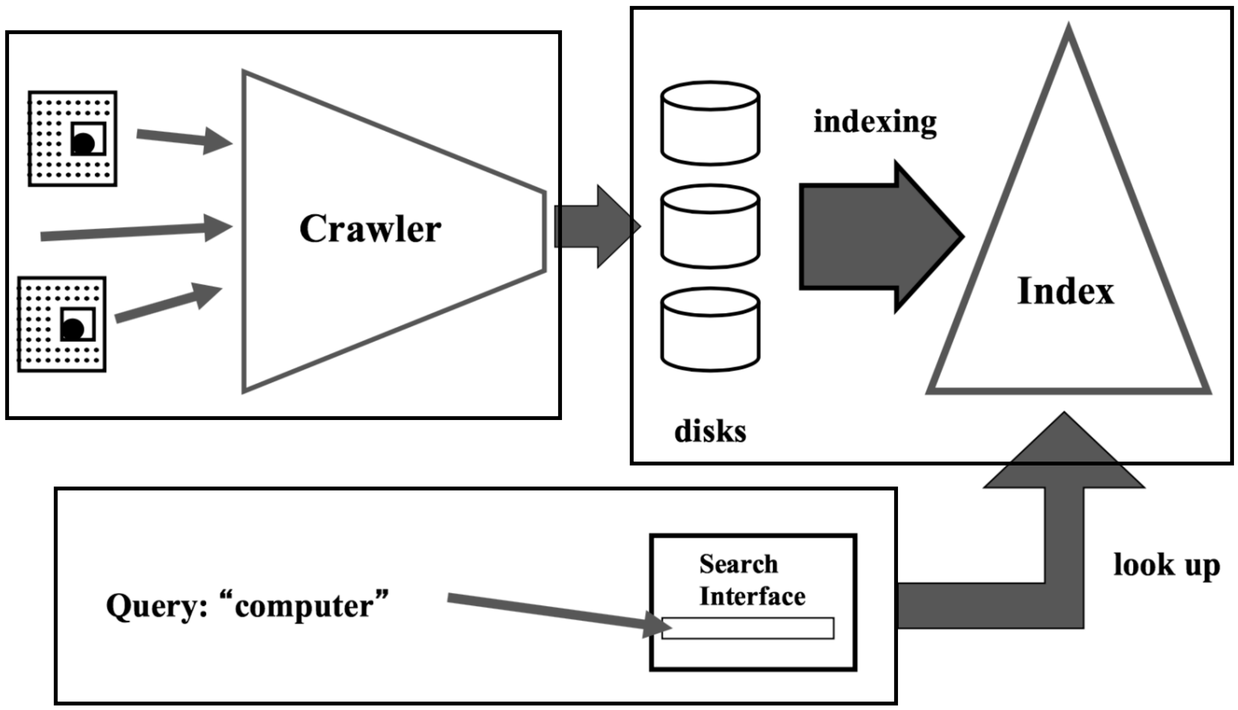
Project: Link analysis and ranking

1. Introduction

The project purpose is implementing link-based ranking techniques (both evaluation and query parts) such as PageRank and Hits, and combine with a term-based ranking function (BM25) on some large dataset. Then integrate these techniques with a search engine. This project focus on PageRank search engines. It will provide users more accurate search results.

2. Basic of Web Search Engine Architecture

There are three steps to establish a search engine: crawling, inverted index construction and query processing.



