COMPSCI 546: Applied Information Retrieval
Assignment 1 Report
Indexer

Name: Nidhi Davawala

System Architecture

Introduction

This project builds an indexer that basically performs the following functions:

1. **Data Fetching:** Fetch and parse the source data
2. **Index Creation:** Map each word/term to a list of documents along with their positions
3. **Query Retrieval:** Generate document scores for the query term and time the retrieval
4. **Term Statistics:** Infer key statistics from the data

Design Choices

The system accomplishes the above mentioned tasks in 2 levels: Index Creation and Query Retrieval. Each of these two packages are described here:

1. **Index Generation:**
    a. *IndexBuilder:* Parse data, generate inverted list, populate maps, write to disk
    b. *Posting:* Represent posting as: (docID, Positions list), operations of posting
    c. *PostingList:* Represent posting list as: (List<Posting>, postingIndex)
    d. *Compression:* Perform vByte and Delta encoding and decoding

2. **Query Retrieval:**
    a. *BuildIndex:* Main class, call to all key methods
    b. *InvertedIndex:* Load maps from disk, fetch posting list, manage retrieval calls
    c. *SelectQuery:* Generate query file with 100 rows of 7 terms
    d. *ComputeDice:* Compute dice coefficients, generate file with 100 rows of 14 terms
    e. *ComputeQueryTime:* Call to query retrieval, measure time
    f. *TermStatistics:* Compute statistics inferred from source data

Please Note: The implementation has been extended based on the architecture shown in class

Challenges

The generation of 14-terms query file requires the computation of dice coefficient of each query term with the entire vocabulary. The initial approach I used computed the postings of all vocabulary terms (~15000) in each and every iteration. This led to a hike in the computation time, with 100 rows-generation taking around 3 hours on my system. To optimize this, I cached the posting lists of each term into a map. Thus my "fetchPosting" method first looks up for the search term in the map, and returns the corresponding posting if available. Only in the cases when the term posting has not been processed, it goes on to call the "computePosting" method which actually computes the postingList for that term and stores it in the map for future use. This reduced my computation time drastically with 100 rows of 14 terms generated within approximately 5 minutes.

Note: The cache was reset when switching from 7-words query file to 14-word query file for fair comparison of retrieval time.

Trade-offs

The key tradeoff in this system is that of storage space and retrieval time. The mappings and lookup map that were written to disk took up space, but it made the query retrieval faster, which is very important. As mentioned previously, the posting list of the terms were cached during independent experiments, which made the retrieval faster and saved a lot of repetitive computation.

Please Note: The implementation has been extended based on the architecture shown in class

API Documentation

Key Methods:

- *parseFile():* Fetches and parses source data file, populates the maps
- *saveInvertedLists():* Compresses the inverted list if needed and writes to disk
- *buildIndex():* Calls parseFile and saves the 3 main maps, along with an inverted list.
- *computePosting():* Fetch offset and number of bytes, read from inverted file and generate posting list.
- *fetchPosting():* Check for search term in cache, call computePosting() if absent or return the posting list
- *generateQuery():* Generate files for 100 rows of 7-terms and 14-terms of query words
- *computeDice():* Compute the dice coefficients for a pair of terms
- *computeTime():* Measure retrieval time for the two query files

Please Note: The implementation has been extended based on the architecture shown in class

Count as misleading features:

Raw count of terms is not always a good measure for ranking documents. Here are some cases when the count could be misleading:

1. Direct matching of words to get its count ignores the presence of the word's synonyms in the document. For example, if the key word is "monkey" and if some document uses the word "apes" instead, then its ranking will be very low and another random document that might randomly contain the work "monkey" will rank higher. This happens as while counting we treat all words to be equally dissimilar, which can be misleading.
2. The presence of stopwords can also skew the scores as they occur in abundance and lead to unfair hike in the score of some documents. It might also happen that all documents would rank almost similarly due to the presence of the stopwords. In tasks when these words don't add much meaning to the sentence, they can be removed. Or we can take a weighted sum of the terms, penalizing the stop words while computing the sum.

Results:

Retrieval Time (in milliseconds)

	7-terms	**14- terms**
Uncompressed	420	717
Compressed	438	657

The compression hypothesis does not hold very prominently in my case. I expected to see a reduced query time in the compressed indexes for both 7-term as well as 14-term query. But this is not clearly seen. Ideally, IO operations are the bottleneck and thus compressed indexes should give better performance during retrieval. But with modern architectures, this restriction is not as valid as it was before as the IO operations are cheaper now. Thus the expected level of speedup is not seen.

Please Note: The implementation has been extended based on the architecture shown in class