546: ASSIGNMENT 1
Programming Assignment - Indexer

Shravya Kadur
ID: 32167135

I. System Design

A. Description

The system is built using Python as the primary language. The three main classes in the system are:

1. **IndexerClass** – Designed to handle the indexing part of the system. It begins by parsing the collection, going through each document and extracting each word. The words are then stored in an inverted index, with each word having its own posting list. The mapping between docId and scene; docId and play is also maintained. Based on the user input, this class stores the created inverted index either in compressed or uncompressed form, with a lookup table on disk.

2. **CompressorClass** – This class is specifically responsible for all compression-related tasks. It takes the inverted index generated by the IndexerClass, uses delta encoding to transform the docIds and V-Byte compression to make the storage of the index on disk less expensive. It is also able to reverse-engineer this process to obtain the original inverted index from a compressed index file.

3. **QueryClass** – The query class handles all queries of the system. It generates a file which stores 7 random words in 100 iterations. This is then used to calculate the appropriate Dice pair (the word in the vocabulary with the highest Dice's Coefficient corresponding to each word) for each of the 7x100 words.

B. Design Tradeoffs

- The very first dilemma was the design of the Inverted Index. The choice was between a clean, easy-to-read list of tuples for each word with (docId, position) pairs and a list of integers with docId, length of the position array, and the elements of the position array themselves. The latter was chosen for ease of storage and compression.
- The term and document frequencies were initially stored with the inverted list, but were later moved to memory in order to enable ease of index compression and retrieval of the frequencies on-demand.

C. Resolved Issues

- With the first iteration of the code, the runtime of the 14-word query took 1 hour+ of computational time. This was fixed by cleaning up the code and reducing the number of disk accesses. The inverted index was retrieved only once in the beginning of the QueryClass, rather than each time the Dice Coefficient had to be computed. This brought the total runtime down to 6-10 minutes.
- Computation of the Dice Coefficient caused some confusion. The text mentioned the use of number of windows with concurrent occurrences of the pair divided by the number of windows each word occurred in individually, while there were class discussions on using the term frequency for computation. This was resolved by using term frequencies since a and b have no intervening terms, and term frequency exactly captures the value of n_a and n_b.

D. Index and Retrieval APIs

Index	
`createIndex() -> invertedIndex`	Used to create the inverted index. Also calculates the mapping between docId with scene and play.
`writeMappings() -> void`	Writes the play and scene mapping on to disk for future use.
`writeIndex(invertedIndex) -> void`	Writes the computed inverted index on to disk. Uses user input to determine the compression preference.
`indexBuilder() -> void`	The main index API. Calls individual APIs to perform indexing.
`arrayToPosting(array) -> postingList`	Converts posting array for a term into a usable posting list.
`deltaEncode(invertedIndex) -> dInvertedList`	Uses delta encoding to transform the corresponding docIds in posting array.
`deltaDecode(dInvertedIndex) -> invertedIndex`	Uses delta decoding to retrieve original posting array from an encoded one.
`vByteCompression(dInvertedIndex) -> vByteIndex`	Performs V-Byte compression to compress the delta encoded array.
`vByteDecompression(vByteIndex) -> dInvertedIndex`	Performs V-Byte decompression to obtain a delta encoded array from a V-Byte encoded one.
Retrieval	
`getInvertedIndex() -> invertedIndex`	Used to retrieve the inverted index stored on disk. The APIs called are dependent on the user's compression choice.
`getPostingList(word) -> postingList`	Posting list for the specific word is retrieved using this API.

`generate7Query() -> void`	Generates the 7-word query file by running 100 iterations and selecting 7 random words in each iteration.
`calculateDice(wordA, wordB) -> diceCoefficient`	Calculates the Dice Coefficient for wordA and wordB.
`getDicePair(word) -> dicePair`	Uses computed Dice Coefficients to determine the word with maximum Dice Coefficient corresponding to the word.
`generate14Query() -> void`	Generates the 14-word query file by calculating the dice pair for each word in the 7-word query file.
`documentAtATime(queryTerms, stats, k) -> docIds`	Finds top k documents with highest scores using the Document-At-A-Time algorithm.

II. Q & A

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

- Counts might be misleading features since each scene is different from the other and the high frequency of a word in a particular scene does not necessarily reflect its importance over another scene. Also, scene length is not proportional to the importance of the scene in the play.
- There is also the issue of counting stop words such as articles, prepositions, etc. They introduce a bias and other more important words might be ignored due to their high counts.

This can be fixed in a number of ways:

- Removal of stop words to prioritize high frequency words which are not commonly found in the English language.
- Semantic analysis to give importance to meaning over count.
- Different n-grams can also be used to obtain context for the terms and perform better analysis.

B. What is the average length of a scene?

Average scene length = 1200.5561497326203

C. What is the shortest scene?

Shortest scene = antony_and_cleopatra:2.8, Length = 48

D. What is the longest play?

Longest play = hamlet, Length = 32887

E. The shortest play?

Shortest play = comedy_of_errors, Length = 16426

III. Experimental Results

The experiment performed includes measuring the time taken for the whole system to execute. This includes the creation and storage of indices, computation of mapping, compression, and subsequently the generation of 7-word and 4-word queries, along with obtaining top documents using document-at-a-time. The results are tabulated across two runs as shown:

	Uncompressed	Compressed
Run 1	6m 35.236s	6m 24.446s
Run 2	6m 9.816s	5m 9.453s

```
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$ time python Indexer.py data/shakespeare-scenes.json 0

real    6m35.236s
user    6m33.324s
sys     0m0.784s
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$ time python Indexer.py data/shakespeare-scenes.json 0

real    6m9.816s
user    6m8.909s
sys     0m0.549s
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$ time python Indexer.py data/shakespeare-scenes.json 1

real    6m24.446s
user    6m23.438s
sys     0m0.643s
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$
(base) Shravyas-MacBook-Pro:Indexer shravyakadur$ time python Indexer.py data/shakespeare-scenes.json 1

real    5m9.453s
user    5m8.398s
sys     0m0.677s
```

The experiment is performed twice so as to nullify the operating system's caching effect. If we only consider the 2nd run, we observe that the time taken to perform indexing and retrieval on compressed data is marginally faster.
This confirms the compression hypothesis, which in general hypothesizes that computation using memory (or by extension, the processor) is faster than retrieving from the disk. The implication of this to our system is that we can use the fast processor to derive the complete index from a compressed form of the index, thereby reducing load on the storage. This also goes to say that depending on the speed of each individual's processor, the processor might serve as either an advantage or a bottleneck. If the processor was much slower, then it would be not worth the trouble to compress the index as it would take longer than simply retrieving the untouched index from disk.

The hypothesis is affected if we use a different compression algorithm. If the degree of compression is higher, it might cause more overhead on the processor and it may eventually fail to keep. At this point, the uncompressed data performs better.