IRE Mini Project Phase 2

Roll no: 2019114006

Name: Suyash Vardhan Mathur

Directory structure

```
.
├── index.sh
├── indexer.py
├── main.py
├── page.py
├── Readme.md
├── Readme.pdf
├── search.py
├── search.sh
├── stats.txt
└── xml_parser.py
```

- index.sh: The main file that is used to create index.
- indexer.py: File that contains code for writing and creating to temporary index files, offsets, etc.
- main.py: Main file used for indexing.
- xml_parser.py: File responsible for parsing and handling the XML dump.
- search.sh: Main file used for searching.
- search.py: File that performs search functionality.

Running Instructions

Run the command below to index

```
bash index.sh <path_to_wiki_dump> <path_to_inverted_index> invertedindex_stat.txt
```

To search with a given index directory, run the following command:

```
bash search.sh <queries file path> <inv-index>
```

Files created

First, a temporary index is created for **each fields**. Thus, separate files exist like **index_b0, index_b1, index_t1, etc.** After this, these temporary index files are combined into one using k-way merging, such that all are sorted alphabetically. Heaps was used for the entire process of merging. The frequency is encoded in 64-base to save space. The format of the final index is:

```
<word1> <document_id1>:<frequency1> <document_id2>:<frequency2>
```

Further, temporary files containing document frequency in last ~50k documents are also created. These contain each token, and with it, number of documents in which that token appears. These are processed document-wise, and the data is dumped after every 50k documents. Then, all these files are combined to calculate the total number of documents in the entire dump where the token occurs. Then, IDF value is calculated, rounded up to 5 decimal places and stored. The format for the **IDF files(idf_0, idf_1..)** is:

```
<word1> <idf-1>
```

Also, a document ID to document title mapping files are created(**title0, title1, …**). This is also sorted based upon the IDs, which are encoded in 64-base encoding. The format for this Document mapping is:

```
<document-ID1> <Document Title>
```

A pre-index file also exists for each of the fields of the inverted index. These were created so that searching for the file containing a particular word becomes faster. The pre-index contains first word in all the files in each line. The respective preindex files are **preindex_r, preindex_t, etc.**. The format for preindex files is:

```
word1
word2
word3
```

A similar preindex exists for document title files(**title_pre_index.txt**). The file contains the first document ID in each document title file. The format of this file is:

```
title1
title2
title3
```

Another preindex of IDF files exists(**idf_preindex.txt**), which contains the first word in every IDF file. The format of this file:

```
word1
word2
word3
```

Another set of files created are the offsets (**offset_i0.txt, offset_b9.txt…**). In each line, these contain the number of characters in that particular file(**index_i0.txt, index_b9.txt**). The format is:

```
<number of characters in line 1>
<number of characters in line 2>
```

Optimizations done

To reduce index size:

- Used 64-base encoding for numbers(document IDs and word frequencies). Reduced numbers from 5-6 digits to 2-3 characters.
- Removed stopwords. Since these are present in almost every doc, reduced the size by a factor of (number of stopwords)*(number of docs).
- Stemmed all the words. Reduces size drastically since it combines various forms of the word while retaining the core meaining.

- Removed unncessary documents like *File:, Wikipedia:* pages. These weren't considered files that would be relevant search results.
- Removed all special characters. Also removed words that contained both letters and numbers like a00. Reduced the size drastically since such strings are meaningless and highly present in online dumps.
- Removed words that were of either very large length or very small length. Reduced the size drastically, since number of stray non-meaningful words with vv(1 or 2) small length are quite high.
- Created different files for all fields rather than encoding this information in the inverted index itself. Reduced 1 character space per document per word.
- Rounded IDF values to 5 significant places. Reduced the space that would otherwise be used by 11+ significant places.
- Rather than creating an offset for title files, directly loaded the relevant file in memory and used O(1) retrieval based upon some mathematics. Save the space of adding offset for title files.

To reduce indexing time:

- Used fstring instead of +, join or other string concatenation operations. This is because other operations are extremely slow, by an astounding factor.
- Reduced the number of unique words and documents to be processed as described above, reducing the indexing time.
- Used split to parse various sections of the dump rather than regex. This is because split is much much faster than regex.
- Used inline intitalisations and declarations since those are faster.
- Wrote the files only after 50k words had been processed/15k titles were processed. Reading and writing every time would've been slower.
- Used heaps to do K-way merging of index files, IDF files. This reduced the complexity from N*(num files) to N*log(num files).
- Used plain dictionaries instead of defaultdict, etc. since those are much faster.

