Scalable Inverted Index Creation

- Main challenge is to build a huge index with limited memory.
- Sort-based Method
    1. Build local inverted index
    2. Merge local inverted index
    3. Obtain a large inverted index

To merge the partial indexes, K way merge sort is used. Heap data structure is used for that.

PreIndex - Document sorted
Final Index – Dictionary sorted

Frequencies stored

1. Term frequency for every term in every document - stored in the field-wise index.
2. Document frequency for every term – stored field-wise in the vocab file
3. Number of tokens in each field of every document (used for document length normalization) – stored in pageStats

Ranking

TF-IDF – term frequency and inverse document frequency

BM25 – document length normalization and term frequency normalization

Dependencies – Only NLTK

File Structure (field – b/t/c/i/r/l)

Indexer Files

1. createPreindex.py
2. indexerMerge.py
3. createPageStats.py
4. splitPageStats.py
5. pickling.py
6. computeAveragePageLength.py

1. mergedvocab[first letter of terms in that file].txt

The vocab file contains the file number to be referred to for a term's index (for each field).

Term	Total Term frequency in corpus	Unique docs						File number					
		b	t	c	i	r	l	b	t	c	i	r	l

2. pageStats[first 2 letters of PageId in that file].txt - consists of id-title mapping

Page ID	Number of tokens in that field on that page						Title
	b	t	c	i	r	l	

3. [field_id][File number].txt

Term	Page ID	Term frequency

Search at Runtime

Following are the sequence of steps followed for returning a set of results matching the search query -

1. User enters a search query in the form of a set of words, the query may contain specific field names which should be searched.
    - Tokens are extracted from the search query after stemming and stopwords are removed
2. When a query is received, the search algorithm first has to identify the index files which must be retrieved since the index is stored in multiple files to reduce file size and for faster access.
3. Field-wise file numbers are stored in the "Vocabulary" file which is sorted by the terms. Even the vocabulary file is a large one and is hence split by the first characters of the terms. First, the offset file relevant to the terms is retrieved. For example, for the query “Lata Mangeshkar”, the offset file corresponding to “l” which is “vocaboffsetl” is retrieved. Offset files are stored in Pickle format for fast read access.
4. First of all, the "Offset" file is loaded into memory. Splitting the vocab files into smaller files helps reduce the memory requirement here and also reduces the load time for the offset file. The offset file enables random access through the Vocabulary file and using the offset file, a binary search is performed on the corresponding vocab file “mergedvocabl.txt”.
5. The record in the Vocabulary file, stores the file number by each field. For example, for “Lata” the record is “lata 12270 4874 244 39 335 58 96 240 28 4 44 16 28”. The first field captures the number of occurrences of the term “Lata” in the corpus. The next 6 numeric data points - “4874 244 39 335 58 96” capture the number of unique documents in which “Lata” is present given an indicator of the popularity of this term in different fields in the Wikipedia corpus. The next set of fields indicates the file numbers in which the term “Lata” is present for each field “240 28 4 44 16 28”

   The order of fields in storage is in sequence
   FIELDS = { b : 0, t: 1 , c: 2,i: 3,r: 4, l: 5 }

6. Once file numbers for each of the fields in which the word is present are known, a list of documents is retrieved from the index file (specific to each field). For example, for retrieving the documents in which “Lata” appears in the field “Body: b”, the corresponding index file “b240.txt” is retrieved. Here again, the offset file corresponding to the field and the file is retrieved first – “offset_b240.pkl”. The offset file is stored in pickle format and is loaded in RAM. To give a comparison of size, b240.txt is 80MB+ while the offset file offset_b240.pkl is 589KB. A binary search for the word “Lata” is performed using the offset file. An extract from the index file b240.txt for the word “lata” – only a part is shown since the record is very large.

