Polytechnic Institute of NYU

Computer Science and Engineering

CS 6913, Spring 2013

Jiankai Dang, Poly #: 0482923

Zhuoran Yu, Poly #: 0489525

Assignment #3 Web Search Engine Report

1. How to Implement a Search Engine: Crawl Web Pages

Building Web Crawler is assignment #1. It is somehow relative to this assignment. So in Part 1, we will briefly talk about crawler, and then talk about how it connects to this assignment.

a) Crawler

Crawler, also named spider, is a program to collect data on the web. It tries to download all files it could research on the web site and store.

b) Data

Crawler downloads pages. Save a number of pages in one file. Compress these files and store them. The data given to us for assignment 3 is not the data set we crawl in Assignment #1. Due to this reason there are some features we want to do but cannot in this assignment.

c) More about Data: Duplicate Contents

In assignment #3, we add a feature that program check duplicate contents in result set. If the load is huge, it is not an efficient way to do duplicate detecting in query processing part. Instead we should detect duplicate contents when store files. If so, the duplicate detecting in query processing part would be more efficient.

d) More about Data: Update

In practice, a search engine should not be “static”. The crawler is always downloading web pages, adding them into

storage. At the same time, we should always be updating our index and show latest result to customers. In this assignment, we could not do this since the data set is static.

2. How to Implement a Search Engine: Build Index

An inverted index, also named posting files or inverted files, is an index data structure storing mapping from each terms to content include it. For example, words (or word id) to its location in a set of documents or in a database files. The purpose of inverted index is to fast full text searches. [1]

For example, Document 1 is ["Inverted Index"], Document 2 is ["Inverted Index is a kind of Index"]. The inverted index of word "Index" is [[1, 1], [2, 2]], meaning that this word is in document 1 for 1 time and in document 2 in 2 times.

- **Build Index**

This part is within assignment 2, we would introduce here very briefly. Suppose the memory is large enough, we could scan all files and add document id into inverted index of terms within the document. Unluckily, in practice memory is probability not enough. We would tell what we do in "efficient building". We will introduce some useful or highlights details of our project

- Parsing the Collection

When building index, we should make a decision whether build index for non-natural word, such as "0fxxx". If not, we could use Natural Language Toolkit [2] to filter. This would make inverted index in a small size. The speed would be highly increased. If so, our user could get result of special search queries. For example, someone want to search a special line of code "string.rfind(x)". However, the inverted index would be much larger. In this project, we choose a third way. We do build inverted index for every term. For efficient, we build up a cache for common words

in natural language. This would be told later in details.

- Efficient Building

There are mainly four approaches to speed up index building: linked list, I/O efficient sort, merging indexes and lexicon partition. We use a mixed solution. Our solution is to divide files into several parts. We build up inverted index separately for each part. Finally, our program uses an I/O efficient algorithm to merge these inverted indexes (in most of time, the algorithm is merge sort). The disadvantage is speed is a little slower, for we build multiple semi-lexicon files. The advantage is scaling. Our program allow adding in new set of inverted index, we don't care where it is from. Inverted files created by one crawler are available to merge into inverted files created by other crawlers (or maybe prior version of itself).

- Index Compression

The goal is to compress inverted index. More, we don't care the size of inverted index in hard drive, for the hard-drive is cheap. What we care is the size of cached inverted index. If the size of one inverted index is small, we could cache more. We also care about the speed of decompress.

There are many possible methods, e.g., VByte, Chunk-wise. Variable byte (VB) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "play load" and encode part of the gap. The first bit of the byte is a continuation bit. It is set to 1 for last byte of the encoded gap and to 0 otherwise.

$$l = \lfloor \log_{128} n \rfloor$$

Where l is the number of bytes we would use and n is the number we want to compress. [2]

3. How to Implement a Search Engine: Query Processing

We assume that we have a document collection $D = \{d_0, d_1, \ldots, d_{n-1}\}$ of n web pages that have already been crawled and are available on disk. Let $W = \{w_0, w_1, \ldots, w_{m-1}\}$ be all the different words that occur anywhere in the collection.

Typically, almost any text string that appears between separating symbols such as spaces, commas, etc., is treated as a valid word (or term) for indexing purposes in current engines.

- **Query Types:**

This project will answer two types of queries:

One word queries: Queries that only contain one word, like soccer, or music.

Free text queries: Queries that contain several words that are separated by space. This project uses AND query strategy, which returns a document if and only if all the terms appear in this document.

