Indexing and Query Processing

Web Search Engines Assignment

Wei Xu ID:0525157

Hao Wu ID: 0532479

*Abstract*—This paper we will explain how to create an inverted index structure from a large data set of web pages. Also how to efficiently indexing terms into the disk. We will also cover how to process a query from user and return them the most related pages.

Keywords—indexing; web search engine; compression; indexing; Query processing

# Introd

In this paper, I will cover how to write a program to create an inverted index structure from a set of downloaded pages. We will use efficient I/O method to do this. After we created the inverted list and the lexicon file, we can analysis a given query and score the pages that contains all the terms of the query and output the top 10 results. We will also cover how to compress the data, how to use Document-At-A-Time Algorithm to find the common document ID for terms, and also how to score a page score. As a result, when the user input a query, we will be able to find the top 10 score pages

# Create the inverted List

This program will first parse the downloaded web pages(many of them). Extract all the words that are in the pages, including all the words in all the tags, including <p> tag, title tag, header tag, etc.

After that, we create some text files (let’s call them raw files) to store the words and also the document id associate with it. We write one raw file for one data file. For example, In the data set folder we have 100\_data, 100\_index, what we create is 100\_txt. We can see that in the raw data file, they contains many duplicate tuples. For example, if cat appears in document No.1 many times, This tuples will appears multiple times in the raw text file.

After we had the “raw files”, we use unix sort to sort the raw data files into text file name sort\_XX(0-41).txt

Which will sort the raw data (remove the duplicate tuples so much smaller). Now within the sort files, we get all the tuples(word and document id).

After that, we can merge (“unix merge”) all the sorted files into a file named merge.rpt, The will contains all the tuples(word document id) in sorted order of all the files.

Now we can build the inverted lists of words. Since the whole inverted lists for all the data are very long. We separated the whole inverted list into different file. 10000 lines per file.

We also build a docID-to-URL table which will help us to find the url given the particular document id.

For the homework 4, we changed the way to create the lexicon and inverted list file.

# More Detail about homework2

(a)HTML parsing, we write c program (parser.h, parser.c).

The usage of the the parser.

function to parse urls and pages into words

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

function to parse urls and pages into words

return value:

a) >= 0, if success;

b) -1, if buffer (for words) too small

arguments:

1) page to be parsed;

2) buffer to store parsed out words and their contexts;

format: "<WORD> <CONTEXT>\n"

<CONTEXT>: 'B', bold; 'H', head (h1-h6); 'I', italic;

'P', plain; 'T', title; 'U', URL.

4) length of buffer

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

(b) For indexing:

format of the index file:

[www.](http://0010024.e-xpert.co.nz/SITE_Default/SITE_ema/search.asp)cnn,com1 0 2161 203.110.28.35 80 ok

first entry is the URL, the 4th entry is the length of the page in bytes, and the last entry is the return code (which could be ok or 404 or 403, etc). To get the starting offset in bytes for the 5th page, we need to add up the lengths of pages 1 to 4.

We didn’t uncompressed the data and the index to read them, we used zlib library to read the data accounting to the length that the index provided. Once we get a particular page, we can immediately parse the html and get all the useful words.

Once we get all the words, we can write it to the “raw files” like 0.txt, 99.txt, which will be sorted in the next step.

(c) We used unix sort to sort all the “raw files” :

we sort all the txt file in the dir and out put it as a sort\_XX.txt, we only keep the unique tuples.

sort $dir/\*txt > sort\_${i}.txt

After this step, we get all the sorted files which will be merged in the next step.

(d)The merge step:

We also used the unix sort(but this time only do the merge, not to sort any data) into the file we named it merge.rpt.

using this command line:

sort -m sort\_\*.txt > merge.rpt

in the next step, we will used this file to create the inverted list. More details will be explain in the Homework 4 section.

# Changes in Homework3

In hw3, our docID\_url file is very simple, it just contains information of the document id and its corresponding url. But we change a little when parsering the pages. We now also store the page sizes, the total count of words in the pages.