lata Au 1 Lq 2 MW 1 _D 1 0GX 1 0T7 1 0cR 2 0e6 1 1DK 1 1FO 2 2o2 1 2qR 1 2sa 1 2t1 1 3+W 1 3N+ 1 3Oq 1 3gl 1 3gm 1 3nT 1 4SI 3 58z 1 5Na 1 5Nb 1 5hS 64 5hW 1 6+k 1 6LV 1 6dh 2 6jx 1 72Y 6 7AZ 1 7Ag 1 7fK 4 8cY 2 B39 1 EYa 3 Ejm 2 F7a 1 FdU 2 NKI 1 OgH 2 Oga 2 OxE 1 PVp 3 Q33 1 R5M 1 ROI 1 S5Z 1 Sro 1 VF6 1 V_n 1 VbA 1 Vlr 25 WBh 1 Whm 4 Wt1 1 Yo9 1 Yol 1 Zbz 1 Zm3 2 _Vh 1 _WP 1 _vN 3 aFu 1 aK6 7 aiz 1 b6Y 2 bPT 1 bU3 1 bsP 2 cSq 1 cT9 1 cXV 1 dPx 26 dQ2 26 dVK 1 e23 3 gIB 1 hEq 1 iAp 1 ifi 1 ilo 1 j+_ 3 jaO 2 kXH

1 lGE 1 m7T 2 m7c 2 n0a 1 nhG 1 oNa 5 oS5 1 ocu 1 ogE 1 p15 1 q+V 1 qU5 1 qdN 1 rKJ 16 rKY 16 rQE 1 rcH 2 sLe 1 sXe 1 sgi 1 sh2 1 t5p 3 tVr 2 uEO 4 uFJ 3 uPS 2 uPT 2 uiZ 2 vCO 1 vDv 3 vXK 1 v_n 1 vbg 1 vcP 1 vxl 1 wo5 1 xWo 1 xXK 1 xct 1 yAn 4 yjO 3 yjf 3 zcP 1 0+Ko 1 0+Rl 2 00Xi 3 01ux 1 02HB 1 03Ms 1 03Qj 1 04BE 1 05Mn 1 065t 2 068T 1 06h_ 1 07As 1 07t5 1 08St 1 09tb 1 0Ade 1 0B5d 2 0BlS 6 0CxV 1 0EQ4 1 0Ems 1 0Ewx 2 0H1A 1 0H9o 2 0Ieo 1 0J0Q 2 0JkR 1 0Jrh 1 0K4R 3 0Kt5 1 0KuX 1 0KuY 1 0LOH 2 0LwS 1 0MeE 2 0Mt4 2 0OU6 1 0QiN 3 0Qlh 2 0RJb 1 0Sve 1 0T6I 1 0TZP 3 0TpM 5 0UAQ 1 0VbH 1 0WAw 1 0WVx 2 0WVy 5 0Wgw 1 0XAh 1 0Xdl 1 0Xjj 1 0Ya0 1 0YoL 1 0_xs 1 0a3+ 1 0a36 1 0aq+ 1 0bWr 2 0bcN 3 0dNd 1 0g0h 5 0hXM 1 0lSB 1 0lZt 2 0ldr 8 0rFd 3 0tN8 2 0vVS 1 0xWG 1 0y2Y 1 0y8X 1 0yAV 1 0ybt 1 0yy0 2 0zoQ 3 1+Cm 1 10Ke 1 14ZN 1 14_6 1 14b6 2 164d 1 16ML 1 16ly 1 17yG 1 1980 2 19cH 1 1APl 1 1Can 1 1D5H 1 1EI7 1 1EXb 1 1E_8 1 1Eof 5 1FMz 3 1Fgv 1 1Fh2 1 1HlX 1 1IdG 2 1Lka 1 1N7T 2 1PrV 2 1Qyo 2 1R3p 2 1RLy 2 1SWy 1 1TBz 6 1TpK 2 1U3X 1 1V1C 1 1V3+ 1 1Viw 1 1Vl4 2 1W5c 1 1WfP 8 1XjE 1 1_nE 2 1a2h 1 1a5a 1 1b8j 1 1boY 1 1czX 2 1dAm 2 1dM+ 1 1ebb 2 1fZu 1 1gna 1 1gvP 9 1j9J 3 1jNk 1 1kY5 1 1muo 1 1oGT 1 1pBA 1..... 240 page ids with term frequency information

7. Index file also returns the frequency of the term present in the document along with the list of documents. In phase 1, only the list of documents/pages by the field was presented as results. In phase 2, the documents are ranked basis fieldwise weights and frequency of word occurrence in the fields.

