**Assignment 3**

**This Assignment is written in python.**

Below are the files

1. readgz.py [Set the the root file location self.rootDirectory]

2. split.py [File size of the desired output sizes ts in main]

3. merge.py [Set s to start and e to the end of file suffixes to be read set memory to the available memory]

4. parserfinal.py

**readgz.py**

This file will generate the temproary inverted index structure[inv\_ind.gz] and Doc\_id postings [doc\_id url count].

Classes in the file :

1. Page: It has two members url and no\_words . URL stores the url of the page and number words in the page is stored in URL
2. FileReader: It the main class that has all the methods and data needed to read the files and write two a temporary index. Explained in detail below

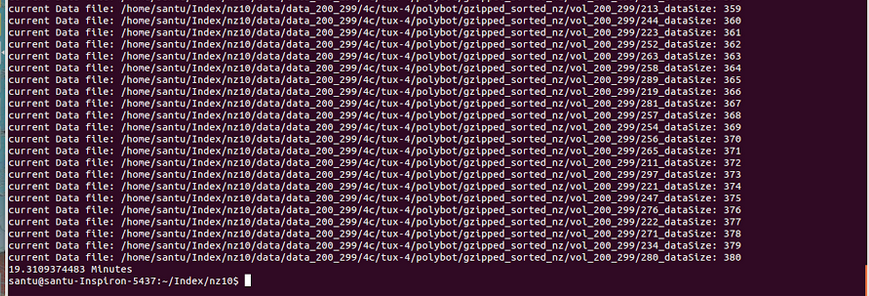
FileReader:

* Data memebers:  
  1. rootDirectory :Holds the path to the directory containing NZ data  
  2. doc\_id : Incremental Document ID   
  3. f\_html: points the opened html file  
  4.f\_index: points to the opened index file  
  5. Doc\_avg : contains the incremental average document size   
  6. Page\_table : holds all the pages url and count parsed finally written to a file  
  7. Temp\_index: Contains temporary unsorted index and written back to the file after reading 4 index files
* Functions:  
  1. writeDoc: Writes to the doc.gz the [doc\_id url count] from page\_table , It also writes anything remaining in temp\_index to inv\_ind.gz and also writes to docmaavg.txt the max\_doc\_id and average doc length   
  2. addToHash : Find the postings in the hash table and appends to postings the word and it count if not found create a new posting  
  3. process\_tok: process the tokens returned from parser and create small hash table containing the word and counts   
  4. writeToIndex: Write from hash table to inv\_ind.gz and truncate the hash table called after every 4 index files  
  5. validToken: returns true for valid tokens could be enhanced
* Main:  
  1. We have created FileReader class. In main, we will first instantiate FileReader class. debug\_set is set to false, so that we can write it to compressed file i.e. Gzip file.

2. We will first read all the files from data directory and validate the index and data file names with regular expression.  
3. We will read the index file and corresponding data file together, parse it and write it into the final inverted index file, 'ind\_inv' for every 5 index,data files.   
4. We have used the C parser given by you and modified by viraj shah.  
5. We will call process\_tok function, which puts all the token to hashtable. The key of this hashtable is the token and the value is the count of the token.   
6. We have assigned the docId in the order the pages are parsed.  
7. We will finally write the docID, URL and number of words into a compressed file 'doc.gz'. The Final inverted index structure will be in the form, word docId number\_of\_words.

**Results:**

readgz.py will take 19 minutes complete for 10% [nz10]of data.



Final output contains:

1. Doc.gz contains url details size:2.89 MB
2. Inv\_ind.gz contains unsorted index postings 194 MB
3. Docmavavg.txt contains max\_doc\_id and average doc length

**Changes from previous submission:**

1. Code to print the time
2. Calculate max\_doc\_id and doc avg and write it to a file

**split.py**

This python code will split the actual inv\_ind file into several files of 60MB each and sort each file using unix sort.

**Class:**

Splitter: This the only class in this file it has the following members and functions

* Members:  
  1. file\_sie contains the actual file size of inv\_ind   
  2. temp\_size contains the max file size generated files.
* Functions :  
  1. sort\_temp: Sort the files in temp which are uncompressed using UNIX sort and after the sort compress and write to other file.   
  2. main:
  + This code will first open the temp file, read each line from index file, check if temp has reached 60MB [NZ10] and append the line if it has not yet reached the maximum size.
  + After splitting file, each file is then sorted using UNIX sort. Finally it will return the number of temp files created.

