CS276 PA1 Report

Rukmani Ravi Sundaram **Tayyab Tariq**

1.Description of the structure of the program: index.py :

The algorithm used for indexing is the Block Sort based Indexing. The data is divided into blocks of 10. Each separate file is considered a 'document'. Tokenization is simply done on whitespace characters.

Indexing - Key Steps: Each block is read into memory one at a time. So at any point in time, only 1 block is in memory. The documents are parsed and (termID, docID) tuples are collected. These are sorted according to termID, then concatenated into postings list for each termID, which are in turn sorted according to docID (needed for merging at query time). Each block gets written into memory as two files. One hold (termID, pointer to postings list, length of postings) tuples (- posting.dict). The other file contains all the postings list of all termIDs (- corpus.index) All files are written as binary files. After all 10 blocks are parsed this way, the next step is to merge. A binomial merge is done, i.e we merge 2 intermediate indexes at a time. Merging proceeds as follows. Read a line from each posting.dict file of each of the 2 blocks. Each termID is looked up in the block index's corresponding posting.dict along with its file position and length (in bytes), so we know where the postings list for that termID ends. Next depending on whether the termIDs are same or different, merge the corresponding postings lists or write the single postings list to the intermediate combined output postings file and also update an intermediate combined postings dictionary file. Repeat until we get a single index. The number of disk seeks/read/write is proportional to the sum of the number of terms in each block index, so O(T), where T is the total number of terms in the vocabulary.

Gamma Encoding - Each postings list was separately gamma encoded and right padded with 1's, for ease of access, as the postings_dict maintains the length of each postings list in bytes.

Query - Key Steps: At the time of query, query terms are sorted in ascending order of docfreq, which is maintained in the postings dict file. Postings list for each query terms are merged two at a time, and the resulting postings list is finally mapped back to its document name and displayed.

Size of Index(postings list):
Uncompressed : 55.3MB
Variable Byte Compression : 16.4MB (29%)
Gamma Encoding : 11.5 MB (20%)

Encoding	Query Time	Index load time
None	0.23s	6.07s
Variable Byte	0.098s	5.842s
Gamma	0.5s	5.2s

Indexing Time:
Uncompressed : 10 minutes,11 seconds
Variable Byte Compression : 7 minutes, 40 seconds
Gamma Encoding : 43 minutes, 58 seconds

2.

a . Larger sizes of a block will likely minimize indexing time, but not by a whole lot. Since the bulk of the indexing time is dependant on disk IO, small performance gain will be obtained if

we process(parse/sort) larger sized blocks in main memory. The size of the blocks has impact during the merge phase, especially if the binomial merge is done. The merge step alone is O(T*logK) in disk IO, where K is the number of blocks, and T is an upper bound on the number of terms in any one block. So the merge step is upper bounded by the time needed to read/write T terms, logK times. By reducing the number of blocks, or in other words, increasing the size of each block, we can improve the merging time. However block sizes are restricted by the amount of main memory available. So a general strategy to minimize indexing time will be to have the largest possible block sizes, i.e equal to the size of the main memory.

2.b. The merging of each block's postings list in its current form limits scalability to large datasets. The number of disk seeks/read/writes performed during each merge is proportional to 3 times number of terms in each block index. This is because we read in one posting at a time into memory, perform the merge step and write it back to disk. This step can be optimized. We could read in a subsection (say K postings lists) of each intermediate index , perform the merge in memory and when the merged postings lists exceeds main memory size, write it back to disk. This way the number of disk reads/writes is minimized to one every K terms (if we assume we read in K terms from each of the two block postings) .

Another part that can be optimized is the simultaneous populating of each term's postings list as we parse the documents, as opposed to collecting the (termID,docID) pairs, sorting them and then combining them into (termID, postings list). This saves us the extra combining step.

A third optimization of indexing time that can be done is to compress only the final postings list, instead of compressing/decompressing each intermediate postings files. This can be only be done if disk space is sufficient to hold all the uncompressed postings.

2.c To improve retrieval time,one optimization that has already been done is to merge the postings list of terms in increasing order of doc frequency, to minimize the number of steps taken to merge. We could also take advantage of multithreading, by retrieving the postings list for the next query term, while the merging for the previous two query terms is taking place.

Using skip lists can help reduce query retrieval time, at an additional cost to the disk space and indexing time, since we now have to populate each posting with an additional skip pointer. But the extra cost in building and maintaining skip pointers may not be too practical in dynamic IR systems.
A variable byte encoding seems to perform a lot better (with respect to retrieval time and indexing time) than a variable bit encoding since computers are more optimized to work with byes than manipulations at the bit level.