamation and libel, I agree with the rules above: Continue to the next page, Home > Add a Comment, Copyright 1997 - 2017 CSO Media Limited...
    </snippet>
  </docId>
  <docId>
    1412193
    <bmValue>6.783655760252166</bmValue>
    <freq>
      to : 22; be : 64; or : 64; not : 59; to : 37; be : 37;
    </freq>
    <url>http://highpress.tripod.com/highlanddress.html</url>
    <snippet>
      ....ith plated metal top or leather to be worn., Waistbelt: Leather with plated metal buckle may be worn, but not worn with a waistcoat., Head Dress, Balmoral must be worn with appropriate crest and/or ba...
      ...ack of neck is optional. The full length sleeves may have a single row of not more than five silver or gold buttons at the vent. If desired, ruffles (not more than 1 inch in depth) may be worn at the ...
      ...een. White dress has to be worn with waistcoat type bodice., Note: The dress is usually made of cotton or similar material. Luxe fabrics should not be used nor should sequins or similar ornamentation...
    </snippet>
  </docId>
  <docId>
    2934907
    <bmValue>6.709539690733926</bmValue>
    <freq>to : 5; be : 9; or : 9; not : 4; to : 6; be : 6;</freq>
    <url>
      http://www.thewarpath.net/newreply.php?do=newreply&p=439494
    </url>
    <snippet>
      ...in Message, You are not logged in or you do not have permission to access this page. This could be due to one of several reasons:, You are not logged in. Fill in the form at the bottom of this page an...
      ... to post, the administrator may have disabled your account, or it may be awaiting activation., Log in, User Name:, Password:, Forgotten Your Password? Remember Me?, The administrator may have required...
      ... you to register before you can view this page., All times are GMT -4. The time now is 07:43 AM., Contact Us - Warpath: Redskins Fan Site - Archive - Top, Powered by vBulletin® Version 3.8.7, Copyright...
    </snippet>
  </docId>
  <docId>
    2856123
    <bmValue>6.680205785535408</bmValue>
    <freq>to : 3; be : 2; or : 2; not : 2; to : 2; be : 2;</freq>
```

Pic 4.1 Example result

For normal short query, it gives results almost immediately.

Limitation:

For long query, however, for example “to be or not to be”, is not very fast. It takes about a minute to return the result.

For now, the program can only handle one query at a time, which means you need to wait until the previous query completes to search the next query.

5. Appendix

Briefly introduce some important modules

5.1 Package InvertedIndexGenerator

InvertedIndexOrchestrator: main workflow

WetReader: read Common crawl data, parse and store them in temporary files

InvertedIndexGenerator: read temporary files and generate inverted index lists

OutputUtils: write functions for temporary files, lexicon and page url table

5.2 Package QueryExecutor

QueryProcessOrchestrator: main workflow

QueryProcessor: get top 10 documents for the query

DAATUtils: get meta data function; decompress data from inverted index list function

BMCalcultor: calculate BM25 value for a given document

5.3 Compressor

Compress and decompress data using Var-byte

5.4 DocIdCounter

Atomic counter, used for keeping track of docId during multi-thread parsing in inverted index lists generating process.

5.5 Front end

Front end is implemented based on a framework from

<https://github.com/lavyun/vue-demo-search>.