8. For ranking different documents, all documents corresponding to [term][field] pairs are retrieved. That is 6 sets of documents each representing the field in which the term ”Lata” appears are retrieved.

9. In the ranking algorithm, we have 12 sets of documents – 6 each for “Lata” and “Mangeshkar”. First of all, we measure the weightage of each term – Lata and Mangeshkar for each field, less popular means the term is rare in the corpus and the presence of that term on the page should be given higher weightage. Weightage is computed for all the 6 fields, that is 12 different weights are computed. Mangeshkar is rare in the Body field as compared to Lata and has a higher weight of 9.15 vs. 8.38 for Lata. This is true across most fields except for “Category” where they are very close in terms of weight.

weight for lata = [8.38, 11.37, 13.20, 11.06, 12.80, 12.30]
weight for mangeshkar = [9.15, 13.30, 13.30, 12.25, 14.43, 13.81]

10. Each of the 12 document sets is now scored with weights, term frequency, and a field-specific factor which is to indicate how important that field is. Please see below FACTOR = {b : 0.2, t: 0.4, c:0.1, i:0.2, r:0.05, l:0.05}

11. Final results are sorted by the cumulative score. To present the results, the Title of these documents are retrieved and presented as results in order of declining scores. Since the title is also a large file, it is split into smaller files basis the pageID (first 2 characters are mapped to a file name). Similarly, offset files are maintained for retrieving the title basis the pageID that is stored in the Index. Random access through offset files is used for loading the title file.

+Em_2 919 5 11 92 2 0 List of awards received by Lata Mangeshkar 12.81
+wmWv 471 2 10 34 6 0 Mangeshkar family 10.2

+tdkU 108 3 6 0 0 0 Template:Lata Mangeshkar 9.21
+3Hq2 567 2 42 88 1 4 Deenanath Mangeshkar 8.86
090fm 591 4 0 0 1 0 Wikipedia:VideoWiki/Lata Mangeshkar 8.29
++aK6 485 2 37 65 0 3 Hridaynath Mangeshkar 8.15
+lOGE 490 3 10 0 1 0 Lata Mangeshkar Award 7.97
+ji4C 857 4 31 0 29 19 List of songs by Lata Mangeshkar 7.58
+BHJ7 437 2 60 57 1 3 Usha Mangeshkar 7.37
+t46c 269 3 16 308 1 2 Mehndi Rang Lagyo 7.36
+xOO6 46 11 0 0 0 0 File:Suraiya with music composer Madan Mohan, wife Sheila and Lata Mangeshkar.jpg 7.28

12. In the example above, the last number in each of the lines is the cumulative ranking score. Most of the pages returned in the top results have Lata or Mangeshkar in the Page Title. The first part of the line contains the pageID and the number of terms in each field of that page. This is what was tried for document length normalization. However, the results were not very encouraging and there was no justification for the time taken to do the normalization.

13. The way normalization was attempted to get the Top 200 results in step 11 and then use the document length to normalize and re-rank the top 200 results. However, a substantial amount of time was taken to retrieve the 200 page titles and their document lengths. Hence, the document length normalization was not included in the final results.

Results after document length normalization – Page Titles

- Wikipedia:VideoWiki/Lata, Mangeshkar
- Kahin, Deep, Jale, Kahin, Dil
- Yaara, Seeli, Seeli
- RSIPV, Lata, (03)
- Lag, Jaa, Gale
- Lata, Bhagwan, Kare
- Latas
- Tujhse, Naaraz, Nahi, Zindagi
- Lata, Brandisov
- Khodeza, Emdad, Lata

Indexing of Large XML Dump of Wikipedia

Indexing of large XML dump of Wikipedia was challenging because of its sheer size and also the breadth and diversity of the topics it covers. With more than 2crore and 13 lakh pages, the corpus covers most of the key concepts existing. The main advantage of the corpus is that is available in XML format.