The format looks like this:

9 025-credit-card.pacificbusinesstrust.co.nz/ 10096 617

For example, the document id is 9 and the url is 025-credit-card.pacificbusinesstrust.co.nz/, the total size of the page is 10096 and the total word count in this page is 617.

# what Homework3 program can do

The user now input a query, we will find all the terms in the query and find the documents that contains all of these terms. We use the BM25 to score all of the documents, and return the urls of the document of the top 10 results.

# How to run Homework3

In the previous homework we get the merge.rpt file, which will contains word id followed by the document id, if the postings are duplicates, we will remain the duplicate values.

We will go the test file under the ps2 folder first.

When we get the merge.rpt file, we can type:

$python merge.py

to create the lexicon file called lexicon.rpt and the inverted list file called inverted.list and the also the frequencies inverted list file which is called frequency.rpt.

Once we get all of the above files, we can begin to do our queries.

Then we can run query.py.

After that you can input some query(words are separated by space)

We will then get the top 10 results of the url pages and also their BM25 scores.

Note: we can also download the pypy which can make the program faster. (first compile the file and run it)

# More details in creting inverted list file

After we ran the hw2 commands, now we should get the merge.rpt, which is a very large file contains all the postings include duplicates.

Now we had the merge.rpt, which is a file contain all the word and its document ID. There are some cases that we had duplicate values. We wrote a function in the merge.py, in the main function, basically what we did is to create the lexicon and the inverted lists. This function will create a lexicon.rpt file and inverted list file which is inveted.list and also frequency of the inverted list file which is called frequency.rpt.

Let me explain the contents of the lexicon file, for each line, it contains the information for each word. It will contains information such as the word\_ID(the word, the starting position of the inverted list in the file, the ending position of the inverted list. Also, we also write a words\_map into a file called words.map. This file will contain information about a word and its position in the lexicon file.

Example:

abandons 602192 604688

In this example, we know that the word ID is “abandons”, the invert list of “abandons” starting position is 602192, and the ending position of that word in the inverted list is 604688. Using this information we can read all the inverted list for word “abandons” which contains all the document id has that word.

Let me explain a little bit about the inverted list file. In the inverted list file, it will contains all kinds of document id separate by a space. We can get the whole inverted list of a word according to the lexicon file we just created.

# Compression

After The idea: We will use “take the difference” this idea to compress the document ID in the inverted list. Since the document ID in the inverted list is in order. I will record starting document ID, and the following document ID can be recorded using the difference by the previous one. This method can reduce lots of space.

Here we will use the vbyte method to do the compression. Here is the idea, if we don’t do the compression, no matter the number is big or small, we will use 4 bytes to store them, but if we use the vbyte, we can store the smaller number using fewer bytes.

The following is how we do it, for each byte, we will use the highest bit to indicate whether there has more bytes coming, if the highest bit is 1, we know that it is not done yet, the number has more bytes following. If the highest bit is 0, we know that this is the last byte for the number. So for example, if the number is less than 128, we can use only 1 bytes, since we can set the highest bit to be 0, and use the 7 bits to represent the number, which is 2^7= 128. If the number is less than 128\*128 = 16384, we can use two bytes to represent the number(we can set the first highest bit to be 1 and the other to be 0). Others will follow the same idea.

But there are some bugs in the program in this part. We later used another method to do the compression. We compress the docID only use three bytes for each docID. So later it is easy for us to read the inverted list file from the inverted list file.

# decompression

Our fisrt idea is to decompress the data by chunks. In the lexicon file, for each word, we stored the starting position of the inverted list and also the ending position of the inverted list. We can use these information to decompress the inverted list, we will use the information from the lexicon file and read the chunk of data out of the inverted list file and begin to decompress the data, so we will get all the document id of contains a word in this way.

But later we changed our decision to write this method. We know exactly each docID is 3 bytes, so when we want to decompose the inverted list for a word, we can look out the word in the lexicon file, find the starting position and ending position from the inverted list file. Also we know the document ID is exactly 3 bytes. So we can decompress the whole inverted list very quickly.

