An Inverted Index (with Positional Information) on the complete works of Shakespeare

I. Structure of source code

The source code constitutes of 5 packages:

i. reader

This package provides the functionality to read/parse documents into document-store. Its main classes are:

a. **Document** is the generic superclass that all types of Documents in the store must extend. All Documents must have a unique integer document-id regardless of the sub-class that backs the super-class. The Document class also stores a backing document-id which is the id of the backing document and is a string. The backing document-id is left to the sub-class to be initialized.

b. **Scene** is the sub-class that extends **Document** so that the scenes in Shakespeare's complete work can be read. Its id (which is same as the backing document-id for the super-class) is the string <**playId>#<sceneId>**. The unique integer id for the document super-class is the "sceneNum" field from the JSON file, ranging from 0 to 747.

c. **Reader** is the superclass that all **Document**-readers must extend. These sub-classes must also implement the read() method of the superclass depending on what type of document needs to be read. For instance, in our case, SceneReader is a sub-class of Reader that implements the read() function for reading the complete works of Shakespeare from a JSON document.

d. **SceneReader** as described above

ii. compression

This package abstracts the compression/decompression mechanism using 2 classes:

a. **VByteEncoder** (if compression is enabled)

b. **EmptyCompressor** (if compression is disabled).

Both these classes extend the abstract class **Compressor** and implement its two abstract methods: **encodeIntegerList()** and **decodeIntegerList()**.

iii. index

This package provides the "indexing" functionality over the document-store. Its main classes are:

a. **Posting**: A posting is created on a per-term & per-document basis. So, for each term that appears in a document, all its occurrences (positions) are stored in the Posting for that document.

b. **InvertedList**: There is one InvertedList for each term in the vocab. The InvertedList of a term stores a **LinkedHashMap<Integer, Posting>** of doc-ids vs their Posting, the number of docs the term appears in and the collection-frequency of the term in the entire corpus. The LinkedHashMap allows to preserve the ordering of document-ids and their Postings while allowing (near) constant-time access to any Posting in the list.

c. **InvertedFileIndex**: This is the index class that has the inverted-list representation of all terms in the vocab. This class extends the abstract class **Index.** To instantiate an InvertedFileIndex-object, the constructor needs to be passed a String-filename (or location) which could mean two different things based on the usage of the index. If the index is being created for the first time (i.e. from the document-store), this filename is the location to which the newly created index will be written to on disk from memory. If the index is being reconstructed from disk, then the filename points to the path of the InvertedFileIndex on disk.

The important public APIs of this class are:

i) **createIndexFromDocumentStore(ArrayList<Document>)**: creates a completely in-memory index from the document-store.

ii) **writeSelfToDisk(boolean)**: Call this API to write the index to disk (at the location passed while instantiating the index). The boolean argument can be used to control whether Index will be written to disk in compressed or in uncompressed form. The first byte written to the index file is either '**C**' or '**U**' denoting whether the index is a compressed one or uncompressed one. This API also creates a lookup table which has entries of the form: **<term> <start-offset> <df> <cf>** in a line for each term in the vocab. The lookup-table is written to <index-filename>.**ttol** extension (**t**erm-**t**o-**o**ffset-**l**ookup). The lookup is automatically searched for in the same directory as the index's location and hence its name or contents should not be modified.

iii) **createCompleteIndexFromDisk()**: Call this API to reconstruct a completely in-memory index from a file on disk. This API is only for validation purposes such as to test if compressed and uncompressed index (from the same document-store of course) is the same, or to check if the index can be reconstructed correctly from an index-file on disk. The query-retrieval doesn't involve this API but instead only loads the required inverted-list into memory using the next API.

iv) **getInvertedListForTerm(String)**: This API retrieves the InvertedList for a term by seeking to an offset (after reading the offset for the term from the **.ttol** lookup table).

v) **getVocabListFromIndex()**, **getCollectionFrequencyForTerm(String), getNumWordsInCollection(), getNumDocs(), getNumWordsInDocument(int)** and **getDocumentFrequencyForTerm(String)** are pretty self-explanatory APIs.

