Assignment #3

Inverted Index with ranked queries and index compression: Inverted index is a mapping from content(words in this case) to their location in the document. The purpose of this is to provide rapid full word searches at the cost of higher processing time at the time word is entered in the inverted index.

Process for creating inverted index:

I. **Decompress:** To start with we are given a sum total for more than 3 million documents. The documents have been compressed using gzip so first we decompress this using “java.util.zip.GZIPInputStream” library. This allows us to uncompress the file in tsv(tab separated value) format. Note: I was unable to download the TREC format suggested as the download keeps on failing halfway through. Gzip is a lossless compression, so we can reconstruct original data.

II. **Program logic:**

- In this code we will do a double pass of data. The first pass is to estimate the barrel size for lexicon partitioning while generating postings. In this pass we create a HashMap (frequencyGraph) and calculate frequency of each word as well as the number of documents in which the word occurs. This allows us to more precisely maintain allot a dynamic barrel size for each lexicon. We will be maintaining this table in the main memory. For all the terms this table seems to occupy around 1 GB of main memory.

- Next is Term to TermID mapping: We are maintaining a HashMap(termIDS) which maps terms to term IDs. We are allotting term id sequentially. Each term has a unique term id. We are maintaining this data structure in main memory. Using Integer instead of string in the lexicon table allows for compression.

- Next is MongoDB database which maintains document text mapped to document ID, this allows us to query data at last and highlight relevant(search query) data. We are doing this by forming a simple database URL and a collection in it called URLCollect. We are filling the database during the first pass of data.

- Next is Lexicon data which is stored in a HashTable. The key for the hash table is the termID, which we get from HasMap TermIDS. While the value is an array of

“Document frequency” i.e. number of documents the word has appeared in at least once and the location in the inverted index file where the postings for that occurrence is stored.

III. The inverted Index Table:

- We can use RandomAccessFile to write to and read from any point the inverted index file.
- We also fills the lexicon HashMap in which barrel size is determined by the frequency details we gathered during the first pass.
- We iterate through each word in the collection of documents. We then split document ID, URL, title and data on the basis of tab(\t).
- We check if the word is in termsIDS HashMap. If not then, we assign it a new term ID sequentially. We add all the words to two HashMap,
    1. The wordposition has the term as key and an integer array which contains the position of the occurrences of that term in the document.
    2. The mapforpage HashMap which contains frequency of occurrence of each word.
- After iterating a document we have a Map containing each word that appears in it and another Map containing locations of terms each occurrence.

We store data in the form

DocID	frequency	BM25	position1	position2	position3	position4

Difference postings are delimited by **“ > ” (greater than),** while (DocID,frequency) pairs and positions are delimited by a **“ :“ (a colon).** DocID and frequency is delimited by a comma and so are different positions.
Which would look like:
docID,frequency,bm25:locations1,location2,location3>docID2,frequency2,bm25:locations,location2,location3;x

x denotes end for expression

- We start by iterating mapforpage Hashmap, for each term we find the relevant termID in the termIDS map. From this termID we find the location data in the lexicon HashTable. We then seek this address using RandomAccessFile and read a number of bytes based on frequencyGraph. We process the bytes to string. If the string is empty we insert the DocID,frequency pair, getting frequency from mapforpage. This is followed by inserting position data by searching the key in wordposition Map. we add all the locations and delimit it with a **colon**.

- sort using comparator, We are sorting all the appearances of a particular term
  in different documents by using comparators. This is provided with a hashmap of a term and the value contains frequency as first term and BM25 as second term. I have converted BM25 in long by multiplying it by 1000. This gives us a BM25 that is accurate up to 3 digits. By sorting the BM25 and then writing it to inverted index we can extract the top 10 ranked results much faster by just getting the first 10 result while extraction

  If the String is not empty we split the data according to aforementioned delimiters and put it into a temporary HashMap foreachDocument with term as key and an integer array. The first element of the array is the frequency and the remaining the position. We also insert data from mapforpage and wordposition in the temporary hashmap. Now we sort the temporary HashMap according to the BM25 in descending order(first element of value). This allows us to store documents with most frequent occurrence at the forefront. Now we insert back these elements.
  This also allows us to get ranked queries when implementing the search interface.
- The HashTable *lexicon* and HashMap *termIDS* and *frequencyGraph* are stored in file “lexicon”,”termID” and “frequency” respectively.

