Prabodh Shetty
32219166
pshetty@umass.edu

Assignment 1
Indexer Report

2019/10/07

1 System design and design tradeoffs

1.1 System design

Our Indexer consists mainly of *InvertedIndex, InvertedList, DocumentPostings* classes with *InvertedIndex* handling high level functions like building index, retrieving documents, and fetching Dice's coefficient, and *DocumentPostings* handling low level functions like getting all positions in a document, calculating term frequency in a document, etc. *InvertedList* is the middle layer which mainly deals with inverted list of each term. It's main functions are encoding and decoding list of *DocumentPostings*, maintaining offset and number of bytes used by postings list in disk.

1.1.1 InvertedIndex

InvertedIndex is the primary class that outside modules will mainly deal with. It basically consists of a **hash table** with **term** as it's key and corresponding **InvertedList** as it's value. Along with hash table, it also contains auxiliary data structures that are used to maintain statistics related to terms. Furthermore, it has high level functions like *createIndex, write, loadLookupTable, getScores, getDicesCoefficient and rebuildIndex*. It's main function is to maintain variables and functions that uses various *InvertedLists* for computation. Thus, we have functions like *write* which mainly delegate actual writing to disks to *InvertedList*.

1.1.2 InvertedList

Middle layer of our system, *InvertedList's* main functionality is to handle all the functionality related to inverted list of each term. Thus, it contains lots of functions related to inverted list of each term like *flushToDisk, reconstructPostingsFromDisk, getDocumentWiseScore, addPosting, etc*. As an object, it maintains a **list** of **postings** which is used in score calculation and dice's coefficient. InvertedList also maintains *offset* and *num bytes* for each list, thus making it a **lookup table** as well. In fact, whenever *loadLookupTable* is called by *InvertedIndex*, a partial *InvertedList* is created with offset and num bytes being loaded (Loading of postings list is always deferred till querying).

1.1.3 DocumentPostings

This class maintains all the information related to position of a term in given document id. Thus, it contains **doc id** and **positions list**. Since, this class maintains positions of each term, **delta encoding/decoding,** and **intersection of positions list** related tasks are usually delegated to this layer.

1.2 Design tradeoffs

Because of our assumption that document ids are ordered, two things were implemented. First, for fetching correct document posting in InvertedList class, I used binary search. Had it not been for the ordering of doc ids, I might had to do a linear search which would have certainly hurt the performance. Second, for calculating adjacent terms count in Dice's coefficient, I used merging technique (from merge sort), which reduced it's time complexity from $O(n^2)$ to O(nlogn).
Another thing that I thought of but did not implement was maintaining a list of valid document ids in document retrieval. In doc-at-a-time retrieval, we iterate through all document ids which creates a lot pf problems for query terms having smaller postings list. We can minimize this by maintaining a list of union of doc ids of each term in query terms. But this will require some additional space, which will worsen if one of the query terms (like "the") has huge postings list. So, I did not implement this.
Finally, first byte of index stored in disk represents whether index is compressed or not. Rest of the index is stored in disk as an array of bytes with following format [docId1 numBytes1 positionList1 docId2 numBytes2 positionList2 .....].

1.3 Documentation

1.3.1 InvertedIndex

- *createIndex*: Uses the raw data from Crawler to create new inverted lists.

- *write*: To write data to disk. It first sets compression byte then writes InvertedList one-by-one for each term to disk and finally flushes lookup table and data statistics to a json file.
- *loadLookupTable*: Used to load lookup table from json file to memory.
- *rebuildIndex*: Used to load InvertedList of all the terms in vocabulary. This is not used currently because in real life situations, this might lead to memory being full.
- *getScores*: Retrieves top k documents corresponding to a query. Uses document-at-a-time algorithm for scoring.
- *getDicesCoefficient*: Returns Dice's coefficient for given input of strings.

1.3.2 InvertedList

- *addPosting*: Primarily used in index creation, used to add a posting to postings list.
- *flushToDisk*: Main function to write data to disk. If inverted list is to be compressed, it first delta encodes the positions list and then encodes this array of integers using v byte encoding. Offset and num bytes of InvertedList also gets updated here.
- *reconstructPostingsFromDisk*: Another main function which is primarily used in loading postings list from disk to memory. Using offset and num bytes, relevant bytes are fetched from disk. These bytes are then decoded to integers which are finally used to re-construct original postings list. This method should always be called before accessing any postings related data.
- *getPostingsListByDocID*: Fetches postings list corresponding to doc id using binary search. This methods assumes that reconstructPostingsFromDisk function has already been called before.
- *getDocumentWiseScore*: Returns score corresponding to term (as inferred from current InvertedList) and given doc id.

1.3.3 DocumentPostings

- *addPosition*: Used in index creation to add new position to list of positions.
- *deltaEncodePositions*: Used to delta encode positions list.
- *deltaDecodePositions*: Used to delta decode positions list.
- *getAdjacentCount*: A binary operator which is used to calculate count of two terms occuring one after another. Used in calculating Dice's coefficient.

2 Count as misleading features

Reasons for count as misleading feature are as follows:-

- There are some words that are repeated more frequently than others. For e.g stop words like "the", "a" usually occupy bulk of documents. Most often these words do not add any value to the document. This leads to documents being almost equally scored because it contains high occurrences of "the". This can be mitigated if we either remove stop words or use tf-idf based scoring(which reduces the impact of high count of stop words).
- Sometimes, just because a word is repeated more often doesn't mean that the document is a good fit. For e.g we can have two documents on having lower word count for "fish" compared to another. But, the thing is, the document with lower word count has "fish" in it's heading. We can (to some extent) infer that the document with lower "fish" count has more importance compared to the one with higher count because "fish" is present in title of the first document. Basically, we also need to consider the weightage(with respect to it's neighbors) of each word in the document.

3 Scene and play statistics

- Average scene length: 1201.8663101604277 words
- Shortest scene(id): antony and cleopatra:2.8
- Longest play(id): hamlet
- Shortest play(id): comedy of errors

4 Experimental results

- Timings for 7 term queries:
  - Compressed index (in milliseconds): 0.7324483700000001 milliseconds
  - Uncompressed index (in milliseconds): 0.5332099499999998 milliseconds
- Timings for 14 term queries:
  - Compressed index (in milliseconds): 0.8663815300000001 milliseconds
  - Uncompressed index (in milliseconds): 0.8293484100000001 milliseconds

As can be inferred from above results, it looks like compression hypothesis doesn't hold (atleast for me). There can be a lot of reasons for why this is happening, out of which only 2 comes to my mind. First, it could be that index is being stored in SSD. Unfortunately, this is not the case. Had this been the case, because of high speed of data transfer from SSD, timing for uncompressed version could have been close to compressed one. Another reason, which I think is most probably the case, could be that all the random query terms have really small postings list. Because of small lists, loading data into memory did not take much time but compressed lists suffered from computation penalty and thus, making their timings comparable.
Results of experiment might change depending on which memory system and which compression algorithm is being used. If we use fast memory systems(like SSDs), then timing for uncompressed one becomes comparable to that of compressed ones because memory system has become less of a bottleneck. Another reason could be type of compression algorithm being used. Is it parallelizable enough? Can branch predictors consistently predict correct branches under this algorithm? If this is the case, then we have further decreased the retrival time for compressed inverted lists.