vi) **compareTwoInvertedIndexes(InvertedIndex, InvertedIndex)**: compares 2 indexes and returns true if they are same, else false. The API goes through all the inverted-lists and the constituent postings to check equality.

vii) **printSelf()**: This API can be used to pretty-print the index for debugging purposes. It calls into the InvertedLists's printSelf() which then calls into Posting's printSelf.

iv. **retriever**: This package provides the query-retrieval functionality including computing the Dice's coefficient for a pair of terms. Its main classes are:

a. **Retriever** is the abstract super class that every Retrieval-sub-class must implement. The constructor takes an object of Index class (of which InvertedFileIndex is a sub-class) – so the retriever can be made to work with any index including the InvertedFileIndex. The two abstract methods that need to be implemented by retriever sub-classes are **retrieveQuery(String[], int)** and **computeDiceCoefficient(String, String)**.

b. **DocAtATimeRetriever** is the sub class that implements document-at-a-retrieval retrieval using the InvertedFileIndex index. To instantiate DocAtATimeRetriever class, an Index (but of the actual type InvertedFileIndex) and the number of documents (in the document store) need to be passed.
Note: To compute Dice's coefficient, the number of consecutive occurrences of A & B in the corpus needs to be computed. Given two postings (of A & B respectively) for a document, **getCountOfConsecutivePositions()** counts the number of consecutive occurrences in the document in O(N) time.

v. **apps**: This is the package with the different applications for index-creation, query-retrieval and evaluation. The applications in this class are:

a. **Indexer**: This application supports the following command-line parameters:

i) **-i <path>**: create index by reading from document store and then writing it to disk at <path>.

ii) **-c**: turn on compression before writing to disk. To be used with **-i.**

iii) **-v <path>**: validation of writing index to disk. Creates an in-memory index and then writes it to disk at <path>. Then creates another index by reading from <path> and calls **compareTwoInvertedIndexes()** to check if the 2 are the same.

iv) **-t <path>:** validate compression. Creates an in-memory index from document-store and writes it to disk without compression to path1. Creates another in-memory index and writes that to disk with compression enabled to path2. Then recreates indexes from disk from files at path1 and path2 and compares them.

b. **QueryRetriever**: This application supports the following command-line parameters:

i) **-r <path to index> <num terms in query or n> <num queries or m>**: Generate m number of n-word queries. The vocab is fetched using the index. It also ensures that the n words in a query are not repeated, although they could be repeated across the m queries. This output can then be used for other evaluations. I store the 700-term list into a member variable *String[] SEVEN_TERM_QUERY_SET*.

ii) **-q <path to index>**: Runs query-retrieval on *SEVEN_TERM_QUERY_SET* , 7 at a time, 100 times. I have saved the results in the file "7-term-query-results.txt".

iii) **-s <path to index>**: Record term statistics such as document-frequency and collection-frequency, longest/shortest play, shortest scene, etc. by querying the index alone. I have saved the results in the file "term-statistics.txt".

c. **DiceCoefficientCalculator**: This application is used to find the highest scoring two-word phrase for each of the 700 terms in *SEVEN_TERM_QUERY_SET*. It only requires the path to the index file on disk as argument. The highest scoring-pairs (700 pairs in total) are saved in a <index-name>.bestpair file along with their Dice's coefficient score.
These 700 pairs are used to initialize another member variable *String[] FOURTEEN_TERM_QUERY_SET* for other evaluation experiments. (I observed that it takes 250K to 270K milliseconds to compute the 700 pairs on my machine).

d. **TimingExperiment**: Takes two arguments: the path to index-file on disk and a string to control whether to run the timing-experiment on 7-term query set or 14-term query set ("7" or "14" needs to be passed as the second argument). The index can be a compressed one or uncompressed one. Given an index, it will be used to perform query-retrieval on the 7-term query set or the 14-term one. Each timing-experiment would invoke 2 runs of query-retrieval on the chosen query-set – the first run to flush out the disk-cache and the second one to get the actual time. This is so that a fair comparison can be done between different indexes by ignoring the effects of reading from disk vs reading from cache on the first run.