IV . **For querying data:**

- We read HashTable *lexicon* and HashMap *termIDS* and *frequencyGraph* from file “lexicon”,”termID” and “frequency” respectively.

- We also store data and URL associated with DocID in a MongoDB database. This saves the data in the form of a cluster and provides fast instantaneous search results.
- For querying data we are provided with a search query, we lookup this query term in the termIDS map and find corresponding term ID. We then look at the termID in the lexicon map and find the address of posting information in the inverted index file. We read the data (how much according to frequencyGraph), split it according to the delimiters set and get the first 10 results. Since these postings were sorted during time of insertion we don't have to sort it. Just take the top 10 results, query DocID in the MongoDB database to find the data stored and use location data to highlight the query where we found it.

First Pass:

First pass to determine barrel size for each posting collection. This allows us to create a custom size where detail for a particular term will go. While creating the frequencyGraph hashmap which contains how many documents the term is in and also how many times the term appears altogether in the collection. It also saves DocID, URL and data to a MongoDB database for providing snippets. This is done by the function barrelSize which is executed after the tsv file is extracted.

Snippet Generation:

Snippet is a programming term for a small region of text which provided us a brief for an extract. Web search engines use snippets to provide a context for searched queries. It is usually a quotable package which contains the most relevant occurrence of the searched query.

In search query we are implementing a snippet generation technique. Instead of showing the entire page we are just showing the URL of the page and small parts of text that contains the searched word. This will provide us the context of the query on that particular URL.

We can do this by providing a small enclosed text around every occurrence of searched query with the sentence before and one sentence after that.

Ranked queries:

For calculating the ranked query we are using the BM25 formula which will rank the documents and we will get the top 10 results with the highest BM25 score.
The BM25 formula we are using is :

$$BM25(q,d) = \sum_{t \in q} \log\left(\frac{N - f_t + 0.5}{f_t + 0.5}\right) \times \frac{(k_1 + 1) f_{d,t}}{K + f_{d,t}}$$

$$K = k_1 \times \left((1-b) + b \times \frac{|d|}{|d|_{avg}}\right)$$

Where:
• N : total number of documents in the collection;
• ft : number of documents that contain term t ;
• fd,t : frequency of term t in document d ;
• |d| : length of document d ;
• |d|avg : the average length of documents in the collection;
• k1 and b : constants, usually k1 = 1.2 and b = 0.75

This illCalculates BM25 for a given term in that particular document .The formula stated in the documentation and is used for ranking the queries.

DAAT API in searchQuery

We are processing queries using Document-At-A-Time Query Processing. It processes a query on a document at a time instead of 1 term at a time.
It assumes document-ordered posting lists, reads posting lists for query terms concurrently. This computes score when the same document is seen in one or more posting lists. Top-k results can be determined by keeping results in priority queue.

It includes functions

1. openList() : This read the complete lexicon and URL table data structures from disk into main memory. It also puts the pointer to starting point of our inverted index for that term and loads the prepares for query execution
2. query(): This reads postings from inverted index and provides the top 10 postings ranked according to the BM25 score.
3. closeList(): Closed the inverted list as well as removes all the lexicon maps from memory.

This way, issues such as file input and inverted list compression technique should be completely hidden from the higher-level query processor.

Compressing inverted index

We are compressing the inverted index into zip format. ZIP is an archive file format that supports lossless data compression. A ZIP file may contain one or more files or directories that may have been compressed. The ZIP file format permits a number of compression algorithms, though DEFLATE is the most common. Being a lossless compression we preserves the location information when compressing the file. Using java.util.zip.ZipOutputStream we can compress the file and uncompress it at time of query execution.