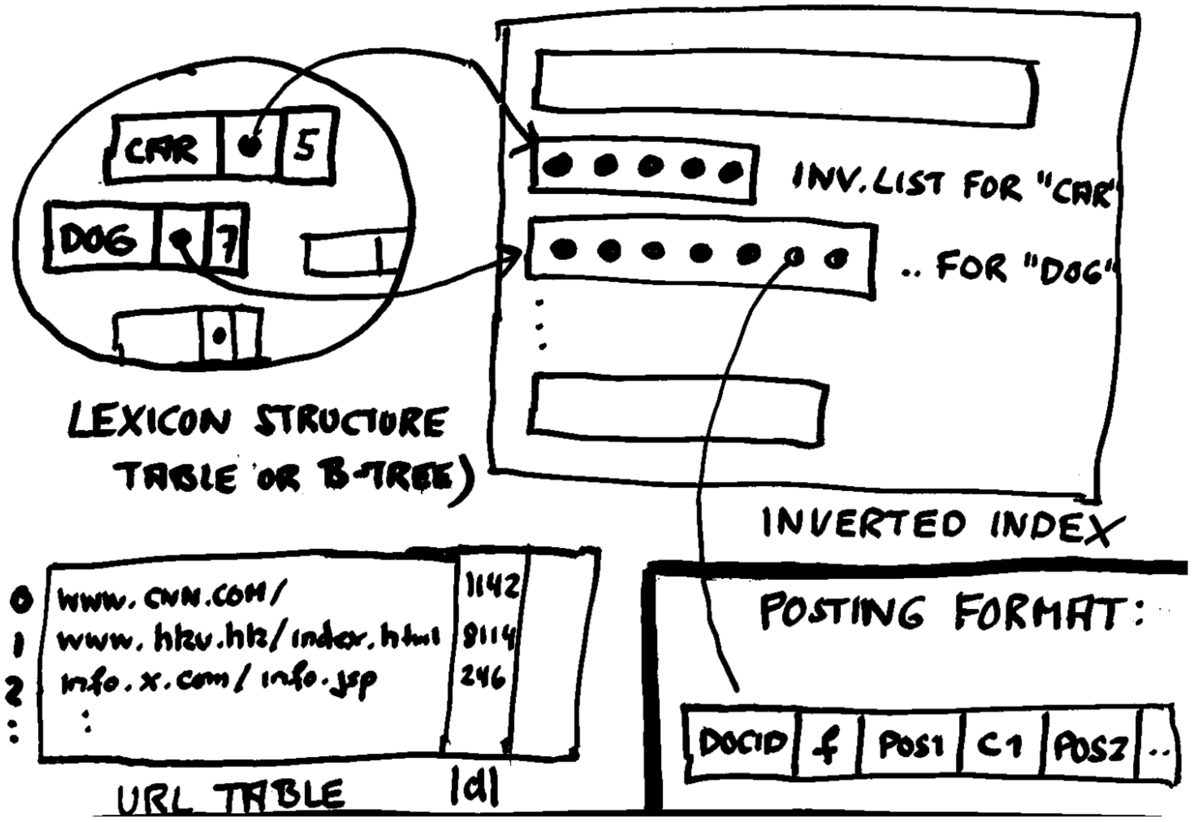
This project use GOV2 dataset for crawling step, so there is no need to crawl web pages from web.

2.1 Inverted index construction

Link analysis and ranking

Inverted index list is an important part of web search engine, which allows efficient retrieval of webpages that contain a particular query word. It will consist of lots of *inverted lists*, where every inverted list I_w contains the docIDs (document ID) of all documents of the collection that contain the particular query word, here I sorted them by docID, plus the term's occurrence of this document. For example, if a query is "dog and cat and mouse" which contains three query words, a search engine traverses the inverted lists of these three words and uses the information embedded therein, the number of search term occurrences (for computing BM25).

The diagram is showed below:



Steps of Generating Inverted Index and calculate PageRank:

1. Program to read the GOV2 data directly from file in sequence.

2. Use the provided parser and jsoup to parse each page to get (word, docID, frequency) tuples. At the same time, extract the hyperlinks from web pages and write the docID and its out links (urls) into file. After parsing a web page, write docID, url and its size into urlTable.

3. When the amount of (word, docID, frequency) tuples in memory is more than a

certain number, use quicksort sort (write a Comparable Method to use Arrays sort which provided by java.util.Arrays) to sort this array on docID and word. Then write the sorted array to a file. Here I will merge these 10,000,000 postings into partial inverted list and write the partial inverted list into file.

4. Repeat above steps until all the data is parsed. We will get a bunch of temporary posting files. The temporary files are written as ascii files, because I will use unix sort to merge them in the next step. The data in each temporary file is sorted. I don't use word ID so that I can generate the final inverted index in alphabet order.

5. Use unix sort to merge all the sorted temporary index files into a single file.

6. Parse the merged posting file to reformat it as the final inverted index format, and use the difference of docID and v-byte encoding to compress the inverted list file. Write the lexicon table at the same time.

7. Convert the extracted hyperlink (these links have not been parsed which means there is no inverted list for them) into docID for calculating PageRank scores. Each time convert 3 million links to docID and replace the out links with docID. Iterate this step until all out links have been replaced with docID.