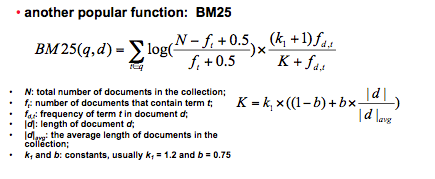
# Query Execution

In the query execution part, we will use DAAT(Document-At-A-Time). Basically, the idea is that given a query, we will fine the inverted list for all the terms. We will also find the document id that includes all the terms and using BM25 to score them. After that, we will find the top 10 or 20 results based on the scores of the document id, and response to the corresponding urls to the users.

We will use 3 import functions to implement DAAT.

OpenList(t): this function is to open the inverted list for the term t for reading. We actually create a words.map dictionary to store the location of a given word in the lexicon file as I mentioned before. So given a term, we can easily find the line that contains the useful information in the lexicon file. Once we find the line in the lexicon file, we will find inverted list for that word, because in the lexicon file, we will store the starting position for that word and also the ending position for that word. Once we get this information, we can get the whole inverted list In this way, we can get the inverted list for that term. We did not write this function specifically, but in our function, we use the words\_map to get the line in the lexicon file, and we then use the lexicon file to get the whole inveted list.

NextGEQ(word): In this method we will take all the terms’ inverted list and find the document id which will contains all the terms. Then we score these document id accounting to BM25 method. We will then use Document –At-A-time Query Processing algorithm. The algorithm to find the document id contains all the terms in the query. The algorithm goes like this, we first find the shortest inverted list among the terms, we will get the next position on the shortest list, say we get k. we will use a for loop to find other terms with the same docID in their inverted list. If there exists, we can compute the BM25 score for that document ID using the following method:



Let me explain a little bit about the actual code. In the query.py:

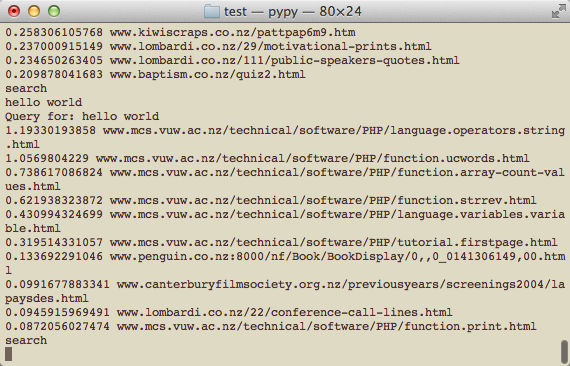
We first translate all the query into lower cases, and also ignore the special characters. Then for each word, we check that if it is in the dictionary, if we cannot find it, we will tell the user to try other query. If all the terms are in the dictionary, we can used the algorithm above to find the common document ids. For each document id we will compute the BM25 scores.

getFreq(lp): this method is to get the frequency of a particular posting. Since we create the frequency.rpt file to store the frequencies of an corresponding inverted list(the position matters). We store the starting position information in the lexicon file, the starting position for the frequency list for a word is also the same. The same as the ending position. So we can used the information in the lexicon file to get the whole chunk of frequencies corresponding of the inverted list. So we can easily get the frequency of a particular posting.The idea: We will use “take the difference” this idea to compress the document ID in the inverted list. Since the document ID in the inverted list is in order. I will record starting document ID, and the following document ID can be recorded using the difference by the previous one. This method can reduce lots of space.

# results

I ran the queries “dog cat” and also “hello world” both of the queries takes less than 1 second to output the results.





# limitation

In the lexicon file, we only evaluates the words that are in the dictionary, we filter lots of meaningless words, such as “zzzzzzzzzzzzz”. So for example, the number will be filtered because it is not a word in the dictionary. But sometime it is not that good.

The compression method we use still have some bugs. So in the program, we use a fixed size 3 bytes to store an integer.

ran the queries “dog cat” and also “hello world” both of the queries takes less than 1 second to output the results.

##### References

Index Compression: http://www.ir.uwaterloo.ca/book/addenda-06-index-compression.html