Polytechnic Institute of NYU
Computer Science and Engineering
CS 6913, Spring 2012
Xiang Liu, Poly #: 0441218

Assignment #3 Web Search Engine Report

1. How to Implement a Search Engine: Create Index

An inverted index (also referred to as postings file or inverted file) is an index data structure storing a mapping from content, such as words or numbers, to its locations in a database file, or in a document or a set of documents. The purpose of an inverted index is to allow fast full text searches, at a cost of increased processing when a document is added to the database [1].

For example, if we have the following documents: Document 1: Information Retrieval and Web Search; Document 2: Search Engine Ranking; Document 3: Web Search Course. Then the inverted index of term 'web' would be [[1, 3], [3, 0]], meaning that 'web' appears in document 1 with position 3 and document 3 with position 0.

- **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.

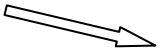
- **Parsing the Collection:**

To build the index file, we first need to decide which words should be indexed by parsing the document collection. In this project, we used Natural Language Toolkit [2] to parse the NZ dataset in two steps: convert an HTML string to raw text; and tokenize and lowercase the words. The first component generates intermediate postings from the files and writing these postings.

- **Building the Inverted Index**

Given a collection of documents, how to efficiently create an inverted index? There are mainly four approaches: linked list, merge sort, merging indexes and lexicon partitioning. In this project, we used merge-sort to build the inverted index. The general idea of merge-sort is to (1) go through documents and create (word, document #, position) tuples, in other words, intermediate postings, (2) sort tuples by (word, document #, position) and (3) merge sorted postings (4) compress index. When generating the tuples, we also need to filter out the tokens in the stop words list, such as 'a', 'the', et al., which do not have actual meanings. Here is an example of index creation:

Document 1: Information Retrieval and Web Search:
['information', 1, 0], ['retrieval', 1, 1], ['web', 1, 3], ['search', 1, 4]
Document 2: Search Engine Ranking:
['search', 1, 0], ['engine', 1, 1], ['ranking', 1,2]
Document 3: Web Search Course:
['web', 3,0], ['search', 3, 1], ['course', 3, 2]



course [[3,2]]
engine [[2,1]]
information [[1,0]]
ranking [[2,2]]
retrieval [[1,1]]
search [[1,4],[2,0],[3,1]]
web [[1,3],[3,0]]

Finally, there are three structures stored on disk: inverted index stored in one or a few files that only contain the document # and the term frequency in that document; lexicon contains word, offsets and length into inverted index; and url table.

2. How to Implement a Search Engine: Query Index

- **Query Execution**

The disk-based inverted index is the main data structure of our search engine. We used two Hashtables (python's dictionary) to store the lexicon and document information in memory. However, the inverted index itself cannot be loaded into memory. After the user inputs a query, the program performs seeks on disk in order to read only those inverted lists from disk that correspond to query words, and then compute the result.

Let's first analyze two query execution techniques, DAAT and TAAT. Document-at-a-time (DAAT) strategies fully evaluate the document score by considering the contributions of all query terms with respect to the document before moving to the next document. While Term-at-a-time (TAAT) strategies are by far most common in traditional IR systems due to the simplicity of their implementation, there are some clear advantages to (DAAT) strategies, especially when dealing with very large data sets. First, it is usually not feasible to maintain partial results for all candidate documents in main memory. Second, I/O operations required for posting retrieval can be easily parallelized when using DAAT strategies, since all posting lists are traversed in parallel. Finally, advanced search features such as Boolean operators, proximity operators, and numeric range constraints, can be handled more efficiently by DAAT strategies, since all conditions can be evaluated at the same time to decide whether a document "satisfies" the query [3]. Thus we prefer DAAT strategies in this project.

Next, let's look at the query process. Assume we use AND semantics: "all query words must occur in result". In DAAT strategies, all inverted lists of all the terms in the query are traversed simultaneously from left to right. 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.

- **Inverted Index Compression**

In inverted index compression, the goal is to compress a sequence of integers, either a sequence of d-gaps obtained by taking the difference between each docID and the previous docID, or a sequence of frequency values. The integers to be compressed are non-negative but do include 0 values [4]. Thus we should use efficient coding for docIDs, frequencies, and positions in

index by first taking differences, and then encoding those smaller numbers.