8. Use outlinks file to get in links. Split the out link list into this format: docID (page) docID (citation) and write into inlinks file. Then sort the inlinks file and merge it to get inlinks list.

9. Use MapReduce technique to calculate the PageRank scores until it is convergence and append the scores into urlTable.

2.2 query Processing

Term-based Ranking: The most common way to compute ranking is based on comparing the particular words that are included in all document and in the query. More specifically, documents are modeled as a pool of words, and a ranking function (BM25) assigns a score to each document with respect to the current query words, based on the frequency of each query word in the web page and in the overall collection, the size of the document. Finally, given the query q = {t_0, t_1, … t_{d-1}} a *ranking function F* assigns to each document *D* a score *F (D, q)*. The search engine then returns the top k documents with the highest scores.

Here use BM25 to get scores, which is a popular class of ranking functions.

Link analysis and ranking

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

- N: total number of documents in the collection;
- f_t: number of documents that contain term t;
- $f_{d,t}$: frequency of term t in document d;
- $|d|$: length of document d;
- $|d|_{avg}$: the average length of documents in the collection;
- k_1 and b: constants, usually $k_1 = 1.2$ and $b = 0.75$

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

In addition, scores based on link analysis is often added into the total score of a document. According to [2], $F'(D,q) = w \times \frac{pr(D)^a}{k^a + pr(D)^a} + BM25(D,q)$ is a good choice, where $w = 1.8, k = 1, a = 0.6$.

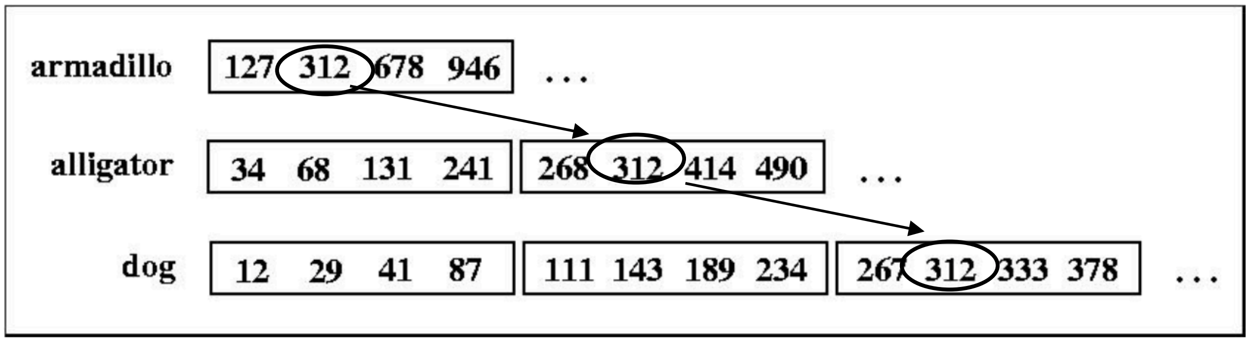
Caching: Caching will make the re-query well performance. Using the cache with LFU technique and store the visited times and compressed inverted list. Results for repeated queries may be cached directly at the query integrator, resulting in benefits from 20% to 70% [3].

Index Layout and Compression: Each inverted list is laid out sequentially on disk or in memory (cache) and compressed using one of many techniques that have been proposed, here I use v-byte to compress for quick performance.

DAAT Query Processing: When two or more inverted lists are of similar size, the best way to intersect them is to scan all these lists. When one list is much shorter than others, it is better to scan the shortest list and find the same docID over the longer lists, which means sorting the lists by their length. More specifically, we fetch one element (docID) from the shortest list, and then perform a forward seek in the next longer list to find the same docID, and if a same docID is found in the next list, and do the same thing on next list, and so on. This approach, taken by most current engines, is called *document-at-a-time* (DAAT) query processing, and it results in a simultaneous traversal of all lists involved in the query. Whenever an element (docID) in the intersection is encountered, we compute its score under the ranking function (BM25); the top-k results are found and they are maintained in a simple heap structure. Under this scheme, we may still extract the entire compressed inverted lists (just related with the query words) from disk unless the forward seeks result in very large skips ahead.

Link analysis and ranking

For conjunctive query, start travel from the shortest list, until find a docID occurs on all lists, calculate the BM25 score for it.