**Output:**

For NZ10 the number of file produced was 12 [i.e. temp0 to temp11]. Time taken is 6 minutes.

Each file size after compressing is approximately 17.6 MB



**Merge.py**

Merge.py will merge all the temp files created by split.py code. This code will even create lexicon structure.

**Classes:**

HeapItem:

**Data members:**

1. tok: the word which the heap item represents
2. Invitem : Postings of the token read from file
3. Fl: File from which it was read
4. is\_last : Set to true for the last item read from file

Merge:

Data members:

1. outputBuffer: output buffer, written to file when the data reaches certain size
2. of : File pointer to the final compressed index structure output
3. lexf : File pointer to the final compressed lexicon structure output
4. r: Total number for temp files from split.py
5. sz: total size of the buffer/(r+1)
6. chunckSize: Size of the chunks of compression

Functions:

1. readInput: reads files of size sz into input buffer
2. constructHeap: constructs heap from the data read from input file and for the last item sets is\_last to true so when processing whenever this is hit ,again the data is read from input
3. write\_final: Writes to final output index whenever the output buffer is full ,it also merges two lists of same token
4. chunckCompress: Compresses each posting , First it splits the data in chunks of 20 [doc\_id count] and for the first chunk the doc\_id is left as it is [for uncompressing the chunk] and for the rest difference with the previous is noted as the doc\_id.  
    Sets data for lexicon so no need of extra parse
5. nwaymerge: Takes start and end as parameters which indicates the range of files to be read with the prefix as file\_prefix [ex: if file\_prefix is set to temp and s=5 and e=10 then it merges files temp5 to temp10] ; Memory indicates max memory to be used and final\_file indicates the prefix of the output index and lex file .  
   Proper setting of s,e ,file\_prefix and final\_file this can be called repeatedly in main to do a nway merge.

**nway More detail:**

At first, we will initialize Merge class which will have 1 output buffer, file pointers of Lexicon file and sorted Index file. Here, we are using r+1 files, r for input buffer and 1 for output buffer. In the first step, we will read data from all input buffers, construct the heap and finally get the top value until heap is empty. write\_final method will create a lexicon structure.

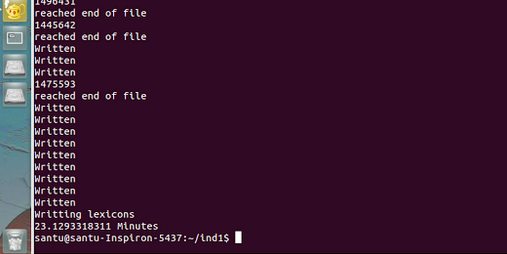
For our project we tried out different combinations of merge and found that 1 pass merge was the fastest with memory size set to 20MB

nwaymerge(0,11,20000000,'temp\_','file\_prefix')

Output:

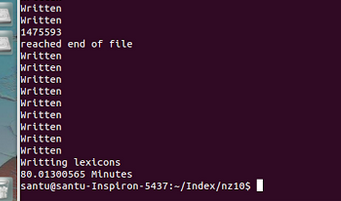
Without compression:

File\_perfixindex size was 230 MB and time taken was 23 minutes for NZ10



With compression:

File\_perfixindex size was 230 MB and time taken was 23 minutes for NZ10:



**Changes from previous time:**

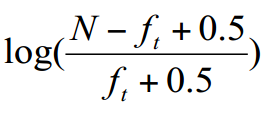
1. support for chunk compression
2. nway method when crafted properly can be used for any way merge
3. Display the time taken
4. Experimented with different data sizes and merges to find one of the optimal ways to merge

**parsefinal.py**

parsefinal.py is the final script which takes the query from the end user as the input and returns the top 10 results. After it displays the top 10 results, the program will wait for the next query to be input.

Upon startup, parsefinal.py file loads lexicon structure, docmavavg.txt and doc url structure into main memory.

**file\_prefixlex** file is the lexicon structure which is loaded into main memory. While loading, it calculates the logarithmic part of BM25 ranking formula and appends it into each query word in lexicon structure.

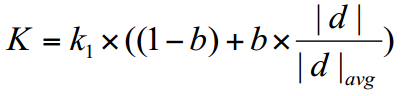


lv = math.log10( (float(maxdocID)-ft+0.5)/(ft+0.5))

The final Lexicon Structure loaded will contain the term, location in bytes, frequency and the above logarithmic value.