This completes the structure of the source-code. The external libraries I am using are:

i. org.json: for JSON parsing of plays and scenes.
ii. org.apache.commons.cli: for parsing command-line options and their arguments and help-formatting.
iii. org.apache.commons.io: for JSON input stream to string manipulation.

Some design decisions and trade-offs:

i. Most of the OOP design that I chose to go with was already discussed above. Overall, my motivation has been that the code/structure is not specific to Shakespeare's works. This is why I have a Document super-class and Scene is one of the sub-classes or a Reader super-class and SceneReader as sub-class. The same with Index and Retriever classes.
ii. I chose to go with a **LinkedHashMap<Integer, Posting>** in my **InvertedList** class, instead of **List<Posting>**. The LinkedHashMap preserves the order in which Postings are inserted, unlike a HashMap, but still provides almost constant-time lookup, unlike a List.
iii. For finding count of consecutive positions in 2 postings (of 2 terms but for same doc-id), I use a 2-pointer mechanism that can find the consecutive-position-count linear time. The implementation details are in the private method getCountOfConsecutivePositions(ArrayList<Integer>, ArrayList<Integer>) of DocAtATimeRetriever class.

II. Why might counts be misleading features for comparing different scenes?

Raw count is a misleading feature because they suffer from a critical problem: all words are assigned equal weights while ranking documents. In real-world scenarios, some words occur too frequently in a document to have any meaningful contribution to relevance of a document. On the other hand, some words which are rare in the corpus have more relevance when it comes to ranking documents. To combat this issue, IDF (Inverse document frequency) is used to scale down the weights of words which appear too frequently in large number of documents and scale up the weights of terms that appear in small number of documents.

IDF of a term ***t*** is defined as:

$$idf_t = \log\left(\frac{N}{df_t}\right)$$

where, N is the number of documents and df_t is the document frequency of ***t***. This weight is then multiplied with the $tf_{t,d}$ i.e. term frequency of ***t*** in document ***d***. So, if a term appears in every document, it gets the lowest scores since its idf_t will be *0*. If a term appears in less documents, its weight will be more and will contribute more to the relevance of the small number of documents it appears in.

III. Some statistics

Average length of the scene: $\frac{\text{index.getNumWordsInCollection()}}{\text{index.getNumDocs()}}$ = 416.28208556149735 words.

Shortest scene: scene-number **549** ("**antony_and_cleopatra:2.8**") with 47 words.

Longest play: "**hamlet"** with 32867 words.

Shortest play: "**comedy_of_errors**" with 16415 words.

All the statistics have been saved to the file "term-statistics.txt".

IV. Experimental results

Using the TimingExperiment app, I documented the time taken on various combinations.
Here are my observations:

Index type (Compressed or Uncompressed)	**Time taken in 7-term query set of 100 queries** (in milliseconds)		**Time taken in 14-term query set of 100 queries** (in milliseconds)	
	First run	**Second run**	**First run**	**Second run**
Uncompressed	179	105	540	248
Compressed	116	70	479	228

** My machine's configuration: Intel i7-8565 CPU @ 1.99 GHz, 16GB RAM, 256GB SSD running Windows 10 OS.

I executed the experiments in the following order:

i. 7-term queries on compressed index
ii. 7-term queries on uncompressed index (this should flush out the caches of compressed index)
iii. 14-term queries on compressed index (this should flush out the caches of uncompressed index)
iv. 14-term queries on uncompressed index (this should flush out the caches of compressed index)

The observations suggest that compressed index results in faster query-retrieval although the difference is of the order of 20-30 milliseconds (105 vs 70 & 248 vs 228). However, the major game-changer is space required to store the indexes on disk. The uncompressed takes up 6,080,105 bytes on disk whereas the compressed index takes up only 1,964,732 bytes which is less than a third of space required by the uncompressed one.

Coming to the compression-hypothesis, which states that it is faster to read compressed data from disk and decompress it in memory than reading uncompressed data from disk – the observations seem to ratify this hypothesis although the differences are very minimal. I feel that a major roadblock in analyzing this behavior more rigorously is lack of understanding/predictability in how the OS caches files on disk to reduce disk access. A better experiment, that would throw more light on the compression hypothesis, would certainly be disabling disk-caching and then performing the query.