Step to Query (PageRank):

1. Upon startup, the query program loads the complete lexicon and URL table data structures from disk into main memory.

2. Create a cache for storing the inverted lists that correspond to query words, which is used for next query when keywords are searched again. Also there is a Hash Map for high frequency words, it is used to record the visited time of query words. When cache is overflow, remove the low frequency word from cache.

3. After input query words, get the inverted list pointer of each word from lexicon. Then open the inverted list file to get the list of the corresponding word and store into cache for next searching.

4. The program is implemented by DAAT query execution technique. If query words contain "or", then compute bm25 score of pages of all keywords, and return 10 results with the highest scores. If query words contain "and" or just use blank separate query words, in this case, program will return the results that contain all query words. Start with the shortest inverted list, then check all the docID of the shortest inverted list is in other lists or not. If all inverted lists contain this docID, compute its **bm25 scores** plus **PageRank scores** and then store into the heap for ranking. Else skip this docID.

5. Finally, this program shows these 10 urls with their scores on the frame. They are web accessible.

Step to Query (Hits):

Link analysis and ranking

1. Upon startup, the query program loads the complete lexicon, outlinks, inlinks and URL table data structures from disk into main memory.

2. Create a cache for storing the inverted lists that correspond to query words, which is used for next query when keywords are searched again. Also there is a Hash Map for high frequency words, it is used to record the visited time of query words. When cache is overflow, remove the low frequency word from cache.

3. After input query words, get the inverted list pointer of each word from lexicon. Then open the inverted list file to get the list of the corresponding word and store into cache for next searching.

4. The program is implemented by DAAT query execution technique. If query words contain “or”, then compute bm25 score of pages of all keywords, and return 10 results with the highest scores. If query words contain “and” or just use blank separate query words, in this case, program will return the results that contain all query words. Start with the shortest inverted list, then check all the docID of the shortest inverted list is in other lists or not. If all inverted lists contain this docID, compute its bm25 scores and then store into the heap for ranking. Else skip this docID.

5. Use top 200 results as root set, then get hyperlinks from outlinks and get citation from inlinks to construct a base set.

6. Use base set to get matrix and transpose of matrix, the initial value is 1. Then initialize Hub Score array and Authority Score array with 1/n. Set the max iteration is 100 unless it is convergence, and each iteration will normalize the hub score. Finally return top 10 urls with highest hits scores as results.

3. Compile and Running

3.1 How to Compile:

This project is developed with Eclipse. Please use Eclipse to open this project to compile and run.

3.2 To get index list, lexicon, urlTable and outlinks (initial release):

index.java // Run this program to get temporary posting file, url Table and outlinks.
unixsort.java // Then run this program to sort the posting file and finally get a

single posting file.

lexicon.java // This program is convert the posting files into the lexicon structure and inverted list file.

3.3 To get PageRank, final outlinks file and inlinks file:

linkToDocid.java // Run this to convert the urls to docID, each time convert 3 million urls.

check_noparse.java // Convert the urls (the result from above program) into docID

inlink.java // Convert the final outlinks into inlinks

pagerank2.java pagerankMapper.java pagerankReducer.java // Use MapReduce to calculate PageRank, each time will iterate 3 times. Input file is outlinks file.

Convergence.java //Because I use MapReduce technique to compute PageRank, it is difficult to check convergence during the process, I run this program to check convergence after finishing pagerank program.

3.4 Query - PageRank:

pagerankquery.java // Run this program and input your query, it will show 10 relevant results on the frame, click them and access the website.

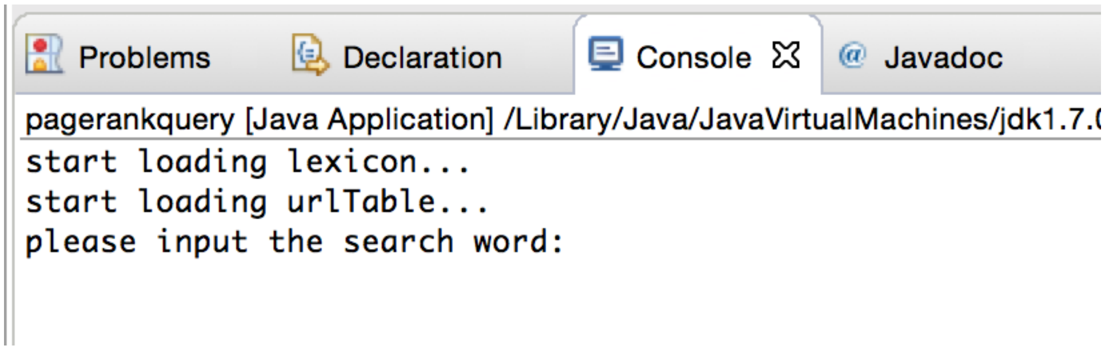
3.5 Query - Hits:

queryhit.java // Run this program and input your query, it will show 10 relevant results on the frame, click them and access the website.

