COMPSCI546 Assignment-1 Report

Jaideep Rao

September 2020

1 System Design

The system is designed to contain 3 independent components - The index builder, the query generator and the query retriever. Each component reads/writes to and from the disk, thus allowing all the components to to be executed asynchronously.
The system delegates the responsibility of reading and writing to disks to separate individual classes that handle all the File I/O operation, as I thought the code would be more manageable and intuitive to understand, would allow separation of logic and re usability of code. As such, these classes interact with all the major components of the system.
The retrieval model uses document-at-a-time approach for query retrieval as it is more memory efficient and would be re-used in future projects.
At the time of V-byte encoding and decoding, as a result of using Byte buffers, I was facing an issue where the excess space within the byte arrays was being populated with 0s that were interfering with the decoding process. I was able to fix this by identifying the issue and manually removing any 0 bits from the array
Another design choice I had to make was to use a naive approach for document and position array intersection while calculating dice's coefficient even though it is less efficient. I mainly had to do this because of some strange and unexpected behaviour I was observing from the code when trying to use the more efficient approach. (No matching terms were being identified for any of the query terms despite having verified that code is able to produce non-zero dice coefficient values)
I chose to convert each posting list into an integer array and flatten it out completely before applying delta encoding to it as it allowed me to update the array in-place and in a single pass, thus saving memory and computation cost

2 API Documentation

class IndexBuilder

- **BuildIndex()**
  public void buildIndex (java.lang.String compression) throws java.io.IOException

  Initiates the process of building the inverted index and calls other helper methods along the way. Takes in a parameter to specify if index must be compressed or not before writing to disk

- **getProcessedInvertedIndex()**
  public java.util.Map <java.lang.String, java.lang.Integer[]> getProcessed-InvertedIndex (java.util.Map<java.lang.String, PostingList> invertedIndex)

  Helper method to convert Posting Lists into Integer arrays and put them in a map

- **writeMapsToDisk()**
  public void writeMapsToDisk (boolean compressionRequired)

  Helper method to write auxiliary hash maps to disk

- **printInvertedIndex()**
  public void printInvertedIndex (java.util.Map¡java.lang.String, PostingList¿ invertedIndex, java.lang.String filename)

  Helper method to write inverted index to file in text format. Used only to visually compare if the compressed-decompressed index read from disk matches with the original index

- **getSceneStatistics()**
  public void getSceneStatistics()

  Helper method to retrieve longest/shortest scene and average length of scene

- **getPlayStatistics()**
  public void getPlayStatistics()

  Helper method to retrieve longest/shortest play and average length of play

class WriteToDisk

- **writeUncompressedIndexToDisk()**
  public java.util.Map<java.lang.String, TermLookupEntry> writeUncompressedIndexToDisk (java.util.Map<java.lang.String, java.lang.Integer[]> invertedIndex, java.util.Map<java.lang.String, PostingList> originalIndex)

  Writes uncompressed index to disk and creates the corresponding lookup table for it and populates it with information about term and document frequency for each term

- **writeCompressedIndexToDisk()**
  public java.util.Map<java.lang.String, TermLookupEntry> writeUncompressedIndexToDisk (java.util.Map<java.lang.String, java.lang.Integer[]> invertedIndex, java.util.Map<java.lang.String, PostingList> originalIndex)

  Writes compressed index to disk and creates the corresponding lookup table for it and populates it with information about term and document frequency for each term

- **writeLookupIndexToDisk()**
  public void writeLookupIndexToDisk (java.util.Map¡java.lang.String, TermLookupEntry¿ lookupTable, boolean compressionRequired)

  Writes the appropriate lookup table to disk based on the compression requirement specified

class ReadFromDisk

- **readUncompressedIndexFromDisk()**
  public java.util.Map<java.lang.String, PostingList> readUncompressedIndexFromDisk (java.lang.String[] queryTerms, java.util.Map<java.lang.String, TermLookupEntry> lookupTable)

  Reads uncompressed index from disk based on the query terms and lookup table and returns an inverted index containing just those terms

- **readCompressedIndexFromDisk()**
  public java.util.Map<java.lang.String, PostingList> readCompressedIndexFromDisk (java.lang.String[] queryTerms, java.util.Map<java.lang.String, TermLookupEntry> lookupTable)

  Reads Compressed index from disk based on the query terms and lookup table and returns an inverted index containing just those terms

- **readLookupFromDisk()**
  public java.util.Map<java.lang.String, TermLookupEntry> readLookupFromDisk (boolean compressionRequired)

  reads the appropriate lookup table from disk based on the compression requirement specified and returns it

- **readDocLengthsFromDisk()**
  public java.util.Map<java.lang.Integer, java.lang.Integer> readDocLengthsFromDisk())

  reads docId -¿ doc length map from disk and returns it

- **readDocIdMappingFromDisk()**
  public java.util.Map<java.lang.Integer, java.lang.String> readDocIdMappingFromDisk(java.lang.String filename))

  reads docId -> sceneId/playId map from disk and returns it

- **fromByteArray()**
  private int fromByteArray (byte[] bytes)

  helper method to convert byte[] to integer

class QueryRetrieval

- **retrieveQuery()**
  public void retrieveQuery (java.lang.String queryTermsFilename, boolean compressionRequired)