XML Parsing
XML Parser was used to scan through the dump. Each page had clearly demarcated element boundaries for Title and Body Text.

```
 <page>
    <title>McCabe Lake 45 </title>
    <ns>0</ns>
    <id>23570393</id>
    <redirect title="List of lakes of Nova Scotia" />
    <revision>
      <id>958843060</id>
      <parentid>956397680</parentid>
      <timestamp>2020-05-26T00:14:09Z</timestamp>
      <contributor>
        <username>Reywas92</username>
        <id>1233313</id>
      </contributor>
      <minor />
      <comment>Redundant, mere name on map fails [[WP:NGEO]]</comment>
      <model>wikitext</model>
      <format>text/x-wiki</format>
      <text bytes="41" xml:space="preserve">#redirect[[List of 45 lakes of Nova
Scotia]]</text>
      <sha1>ftjrqwc7jc1ny6rl8vrpvl9jl709ob0</sha1>
    </revision>
  </page>
```

The length of the body text was available as an attribute of the XML tag. During the exercise in Phase 1 of the mini project, a 90,000 lines long Body text on a page was encountered and it became apparent that special handling of large pages is needed. The large size of the body Text made the string processing very slow in the case of large pages. Based on page length, processing was broken into smaller steps and the same task was accomplished in a shorter period.

```
if self.numTextBytes < 50000:
        self.text += content
else: # append in multiples of 3000
        self.currenttext += content
        if self.scounts > 3000:
                self.text += self.currenttext
```

The page content was processed, whenever the end of the page was encountered.
A 64-bit encoding of the pageCount was used for PageID to reduce the index size.

```
ID = num_encode(pageCount)
title, body, info, categories, links, references = d.processText(self.text, self.title)
i = Indexer( ID, title, body, info, categories, links, references)
i.createIndex()
```

Process Text
ProcessText takes the Body Text as input and identifies the section boundaries for Category, Infobox, References, and External Links.

For example,
References section began with “==References==” as the separator
External links section began with “==External links==” as a separator, while the line containing the links began with “* {{“ or “*[[“

Category data was present anywhere in the body, with clear separators like
[[Category:1984 births]]
[[Category:Zambian footballers]]
[[Category:Zambian expatriate footballers]]

Infobox begins with “{{Infobox” and ends with “}}”

```
{{Infobox body of water
 | name = Six Mile Lake, Nova Scotia
 | image =
 | caption =
 | image_bathymetry =
|pushpin_map=Nova Scotia
 | caption_bathymetry =
 | location =[[Halifax Regional Municipality]], [[Nova Scotia]]
 | coords = {{coord|44.651554|N|63.712414|W|type:waterbody|display=title,inline}}
 | type =
 | inflow =
 | outflow =
 | catchment =
 | basin_countries = Canada
 | length =
 | width =
 | area =
 | depth =
 | max-depth =
 | volume =
 | residence_time =
 | shore =
 | elevation =
 | islands =
 | cities =
}}
```

Using the markers for each field, text corresponding to each field was identified and then stop words were removed and strings tokenized and stemmed.
Before this, special characters and extraneous data related to HTML links/urls were removed. Numbers were retained to enable searches on dates and other important milestones. To mention here, re.sub for removal of these characters is very efficient before the tokenization – an attempt to do this after tokenization severely impacted the time taken to remove these characters.

```
data = re.sub(r' |&lt;|&gt;|&amp;|&quot;|&apos;', r' ', data)
```

Stopword removal and stemming both were time-consuming operations because of the need to go through the entire text length term by term, hence both of them were combined in a single pass by loading all the stopwords in memory.

```
stemlist = []
for t1 in data:
        if stop_dict[t1] != 1:
                stemlist.append(stemmer.stem(t1))
return stemlist
```

ProcessText returns a list of stemmed tokens (term) for each field.

Pre Indexing – Inverted Index for a set of pages
Indexer takes this input and then computes the frequency for each term in each field of the page and stores it by the term. Inverted Preindex is created for a set of pages (20000 in this case) for each of the 6 fields. That is, the list of pages and the term frequency on each page are returned for all the terms that appear in these set of pages.