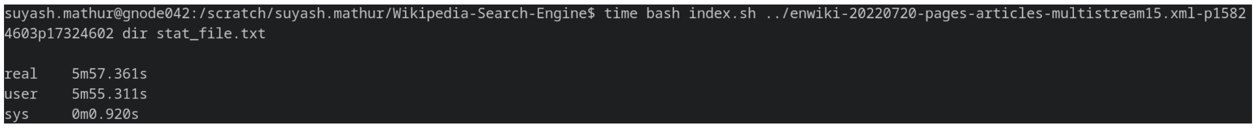
To reduce search time:

- Used pre-index to find the file where the word would occur. For this to happen, had to merge to make words in all the files sorted globally. Reduced the time for every file opening and reading first line.
- Used binary search on the pre-index words to find the relevant file index instead of linear search. Reduced the time from O(N) to O(1).
- Used offsets to find the entry of the word in the inverted index file through **seek** which is implemented in C and hence fasrt. Reduced the time from O(N) to travel to ith line to O(1).
- Used binary search on the words in the file too to retrieve the word from the inverted index faster. Reduced the time complexity from O(N) to O(logN).
- Used pre-index to find the file where the title information of a particular document ID is stored. Reduced the time for every file opening and reading first line.
- Used O(1) operation to find the line where the document title for the document with the particular document ID would occur. For this to be done, I subtracted the document ID(base 64) from the preindex document ID(base 64), and didn't search linearly or binary for it. Reduced time from O(N) to O(1).
- Kept separate files for all fields to make field queries faster. Reduced time by lot when the number of documents of that field would have been much lesser.
- Cached IDF values in a dictionary so that for a general query, IDFs aren't continuously retrieved for all the fields. Reduced time by factor of 7 times of reading IDF files.
- Used pre-index for IDF values to find the relevant file faster. Reduced the time for every file opening and reading first line.
- Used heaps for finding the top 10 search results instead of sorting the entire query results. Reduced complexity from O(N*logN) to O(N*log(10)).

Benchmarking

Phase-1 stats

I ran the exact submitted code on ADA using 8 CPUs and 0 GPUs on gnode042. The total runtime is below:



```
suyash.mathur@gnode042:/scratch/suyash.mathur/Wikipedia-Search-Engine$ time bash index.sh ../enwiki-20220720-pages-articles-multistream15.xml-p1582
4603p17324602 dir stat_file.txt

real    5m57.361s
user    5m55.311s
sys     0m0.920s
```

The total index-size, including all Document ID hashing and vocabulary files, comes out to **237492 bytes** which is much lesser 1/4th of the given file size. The stat files can be created, and give output:

```
2064728
1580369
```

Phase-2 stats

The time for running the code is almost same. On lab PC(i5), it takes 358.36 seconds for indexing to be done. The total index size is **298592 bytes,** which is lesser than the 1/4th limit.

The entire dump indexing is still processing, and should be ~18 GB in size. The time anticipated is ~20 hours.

Code Explanation

Indexing:

- The temporary index is created for every ~50k documents per field containing word with the document IDs and its frequencies in it.
- Similarly, document frequencies are also stored.
- Simultaneously, document titles are written, as well.
- After temp index is created, k-way merging is done with Heaps. Simultaneously, offsets also get created and stored.
- In a similar fashion, document frequencies are also combined, and then IDF is calculated and stored in files.
- At the end,. preindexes are also written in a single go.

Searching:

- First, the query is tokenized and cleaned up and stemmed.

- Then, the file number containing the particular word in retrieved using binary search on preindex.
- Then, using offsets, binary search is done using seek within that particular file to retrieve the doc IDs and TFs for the word.
- Then, the file containing the word's IDF is found using binary search on the IDF preindex.
- Then, the file contianing IDFs is loaded directly into memory, and binary search is done on it to find the IDF for the word.
- Now, every document is scored by summing TF*IDF for all words, and taking a weighted summation based upon the fields in which the token was queried. This was done since all fields have different importance, like title and infobox have more importance than body, which in turn has more importance than references, eternal links.
- Now, heap was used to find top 10 document IDs.
- Now, their titles were retrieved. For this, first the file containing the Document ID was retrieved by doing binary search on the title preindex.
- Next, the exact document was found in O(1) by finding which line it would occur on = difference of document ID and the first(preindex) document ID.
- Then, the relevant document titles are found and written to the output file.