- **Query Parsing:**

We notice that input query would not follow any rules. We could not treat it as a natural language format. For example, "Idon'tLikeyou" would exist as a query. Also, we should also expend some short-format word, like "it's" should be "it" and "is". Therefore, we use a NLP parser to parse every query. We also delete non-meaningful words like '*#$'.

- **Query Execution:**

Given an inverted index, a query is executed by computing the scores of all documents in the intersection of the inverted lists for the query terms. This is most efficiently done in a document-at-a-time approach where we scan the inverted lists, and compute the scores of any document that is encountered in all lists. (This approach is more efficient than the term-at-a-time approach where we process the inverted lists one after the other.) Thus, scores are computed, and top-k scores are maintained in a heap structure. In the case of AND semantics, the cost of performing the arithmetic operations for computing scores is dominated by the cost of traversing the lists to find the documents in the intersection, since this intersection is usually much smaller than the complete lists.

- **Efficient Cache:**

Our cache technique is based on natural language words frequency. We load an open **English words frequency** file ("EnglishWordFrequency2.txt") [8]. Depending on the word frequency, we cache inverted index of selected words with descending frequency.

- **Ranking function:**

So far, our program could present the result set based on your query. However, there is still a big problem. We could not show hundreds of result pages to users. We should rank them and show top 10 or top 20 of them. Here we would discuss the ranking function we use. We combine three kinds of ranking function together: Okapi BM25, PageRank and AlexaRank.

Okapi BM25 is a ranking function used by search engines to rank matching documents according to their relevance to a given search query. It is based on the probabilistic retrieval framework developed in 19702 and 1980s by Stephen E.Robertson, Karen Sparck Jones and others [3]. The ranking function is:

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

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

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 $b = 0.75$.

PageRank is a link analysis algorithm, named after Larry Page[4] and used by the Google web search engine, that assigns a numerical weighting to each element of a

hyperlinked set of documents, such as the World Wide Web, with the purpose of "measuring" its relative importance within the set. The algorithm may be applied to any collection of entities with reciprocal quotations and references. The numerical weight that it assigns to any given element E is referred to as the PageRank of E and denoted by $PR(E)$.[5]

Alexa ranks sites based primarily on tracking information of users of its toolbar for the Internet Explorer, Firefox and Google Chrome web browsers. Therefore, the webpages viewed are only ranked amongst users who have these sidebars installed, and may be biased if a specific audience subgroup is reluctant to do this. Also, the ranking is based on three-month data,[16] and thus takes a long time to reflect changes in content that may happen after a domain has been sold or undergone a major redesign. Furthermore, low rankings cannot be accurate, not merely due to the paucity of data but also because of statistical laws related to the long tail distribution.[6]

How to combine these three results is another problem. In practice, we use

$$Score = BM25 * \left((PR - k_4) * k_2 + k_4\right) * (1 + \frac{k_3}{AlexRank})$$

where k_2, k_4 and k_3 are constant variables.

In our program, we implement BM25. Use AlexaRank API to get AlexaRank number. We wrote a crawler to crawl Google's page rank number.