For the term 102942211108157, page ids are stored along with term frequency

```
102942211108157 045XD 1 045XK 1 045XR 1
```

Idea is to create multiple such files and then use K-way merge to get to the final index. Storing the whole corpus in memory is out of question and hence writing to a PreIndex is extremely important. More than 1400 files for each of the fields were created during this process. K-way merge of the files sorted by terms is an efficient way of combining these files.

While PreIndex files are written, another file containing the pageID, and page Title information is written as “PageStats”. The file size of PageStats.txt became very large and runtime search performance slowed down because of the time taken to load the offset file PageStats. Hence, this file was split by the pageID and first 2 characters were used to write to multiple files. Essentially, 82 pageStat files were written along with 82 offset files.

```
0A++8 20 3 10 87 8 0 Cape VerdeIndia relations
```

where 0A++8 is the pageId, Cape VerdeIndia relations is the page title. And 6 numerical fields in between contain the number of terms in each field of the page. This information will be used for document length normalization during ranking of search results.

(Page ID is 64-bit encoded and hence each file had 64*64*64 pages)

Splitting the PageStats file into smaller files reduced the offset file size and substantially reduced the file load time.

Merging Pre-Index (Inverted Index)

Merging of pre-index files for each of the fields requires opening more than 1400+ files and for each of the fields (say, for Body text 'b')-

1. reading the first line of each file (indexb1.txt .... indexb1425.txt)
2. Find the lowest term in the lines,
3. Read a line from all files which have the same term (lowest)
4. Merge the list from all the files
5. Add the term to the Vocabulary of the field
6. Add the unique number of documents in which the term appears for that field
7. Go to the next term and repeat the steps above

The above K-way merge process is highly memory efficient (only one line from each of the files is read into memory). To make the file writing process efficient, we collate 50000 terms together and write into an index file – "b0.txt", next 50000 goes in "b1.txt" and so on. In addition to creating an index file, an offset file is also created that enables binary search and random access for the index file. More is explained when search algorithm is discussed.

The merging process creates a field specific Vocabulary file. However, some terms appear in some fields and not in other fields. To make the process efficient for run time search performance, all the field level vocabulary files are merged into a mergedvocab. MergeVocab is also created using a K-way merge process similar to the one discussed above. To ensure that Vocabulary size remains manageable, Vocabulary files are split into smaller files basis the first character of the terms. In addition, offset files are created for each of the partitioned vocab files.

Performance Enhancements Deployed After Experimentation

1. Instead of file compression to reduce index size which slows down real-time search performance, Page IDs are encoded in base 64. For a large DB like Wikipedia, page ids were running into more than 10 characters. Even the mapping of pageTitle and PageId is stored with the 64-bit encoding. PageID is repeated multiple times in the Index and gave an almost 20% reduction in the total index size. And since there is no need to decode page ID back, there is no performance hit.

2. Offsets are stored for ease in binary search. They also reduce the file size to be loaded at the time of searching. Index files are also split across multiple files. This enables smaller offset files to be loaded in one shot in RAM.

3. Use pickle. It stores files in a binary serialized format which decreases the time for file I/0. The offset files have been stored in pkl format. There is an increase in file size, hence only offset files were converted into pkl files. Index files were accessed using offset information and loading the full file in memory was not done.

4. Split the vocabulary and pagestatistics (which contains the title and number of words in each field of the page) into smaller files using a fixed mapping. For example, vocabulary was split on the first character of the term. And pagestatistics on the first 2 characters of the page id. The substantial reduction of the offset files helped in load time and improved the run time performance of search. The number of times vocabulary is accessed depends on the number of unique words in the query. And the number of times pagestatistics is

used depends on the number of results presented. The overhead in loading offset files is far lower than the savings in load time for smaller offset files.

5. Store terms instead of term-id in the vocabulary. Words are only accessed once and stored once. Creating a mapping between the term id and terms does not really save space. Plus the performance slows because of the mapping between term and term id.