CS276 Programming Assignment 1

Raj Bandyopadhyay, Rafael Guerrero

April 15, 2013

Q1: Program Structure and indexing steps

We implemented this assignment in Python, mostly following the skeleton code structure. The indexing required the following steps:

1. Create an inverted index file on disk for each block (input directory). The format of each entry in the index file is: "termID:docId1,docId2,...", with one entry per line.
2. Place the names of each sub-index file in a queue
3. While the queue has more than one element
   (a) Pop the first two elements off the queue
   (b) Merge the corresponding index files into a new index file on disk
   (c) Add the name of the new index file to the queue
4. In the case of task 1, all index files are uncompressed. For tasks 2 and 3, we compress the index for the very final output index file
5. Delete all intermediate files

For querying, the steps are as follows:

1. Read the dictionaries for terms, words and postings into memory
2. Use the dictionaries to identify the location of a postings list in the index file
3. Seek to the location and read the postings list into memory
4. Uncompress the postings list if necessary
5. Merge (intersect) the postings lists for all the terms in the query, starting with the smallest list

Statistics

We report the indexing time as the total time required to build the index. The query time is the total time required to execute the eight development queries (loading the index files into memory only once).

We find that Variable Byte encoding gives a 5-fold size reduction over the uncompressed index, while Gamma encoding gives a further 30% reduction in size over Variable Byte encoding. However, there is a tradeoff in increased query time.

Table 1: Statistics for indexing algorithms

	Size of index	Indexing time	Querying time	Peak Mem (Index)	Peak Mem (Query)
Uncompressed	83.6M	100s	3.2s	502.072K	145.100K
Variable Byte	16.4M	131s	3.5s	502.232K	144.764K
Gamma	11.4M	131s	3.9s	502.356K	168.928K

Q2: Questions

a) Tradeoff at different block sizes

- The size of the blocks loaded into memory will ultimately determine how many disk seeks are needed to process the data. The larger the blocks the less disk seeks will be necessary, and the indexing time will decrease; the smaller the blocks the more disk seeks will be necessary and the indexing time will increase.
- The general strategy when working with limited memory but still wanting to minimize the indexing time consists on minimizing the number of disks seeks during the sorting phase. We should choose the block size large enough to fit in the available memory to permit a fast in-memory sort, as well as to minimize the disk seeks as much as possible .

b) Limits to scalability and possible optimizations

Yes, we are limited by the following facts:

- We load the document, term and file position dictionaries into memory. For a much larger (web-scale) index, we may not have enough memory in a single machine to maintain all of these dictionaries.
- We also assume that a single postings list is small enough to fit in memory. This may not be the case for very large document corpora.
- For long postings lists, the merge operation might take a while.

Possible optimizations for scalability

- Compress the dictionaries using the techniques described in lectures (e.g. blocking)
- Use skip lists to improve querying by reducing time for intersecting postings lists. This will consume extra memory, however.

c) Improving indexing or retrieval performance

There are a few ways we can improve indexing and retrieval performance:

1. Buffering (for indexing): In our merge process for blocks, we read in one postings list at a time from disk to perform the merge. This requires one disk seek per posting list for each block. However, we could implement two buffers, one for each block, and read in N posting lists at a time for each disk read. The merge can be done in memory for the two buffers. Each buffer is refreshed once it's empty.
2. Caching (for querying): We can maintain a small cache with K entries, each containing the postings list of an already-queried term. When a new query is entered, we first check for the existence of any of its terms in the cache before heading to disk. The cache can be maintained using an LRU (least recently used) policy.