  Takes in name of file containing queries, and which index they need to be run against. Perfoms document-at-a-time processing and generates a ranked list of documents for each query

class SelectQueryTerms

- **selectTerms()**
  public void selectTerms (java.lang.String queryTermCount)

  Takes in count of number of terms per query and generates query terms randomly and writes the file to disk

class Posting

- **getDocId()**
  public int getDocId()

  returns doc id of a posting

- **getPositions()**
  public java.util.ArrayList<java.lang.Integer> getPositions()

  returns array list of positions for a posting

- **getDocumentTermFrequency()**
  public int getDocumentTermFrequency()

  returns count of occurrences of a term in a posting

- **getFormattedPosting()**
  public java.util.ArrayList<java.lang.Integer> getFormattedPosting()

  returns flattned representation of posting as an integer array and adds size of position array

class PostingList

- **getPostingIndex()**
  public int getPostingIndex()

  returns posting index of a postingList

- **getPostingList()**
  public java.util.ArrayList<Posting> getPostingList()

  returns array list of postings for a postingList

- **hasNext()**
  public boolean hasNext()

  checks if there are more postings to be iterated over

- **getCurrentPosting()**
  public Posting getCurrentPosting()

  returns the posting that is currently under consideration

- **skipTo()**
  public void skipTo (int docId)

  skips the posting index to a posting with a specific doc Id

- **add()**
  public void add (int docId, int position)

  takes in a docId and a position, creates a new posting object from this and inserts it into the list of postings

- **addPosting()**
  public void addPosting (Posting posting)

  adds a new posting object to the list of postings

- **getIntegerFormattedPostingList()**
  public java.lang.Integer[] getIntegerFormattedPostingList()

  returns the integer array formatted representation of a postingList

- **getTermFrequency()**
  public int getTermFrequency()

  returns term frequency for a particular term

- **getDocumentFrequency()**
  public int getDocumentFrequency()

  returns document frequency for a particular term

3 Why might counts be misleading features for comparing different scenes? How could you fix this?

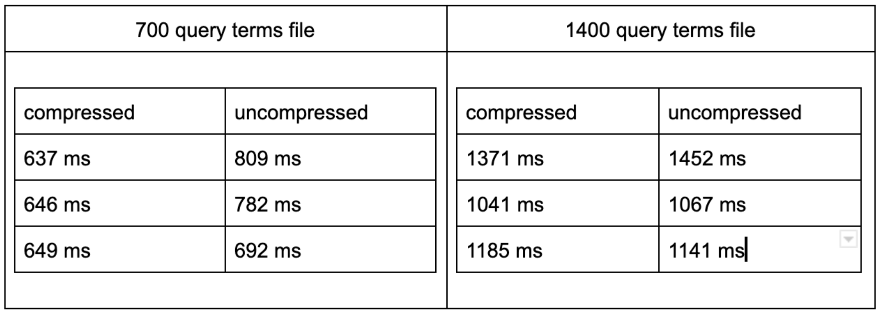
When dealing with queries containing multiple terms, using counts of occurrences of individual terms as the basis for comparison of 2 documents may be misleading because this method does not take into account whether or not these terms occur within the context of each other or are in any way related to each other in their usage within these documents. This can be fixed by looking at positional information of words rather than just looking at the occurrence counts. Query terms should be required to occur within a certain distance of each other (in words) to provide a reasonable degree of confidence that they occur within the same contextual setting and would be of relevance to the user

4 Corpus statistics

Longest scene is: **loves labors lost: 4.1**
Shortest scene is: **antony and cleopatra: 2.8**
Average length of a scene is: **1199 words**
Longest play is: **hamlet**
Shortest play is: **comedy of errors**
Average length of a play is: **24250 words**

5 Experimental results

In order to evaluate the performance of this indexer and to observe the effects of compressing the index, a timing experiment was conducted as follows: 2 files containing 700 and 1400 search terms respectively were generated. The indexer was used to generate 2 inverted indexes, one without using any compression, the other using delta encoding in combination with vbyte compression to store the index. Queries from both files were run against both, the compressed as well as uncompressed index. For each query, the required parts of the appropriate index were read from disk, and document-at-a-time was performed to retrieve a ranked list of relevant documents. The time taken to read the index and perform the query was recorded for both indexes and are available below:



700 query terms file		1400 query terms file	
compressed	uncompressed	compressed	uncompressed
637 ms	809 ms	1371 ms	1452 ms
646 ms	782 ms	1041 ms	1067 ms
649 ms	692 ms	1185 ms	1141 ms

Based on the observed results, I would say that I think the compression hypothesis holds. We can see that in 5/6 cases the compressed index was faster to load and retrieve queries from. This seems to suggest that the machine's processor is capable of reading and decoding the index faster than the storage system is capable of supplying it, which is leveraged by compression to provide performance optimization.

The results of the experiment may change if there is a change in the evironment that the experiment is conducted in. For example, running the experiment on a machine with a solid state drive as opposed to a regular hard drive. In this particular case, since the storage and access principles used in both these types of storage mechanisms are different from each other, and hence may produce different outputs. I would expect to see the advantages of using compression diminish when working with solid state drives since they are already optimized to provide high speed access to data