CS 276 Programming Assignment 1 Project Report

Jiaji Hu, Xuening Liu

April 16, 2014

1 Program Structure

1.1 Basic Structure

For creating the index, we use Blocked sort-based indexing (BSBI). Globally, we maintain three dictionaries, the term dictionary, document dictionary and the postings dictionary. The term dictionary and document dictionary are used to save space in the index as well as for convenience in operations. The postings dictionary is used to quickly find the correct postings at query time.

In this assignment, the data is given to us already segmented into blocks (different folders). Next, we read the terms from the documents(files) of each block, generate a $< termID, docID >$ pair. All such pairs that appear in the block are then sorted with termID as the primary key and docID as the secondary key. From that, a postings list can be created for each termID. There are then written out to disk as intermediate results. In this assignment, we used three different methods to encode and store our index, which will be discussed in detail in the next subsection.

In our implementation, we opted to include the length of the postings list (the number of docIDs) at the front of the posting list, and then write it out to disk. By doing so, we no longer need to record the length of the lists in the postings dictionary. (By not saving the frequency in the postings dictionary, but in the actual index, we can now use compression encoding on that number, potentially saving some space in total.)

Finally, we merge the intermediate blocks into the final index. In this assignment, we perform binary merge on the blocks. The effect of the merge is that postings lists with the same termID will be combined. The algorithm to merge postings lists is quite similar to the algorithm for the intersection of postings lists, which we discussed and implemented in class.

To handle queries, we parse out the query terms, find their termIDs using the term dictionary, find their postings lists using the posting dictionary, and intersect the postings lists. For efficient lookup, the terms are ordered by the length of their postings lists. After the intersection, we translate the docIDs to document names, and output them as the result of the query.

1.2 Encoding methods

(a) Encoding for Basic algorithm

For task 1, we simply write out the 4-byte integers. In this scheme, every integer takes up 4 bytes.

(b) Variable Byte Encoding

For task 2, there are two differences. Firstly, we use gap encoding for docIDs. This will make the numbers smaller, and therefore use less memory after compression. Secondly, we use Variable Byte encoding, where the first bit of each byte is a continuation bit to signal termination. Under this scheme, small numbers take up much less space. However, doing this introduces extra steps for encoding/decoding, so it is slower than the uncompressed version. Therefore, there is a trade-off of less speed for better space when using variable bit encoding. However, The savings in space could well be worth the decrease in speed.

(c) Gamma Encoding

Gamma encoding adapts the length of each integer on the bit level. It uses a unary code to represent a length, and then a binary code of the indicated length to represent the offset value. Using gamma encoding, we can gain further performance in terms of compression. For example, under this scheme, only one bit is required to encode the integer '1'. Formally, gamma encoding achieves compression within a factor of optimal.

However, the encoding/decoding steps for gamma encoding can more expensive, because there needs to be many bit-level operations, such as shifting and masking, in the encode-decode process. This leads to even more sacrifices in speed, and this is another trade-off situation.

2 Statistics

The following table shows statistics of different algorithms on the dev dataset.

Algorithm	index time(s)	index size (MB)	index size - including dicts (MB)	average retrieval time (s)
Basic	67	58	72	1
VB	149	18	33	1
Gamma	178	13	29	1

3 Questions

a) The indexing process mainly consists of two steps: First, we process the input data in blocks. Second, we merge the processed intermediate blocks into a single index.

With large blocks, the first step (in particular the sorts) takes longer, but the second step can be faster (we potentially make fewer merge passes). With smaller blocks, the first step takes less time, but more merge passes may be necessary, slowing down indexing time. So there is a trade-off of time between step 1 and step 2 for different block sizes.

We note that the first part is asymptotically more expensive, but the second part can involve a large amount of extra disk I/Os for multiple passes with binary merge, which is very expensive in terms of time. Therefore, generally, it is a good idea to make the block size as large as the computer memory allows (leaving room for other things such as term dictionaries, etc.)

b) With the current indexing program, the term dictionary could grow so large that it cannot fit in memory. If we are forced to move the term dictionary to disk, the extra I/Os would greatly affect the indexing time. Note that there is a similar issue with the document and postings dictionaries, but since we do not need to read from them very often, they pose less of a problem.

The following optimizations can be done:

(i) Use multi-way merge for merging intermediate blocks. Note that this can be done efficiently with heaps/priority queues. (Improves indexing time.)

(ii) Keep a cache in memory for the most popular queries, and/or extra intersected postings lists for frequent joint queries in index. (Improves retrieval time.)

c) (i) Instead of BSBI, use single-pass in-memory indexing(SPIMI). This is used to solve the scalability issue with oversized global dictionaries mentioned in the previous question.

(ii) The next step would be to use distributed indexing. Using a Map-Reduce architecture should solve scalability problems.