3.6 Instruction

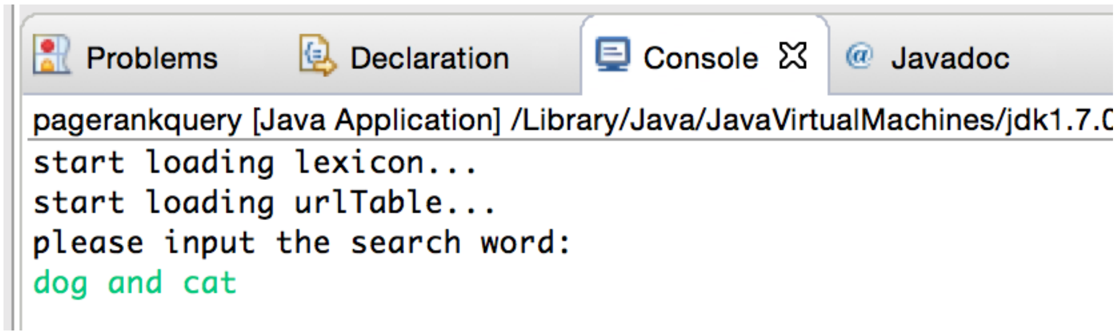
- Run the pagerankquery.java (queryhit.java) in Eclipse, waiting for loading the lexicon and urlTable (if use Hits technique, there are inlinks file and outlinks file), until it shows “please input the search word”.

Link analysis and ranking



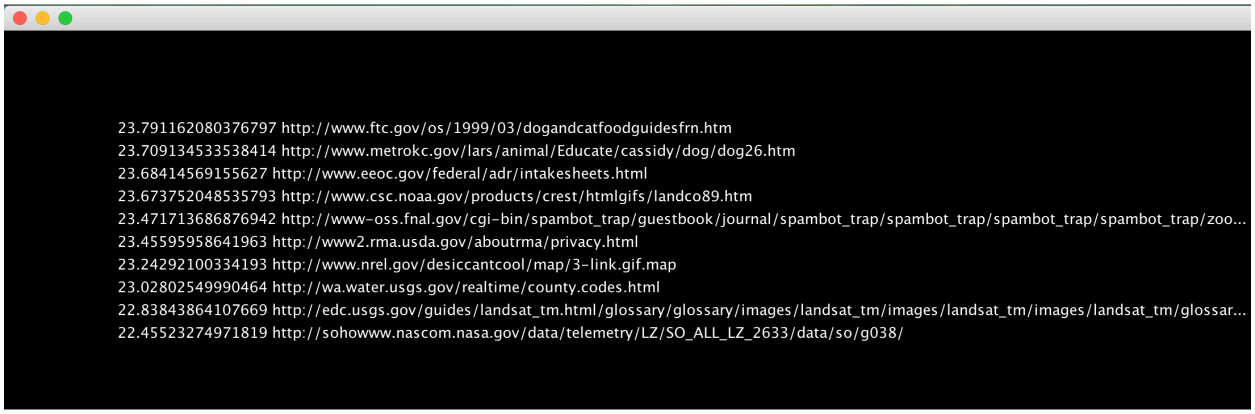
```
pagerankquery [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.
start loading lexicon...
start loading urlTable...
please input the search word:
```

- Input the word you want to search and press enter (conjunction).



