3.1 Index Construction

- Term Tokenizing:

R.E pattern “<DOC>\s+(.*?)\s+</DOC>was used to find the grouping of contents between these tags. And then iteratively going through each documents, content was extracted as per described below.

The parsing of the content in HEADLINE tags is done with the regular expressions that would ignore the punctuation and markup tags and read the content from <P> tag. Line 77 to line 83 in index.py file.

The parsing of the content in TEXT tags is done with the regular expressions that would ignore the punctuation and markup tags and similarly read all the content from <P> tag. Line 85 to line 91 in index.py file describes in detail. The R.E. used in the code - "[^\w\s]", will remove any non-alphanumeric characters including punctuation and hyphens.

- Term occurrence information

The term occurrence information is gathered using counter for counting occurrence of token in each document & a global hash table/dictionary to store the final Inverted list content. Accessing of counters is similar to that of dictionary. We pass the list of all token of document to a counter variable & it gives the term occurrence within that document for each term.

- Construction of lexicon and invlists file

The “InvertedList” dictionary in the code contains the unique term as key and the list of values denoting the document number it is present in and the occurrence of that term in that document. Using this hash table, we have created both lexicon & invlists files. We haven’t maintained a separate dictionary for lexicon as inverted list contained all the entries & we know the size of list for each term, we can calculate in how many documents that particular term is present. Using these information and while maintaining the offset for each term to identify where the data is put in the invlists file, we created both files.

- The format of files

The lexicon file contains a 3 variable tuple separated by “:”; the term, term occurrence and the offset value to identify from where to read for that term in inverted list.

The invlists file contains binary integer data.

The map table contains a 2 variable tuple separated by “:”; an entry of unique Integer identifier mapped to each actual document identifier.

3.2 Stoplist

If user provides –p flag & gives the file path for stoplist file, we are storing the stop words into a dictionary & while processing each token, check them against this dictionary, if they are present, then we will omit those tokens.

Here, the list of stop words are not that high, so we can just simply store them into a hash table & compare them with each term for efficient comparison.

The data structure in which these words are stored is a dictionary based counter, which uses below as the hashing function, (a slightly modified version)

https://stackoverflow.com/questions/8997894/what-hash-algorithm-does-pythons-dictionary-mapping-use

A hash function should be easy to compute & should give uniform distribution, the hash function provided in python do this easily for string values, so it is an appropriate hash function for our task.

3.3 Index Search

When user runs search.py program & pass the lexicon, invlists and map files' path from command line argument, we fetch the lexicon & map file's content into the memory using a dictionary to store those values. Here, these file are store in hash table because the size of those will be large and hash table will me more efficient compared to other things for faster accessing of those value.

Lexicons are store in the dictionary with lexicon token as key & a list as value; which holds two elements. First is the number of documents in which that token occurs in & second is the offset for that token from which we will read in inverted list file instead from traversing it from start.

When user enters query terms in console, for each term, we first convert it into a lower case & then match that term with the entry in lexicon dictionary, which, if term is present – return a list containing the number of documents it present I & an offset from where to start reading inverted list.

Since, Inverted index is large in size, it is not loaded into the memory; instead it will be directly accessed to gather occurrence statistics. Since we have an offset value, we will not traverse through whole inverted index, we will just seek to the offset of that term and based on how many

documents it is present in, we will run a for loop to access all values of that term & display it to user.

Here, the value obtained from inverted list file has the sequence number for document id 7 not the actual doc id. So while processing the inverted index file values, we will also look up for the document id corresponding to the integer identifier retrieved from inverted index using a dictionary which has stored the integer id to actual doc. Id mapping from the map file provided by user.

3.4 Index Size

Size of Inverted Index: 239 MB

Size of Lexicon file: 6.5 MB

Size of Mapping file: 2.66 MB

Grand Total: ~248 MB

It is way lesser than the original collection. Approximately half the size of original file in our case.

As the Inverted Index store the word → documents mapping instead of document → words mapping, the content of the documents can be significantly reduced by storing the occurrence statistics of each word, instead of storing whole word each time it appears in a document. As it converts all these terms into frequency, the content that need to be stored to hold that information is much less than original content and so index is smaller than original.

3.5 Limitation

In our program, we haven't done anything for hyphenated words, stemming & plurals. Hyphen is removed directly and singular & plural lexicons have separate entries in lexicon file for a word. We can use some stemming methods to remove those difference.

Currently is writes integer in a 4 bytes value, instead we can use some compression technique to store inverted index values which can result in even smaller index & faster access to those values.

It does not store any information about relative position of tokens in document, which can identify context specific meanings or similarity of words occurring together to find relevant documents related to query term.

Currently, the program only considers frequency of query term in a document, actual system may contain several other values like length of the document (large document tend to have more probability of occurring a word), weight of each term in a document etc. to correctly identify the relevant pages regardless of length of document.