- Result Set: One more Check

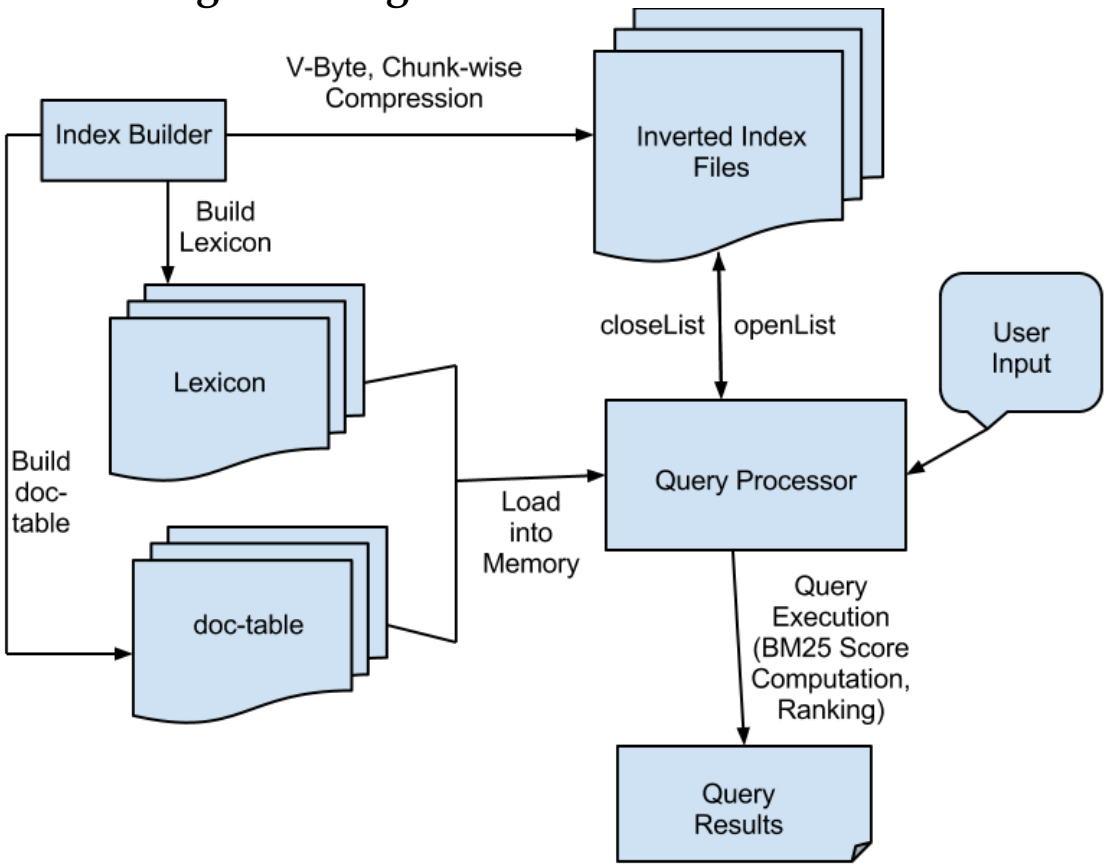
In practice, there is always a delay between time spider download the page and the time user look at the result page. In this project, we found there are lots of pages not visit-able at this time. So we check the visit-ability before show page contents to user.

- Result Set: Show a Glance of Content

If one result page is too large, a novel for example, user always wants to look a glance of it rather than open it. In our

program, we would show a 140 characters sub-content (same length as a tweet) to user. We try to find the best sub-content in a short time.

4. About Programs: High-level Structure



5. About Programs: Files in details

a) Processor.py

Query Processor: index information loading, BM25 Score Computation, Ranking

b) IndexCompression/IndexCompression.py

Compress Inverted Index Files, using V-Byte and Chunk-wise Compression.

c) queryParser.py

query parser module

d) makeCache.py

cache module

e) checkResult.py

Check the result, if visit-able. Also give best sample content.

f) getPageRank.py

get PageRank and AlexaRank number by their APIs

g) EnglishWordFrequency.py

The English word frequency file

h) simHash.py

The similar hash function, the same as Assignment #1.

6. Search In Action: How to Run it

- Our program needs additional python packages, NLTK. To install this package, see instruction here. [7]
- In the folder of our program, type in the command line "python Processor.py". The program would ask you to type in some commands.
- "quit" command: quit the program
- "search" command: type "search" and follow the instruction to type into your query (any format). Then the program would show you the result.
- "search-complex" command: the same as "search". This time, program would show you a full result, with 140 chars sample content.