**docmavavg.txt** file will contain the maximum doc id and the average doc length.

**doc.gz** file is the doc url structure which is loaded into main memory. While loading, it calculates the k value of the BM25 ranking formula and appends it into each doc entry in the doc url structure. It uses the maximum doc id from the docmavavg.txt file while computing. The formula is given by



The final Doc url structure loaded will contain the doc id, url, word count and the computed K value.

After all the above is loaded, we will input the search string from the user. We use DAAT query processing technique for the query processing. We use SearchQuery object to find the top 10 results.

SearchQuery class in the code has the following methods

1. Openlist(self,pos,freq)
2. Closelist(self,lp)
3. getFreq(self,lp)
4. nextGEQ(self,listobj,k)

We have used a class called list, which will have the following properties.

Term - Stores the term or query word

Frequency – Stores the number of documents containing the term

Valuelist – stores the list of (document id, count)

Seek=0 – Initially Seek is set to 0. Used in DAAT Query processing.

Now lets go into each methods.

1. openlist(self,pos,freq)

This method will open a compressed inverted indexed structure, creates a list object with all the fields set and returns the list object.

The inverted indexed structure is compressed using chunk compression algorithm. About chunk compression algorithm, it compresses each posting  , First it splits the data in chunks  of 20 [doc\_id count] and for the first chunk the doc\_id is left as it is [for uncompressing the chunk] and for the rest difference with the previous is noted as the doc\_id.

This method will directly seek to the pos location and gets the term and list of compressed values. This will not load entire index structure into main memory. The seek value of list object will be set to 0 initially.

1. Closelist(self,lp)

This method will empty all list values and makes memory free for the next search query.

1. NextGEQ(self,listobj,k):

This method will return the next greater than or equal to ‘k’ item in the listobj. Since the indexed structure is compressed, we should decompress it and find its value. Initially it will not decompress all the chunks. Instead it will find in which chunk the k object is and then decompresses only that chunk, find its next document Id and returns it. Each chunk size is 20. So in worst case, 20 comparisons are required to find its nextGEQ element and ‘n-1’ chunk comparisons to find the actual chunk.

1. getFreq(self, lp):

This method will return the frequency of the term t in document d. The document id is given by the current seek position.

When user enters a search query, we will split the string into terms and call openList() function for each term. We will use the loaded lexicon structure to find the index position in the compressed index structure for that word. Thus for n words or terms, we will have n list objects each containing the term’s frequency, term’s valuelist and term’s seek position which is set to 0.

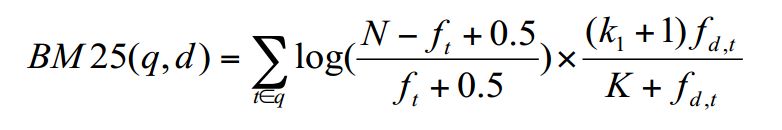
After having the list of listojbects, we will sort them on the basis of the length of valuelist. Finally first object in list of listpointers will have smaller valuelist and last object in list of listpointers will have largest valuelist. This will help in improving performance while doing DAAT.

This is how DAAT works

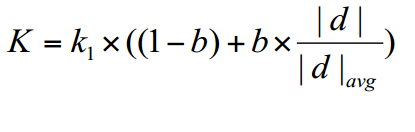
* Open all 3 inverted lists for reading using openlist() function.
* This returns 3 pointers lp0, lp1 and lp2 to the start of the lists.
* Call d0=nextGEQ(lp0,0) to get docID of first posting in lp0
* Call d1 = nextGEQ(lp1,d0) to check for matching docID in lp1
* If (d1>d0), start again at first list and call d0 = nextGEQ(lp0,d1)
* If (d1=d0), call d2 = nextGEQ(lp2,d0) to see if d0 is also in lp2
* If (d2>d0), start again at the first line and call d0 = nextGEQ(lp0,d2)
* If (d2=d0), then d0 is in all the three lists; compute its score, then continue at first list and call
* d0 = nextGEQ(lp0,d2+1)
* Whenever a score is computed for a docId, check if it should be inserted into heap of current top-10 results; at the end, return results in heap.

In the above, when we find the common docId in all the three lists, we will calculate its score. Score is calculated using the popular function BM25.

The BM25 function is given below.



Where K is given by



And

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

**Screenshots:**