Running the program

1. Run the Web.java file which will create the inverted index and compress it to a zip file compress_index.zip.

2. Run ManualCleanUp to remove leftover files .

3. Run searchQuery which uncompress the inverted index and searches query in the inverted index

4. Run ManualCleanUp again to remove leftover files .

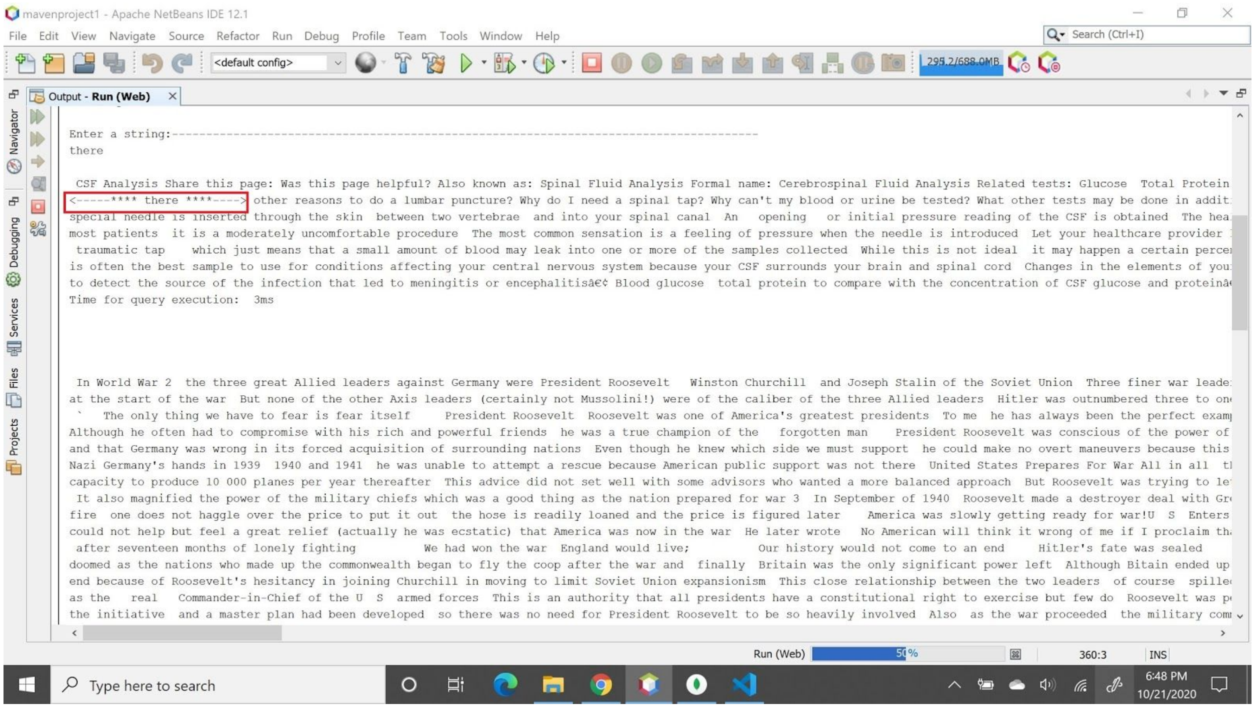
Criteria:

We are unzipping the msmarco-docs.tsv.gz file into a 22 GB tsv file. We will be loading a frequencyGraph hashmap into a file and it will contain the information gathered during first pass. It will contain how much data we need to retrieve to get complete posting information. We are saving this data in a file named frequency which was around 800MB.

We will be saving lexicon in a file named lexicon. This will be a HashTable which will contain term ID, frequency of that term i.e. how many documents did the term appear in and the location of posting for that file in inverted index file.

TermID table, It is stored in termID file and contains mapping for term to termID. This mapping allows for faster execution, finding term and compressing indexes.

Search term “there” being enclosed by ←** **---->



Result generated using snippet generation