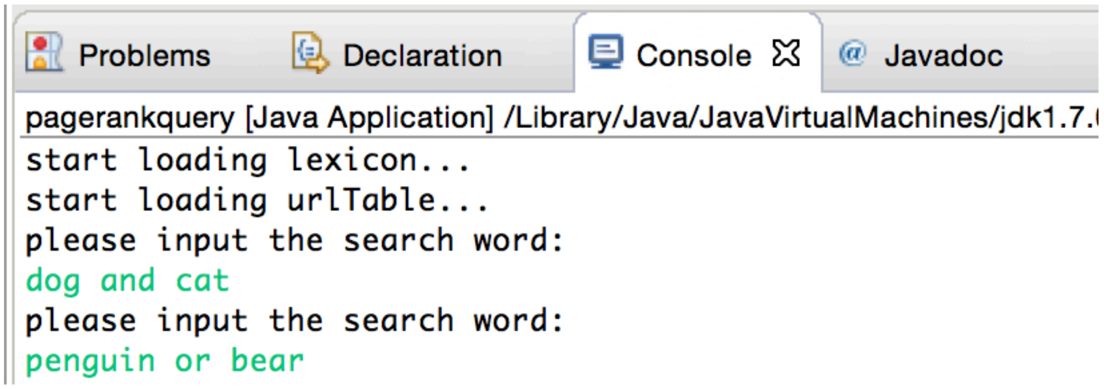
```
pagerankquery [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.
start loading lexicon...
start loading urlTable...
please input the search word:
dog and cat
```

- Then there will be a pop-up window which display the top 10 urls with their scores



```
23.791162080376797 http://www.ftc.gov/os/1999/03/dogandcatfoodguidesfrn.htm
23.709134533538414 http://www.metrokc.gov/lars/animal/Educate/cassidy/dog/dog26.htm
23.68414569155627 http://www.eeoc.gov/federal/adr/intakesheets.html
23.673752048535793 http://www.csc.noaa.gov/products/crest/htmlgifs/landco89.htm
23.471713686876942 http://www-oss.fnal.gov/cgi-bin/spambot_trap/guestbook/journal/spambot_trap/spambot_trap/spambot_trap/spambot_trap/zoo...
23.45595958641963 http://www2.rma.usda.gov/aboutrma/privacy.html
23.24292100334193 http://www.nrel.gov/desiccantcool/map/3-link.gif.map
23.02802549990464 http://wa.water.usgs.gov/realtime/county.codes.html
22.83843864107669 http://edc.usgs.gov/guides/landsat_tm.html/glossary/glossary/images/landsat_tm/images/landsat_tm/images/landsat_tm/glossar...
22.45523274971819 http://sohowww.nascom.nasa.gov/data/telemetry/LZ/SO_ALL_LZ_2633/data/so/g038/
```

- Click the url and go to the webpage
- If you want to search more words, just go to Eclipse console, type your query again. (disjunction)



```
pagerankquery [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.
start loading lexicon...
start loading urlTable...
please input the search word:
dog and cat
please input the search word:
penguin or bear
```

- To finish searching, close the window and stop the program.

4. Reference

[1] Y. Chen, T. Soul, A. Markowetz. Efficient query processing in geographic web search engines. *SIGMOD '06 Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 277-288, June 2006
[2] N. Craswell, S. Robertson, H. Zaragoza, M. Taylor. Relevance Weighting for Query Independent Evidence, *SIGIR '05 Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 416-423, August 2005
[3] E. Markatos. On caching search engine query results. In *5th Int. Web Caching and Content Delivery Workshop*, May 2000.
[4] Y. Chen, Q. Gan, T. Soul, I/O-efficient techniques for computing pagerank. *CIKM '02 Proceedings of the eleventh international conference on Information and knowledge management*, pages 04-09, November 2002.

Accessory:

URL Table Output Format:
docID, URL, size, PageRank scores
- docID is the number of the page of this url.
- size is the length of this page

Lexicon Table Output Format:
Word, Number, inverted list Pointer
- Word is the word.
- Number is the total number of docs that containing this word.

Link analysis and ranking

- inverted list Pointer points to the inverted index list.

Inverted Index Output Format:
docID, frequency, docID, frequency...
- docID is compressed.
- frequency is the occurrence of the word in this doc.

Inlinks Output Format:
docID, docID, docID, docID...
- The first docID is the webpage.
- Other docIDs are links that cite this webpage.

Outlinks Output Format:
docID, docID, docID, docID...
- The first docID is the webpage.
- Other docIDs are hyperlinks of this page.

Compress:
- Use difference of the docID for the same word and v-byte decoding to compress the docID

Result and Performance:
Here I use 100 GOV2 data set for test.
url Table: 64 MB (only letters)
Size of inverted index: 1.1 GB
Lexicon: there are 2780891 distinct words

Search common words will less than 1s.
Search stop words will cost 2-4s.