Report
ISYS 1078/1079 - Information Retrieval
Assignment 1: Indexing

Student number and name:
Assignment 1 Group 4
S3384039 Shuijia Zhuo
S3542943 Anesu Chiodze

3.1 Index Construction

1. The method “mapDoc()” in Index.java(main class) is used for parsing and Document.java set up for saving the document stuffs.

When the program load the latime file, start on reading the file every-line, when the program find a line with “<DOCNO>....</DOCNO>”, it will add the content between the <DOCNO> tags to the arraylist called “docList” and also give this docNum a unique docID (an ordinal number as a document identifier).

Since we need the contents of HEADLINE and TEXT, there are three Boolean sets called “openHeadlineTag”, “openPTag” and “openTextTag” for collecting the contents we need. After the DOCNO is collected as a new document, the program will keep loading the following lines after the <DOCNO>.

When two Boolean sets are true, the program will collect the words, which will work like

Find <HEADLINE>(openHeadlineTag = true) ->
Find <P>(openPTag = true) ->
(double true)load the contents line by line and tokenize each word (by line.split(“ ”))->
add word to Map (add word and frequency) ->
Find </P> (openPTag =false) ->
Find </HEADLINE>(openHeadlineTag = false) ->
Continue to find <HeadLine>&<Text>
(same as the Text Boolean)

During the tokenize and addword process, a method called “cleanString” will apply to the word String, in order to clean all punctuation and markup tags in front/back of

each word, and also make them to lowerCase. For acronyms and hyphenated words, we treat them as whole word, so after “cleanString” method, we keep these words in our word list without breaking them into parts, for example, “we’ve” as a whole word and “300-234-321” is some kinds of phone number that we shouldn’t break.

2. In order to gather term occurrence statistics, we use the “addWord” method in Document.java, when the “dict” Map gets a word; if dict.get(word) == null (which mean its a new word), it will record this word and the int 1 (which for frequency), otherwise it add up of frequency.

3. We create a list which contains all the collected words and sort them (not fully necessary, for redundancy reasons). Then, we check if the word exists in the lexicon Map “Map<String, LexiconNode> lexicon”. In order to accomplish the quick fast search, we decided to use a map data structure (which allows for us to quickly find what we need), specifically a tree map (which allows us to make sure it is in sorted order). If it does not exist in the lexicon list, we create a lexicon node (i.e a lexicon) which we then create an inverted list node for that lexicon (which contains the document ID and the frequency). Initially we were going to calculate the frequency in this method, but we realised that we can save time and resources to do it earlier during the mapping process (hence you will see a method such as “addToCounter” within the inverted list for the lexicons). To fit the newer changes, we implemented a way to instantly set the “counter” which is the frequency of the term within the document collection. The Document ID and the pre-processed frequency are stored in docList, so we can easily retrieve them by calling “doc.getDocID()” and “doc.getMap().get(key)”.

After all, we would able to write out the file of lexicon and invlists easily by calling the items in lexiconList. For the “lexicon” file, we write the term (word) and the pointer, and for invlists we write the binary data of the invertedList size, documentID and frequency obtained from the lexicon list and the corresponding inverted list information for each lexicon..

4. We decided to store lexicons as plain text file, which contains word(String) + pointer(int.). For invlists, we store document frequency, documentID and within-document frequency in binary. And map, we store the documentID and documentNo as plain text file.

3.2 Stoplist

We use each word in the stop word list as a key in the hashmap, in order to remove the stop word from the hash map. We use hashmap as a data structure to keep track of the words, because it is easier for us to look up a word and we don't need to iterate the hashmap to remove a particular item. In our program, we add all the words into String[] by BufferReader, and then we compare the stop-words to the words in Hashmap after indexing. If there's a same word we remove it from the hash map.

3.3 Index Search

Hashtable is the data structure used to hold lexicon and map in memory. We load the lexicons and map separately, store them in Map<String, LexionNode> lexions and Map<Integer, String> map.

We check the entered command line is valid or not(i.e. The line should be /search lexicon invlists map word1 word2 etc.) at first. Once it is valid input, we put all the input words into a String[], and do a for loop, which looks like

```
for(every word in String[]) {
        If (term exists in lexicon) {
                Get additional information for the term in the Map “lexicons”, get the
        pointer.
                Use pointer to find the exactly line in invlist,
                        get binary data and store in ”invlistDataParts”.
                Print the current query term.
                Print document frequency (transfer binary data back to int).

                For (every location of word){
                        Get the documentID in “invlistDataParts”,
                                find documentNO in Map”map”.
                        Get the counter in “invlistDataParts”,
                                find within-document frequency.
                        Print documentNO + within-document frequency.
                }
        }
}
```

If there is no matching of query to word, the program will print nothing.

The search speed is quiet fast and it only takes around 30 seconds to find 3 query terms. The reason for the faster querying times is because we use pointer as a coordinate for jumping to the line of interest, and it helps us to reduce time and resources since we only need to load small amounts of info into memory.

3.4 Index Size

Without stoplist
Lexicon: 7,568 KB
Invlist :597,046 KB
Map: 2,597 KB
Total: 607,211 KB (Larger)

With stoplist
Lexicon: 7,562 KB
Invlist: 439,584 KB
Map: 2,597 KB
Total: 449,743 KB (Smaller)

Our indexing would in theory reduce in size because of the removed tag information and duplicates in words. However, we need to keep in mind that we also store additional information for every term such as the pointer to the inverted list file. And then in the inverted list file we store even more information for every term. This would technically make the total file size bigger. Although this assumes that the collection has a lot of unique words like our situation (i.e if it had the same word several times in the collection, we would have only one line in the invlist). In our case removing stop words reduced the file size to be smaller because there was a lot of them as expected in a normal collection. If we had compressed the invlist (the biggest file) we could have potentially significantly reduced the file size (but inherently increased processing time) .

3.5 Limitations

- A memory issue may occur when running a large latime file on a system with less memory. I downloaded latime (480MB) into my laptop and tried to index it, but an error “GC overhead limit exceeded” occur in half-way of indexing. The latime-100 is fine since it is a small file to index, but once the file becomes larger, my laptop won’t able to run though the program. The index is working in RMIT server so it will be a limitation for the small Ram PC. We can overcome this issue and make the program adaptable by taking into account the memory on the given

environment and then process the information in “chunks”, but this would take a significant modification to the current source code.

3.6 Contributions

The part 1.1&1.2 and report:
Shuijia Zhuo
The part 1.3&2 and report corrections:
Anesu Chiodze