7. Search In Action: Experiment

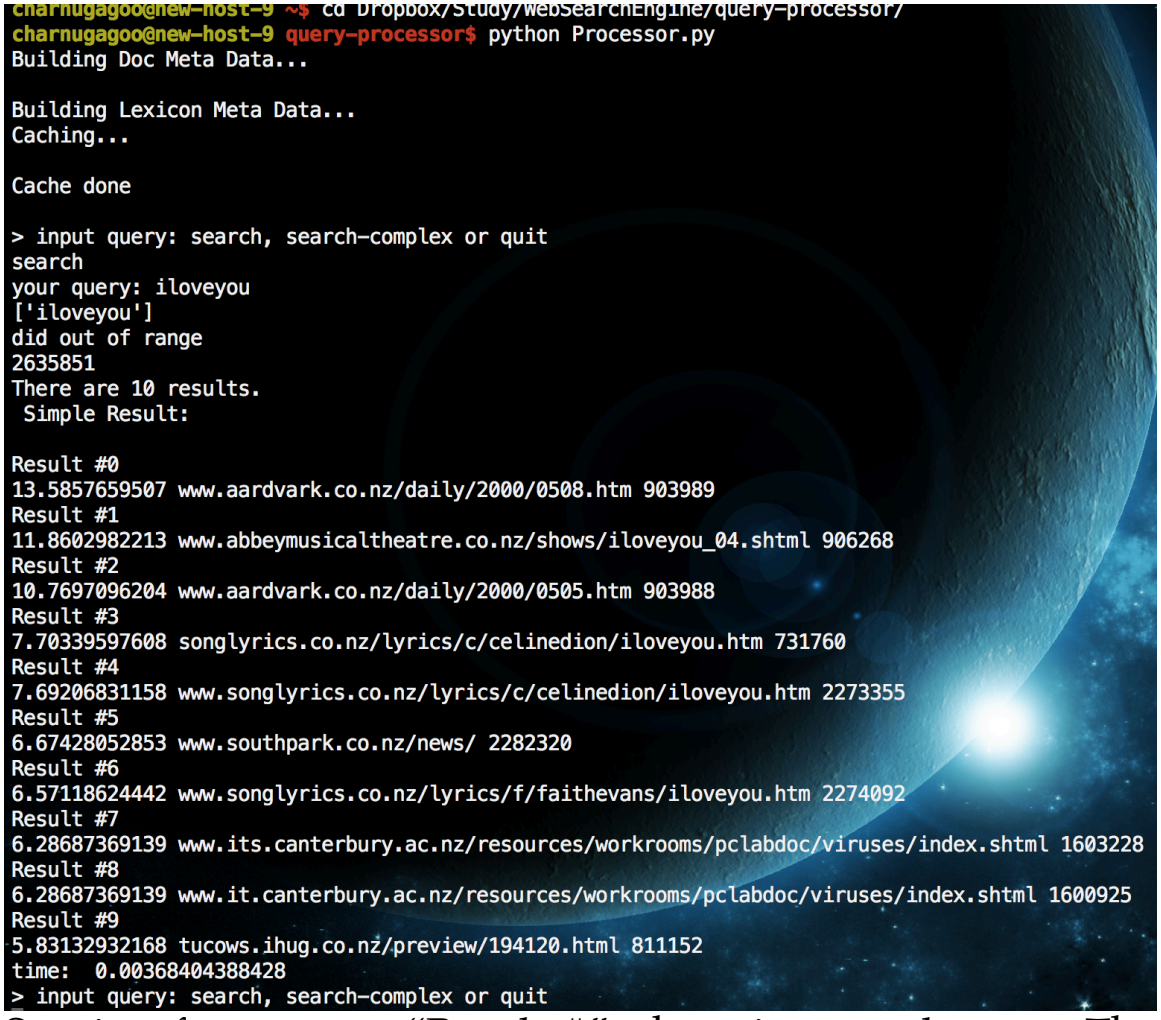
a) How long does it take on provided data set:

The time cost really depends on the query. For some special query, like very common words "for, the", it would be as slow as several seconds (3s at most for the largest cases "for" and "the"), although we have already cache them in memory (cache decoded data for these words). For regular cases like "Beijing, New York City, Poly", the speed would be as fast as micro seconds level. For example, the "iloveyou" query showed in **b** uses 0.003 s.

b) Some Sample Result:

Sample result page:

1. Sample simple result of "iloveyou":



Starting from every "Result #", there is a result page. The first is the score, second is url, third is the did.

2. Sample complex result of "iloveyour":

We would first show you the same as simple result.

After several moments, we would show you a complex result, including whether two pages have similar contents. Also show you a sample 140 characters string of every result page.

8. Search In Action: Limitation

This program has implemented all the requirements of the 2nd and 3rd assignments. Also, during the index construction, the disk-based structures are all in binary formats. However, there are still some limitations:

This simple search engine does not support Phrase Queries. Phrase Query means that the input is a sequence of words, and the matching documents are the ones that contain all query terms in the specified order.

Besides, we found most pages given in the data set could not be arrived. Most of them return 404 errors. I cannot get latest page content. Also, it delay the time to get latest pages. So the complex-result is not good enough.

Reference:

[1] http://en.wikipedia.org/wiki/interted_index

[2] http://nlp.stanford.edu/IR-book/html/htmledition/variable-byte-codes-1.html

[3] http://en.wikipedia.org/wiki/Okapi_BM25

[4] "Google Press Center: Fun Facts". www.google.com. Archived from the original on 2009-04-24.

[5] http://en.wikipedia.org/wiki/PageRank#cite_note-1

[6] http://en.wikipedia.org/wiki/Alexa_Internet

[7] http://nltk.org/install.html

[8] http://www.datatang.com/data/10131#

[9] Long/Suel: Three-Level Caching for Efficient Query Processing in Large Web Search Engines. 14th International World Wide Web Conference, 2005.