Examples:

- if postings only contain docID:
  (34) (68) (131) (241) … -> (34) (34) (43) (110) …
- if postings with docID and frequency:
  (34,1) (68,3) (131,1) (241,2) … -> (34,1) (34,3) (43,1) (110,2) …
- if postings with docID, frequency, and positions:
  (34,1,29) (68,3,9,46,98) (131,1,46) (241,2,45,131) … -> (34,1,29) (34,3,9,37,52) (43,1,46) (110,2,45,86) …

Afterwards, do encoding with one of many possible methods, e.g., simple method: vbyte; Chunk-wise Compression. Variable byte (VB) encoding uses an integral number of bytes to encode a gap. The last 7 bits of a byte are "payload" and encode part of the gap. The first bit of the byte is a continuation bit. It is set to 1 for the last byte of the encoded gap and to 0 otherwise.

Encode number as follows:

- if < 128, use one byte (highest bit set to 0)
- if < 128*128 = 16384, use two bytes (first has highest bit 1, the other 0)
- if < 128^3, then use three bytes, and so on …

Examples: 14169 = 110*128 + 89 = 11101110 01011001
33549 = 2*128*128 + 6*128 + 13 = 10000010 10000110 00001101

- **Ranking**

So far we basically have a search engine that can answer search queries on a given corpus, but the results are not ranked. Now, we will discuss ranking algorithms to obtain an ordered list of results, which is one of the most challenging and interesting parts. The ranking scheme we implemented in this project is Okapi BM25. In information retrieval, 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 the 1970s and 1980s by Stephen E. Robertson, Karen Spärck Jones, and others [5].

The ranking function:

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

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

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 $b = 0.75$

3. WHAT the PROGRAM DOES?

The program code is entirely written in Python and has three main parts:

- Parser: It parses the pages of NZ10 dataset and returns the intermediate postings. We implemented it by first uncompressing the files and reading a certain length, which is specified in each line of the index file, of data in *_data files. Finally, the intermediate postings in the format of [term, term position, urlID] tuples are returned. We used python nltk (Natural Language Toolkit) to clean the page data and tokenize it.

```
raw = nltk.clean_html(chunk1).lower()
tokens = nltk.word_tokenize(raw)
```

- Inverted index construction: After tokenizing the file and giving each URL a unique ID, we will create an inverted index. This section will create the complete inverted index containing three files. First file will have all the unique words, how many documents contain each word and the start position and length in another file that corresponds to the specific word's inverted index; the second file, which has been encoded and compressed, is the inverted lists of all the words; the third file has the URL ID, URL and the length of page of each URL. For each intermediate posting, we first eliminate the items that are in stopwords list, then sort it by < word, docId, position >. Then use python heapqlib to merge the sorted files:

```
for line in heapq.merge(*[decorated_file(f, keyfunc) for f in files])
```

At the end of the program, all structures are stored on disk in **binary** data format:

```
f = gzip.open(output_path,'wb')
```

- Query execution and results ranking: In this part we are able to get the query from user and search all possible documents where all the words occur (because we use AND search strategy). After that we implemented BM25 algorithm to rank the results and the top 10 URLs with corresponding scores are returned:

Upon startup, the query processor reads the lexicon and URL table data structures from disk into main memory. However, the index itself is not loaded into memory. After the user inputs a query, the program then performs seeks on disk in order to read only those inverted lists from disk that correspond to query words, and then compute the result. In this program, we used efficient coding for docIDs in index by first taking differences, and then including Variable-Byte index compression. To find out the first ten ranking result with highest scores, we used Python heapq.nlargest method to implement:

```
for nrank in heapq.nlargest(10, res, key = lambda x: self.bm25_relevance(words,x))
```

4. HOW TO RUN THE PROGRAM:

First uncompress the NZ10 tar file.

Second, run **python parser.py**. In default, the NZ10 data and index files are in the folder "datset" in the Python directory. The intermediate postings are generated into output directory (new folder will be created if no such directory exists).

Third, run **python mergesort.py**. Upon startup, enter the folder containing the intermediate posting files. The inverted index, lexicon and URL table will be generated.

Finally, **run python query_new.py**. Upon startup, the query processor takes a little longer, but each query then is answered fairly quickly. After the user inputs a query, the program performs seeks on disk in order to read only those inverted lists from disk that correspond to query words, and then compute the result. After returning the result, the program waits for the next query to be input. As figure 1 show, by entering the query, the top 10 ranking URLs are returned with the BM25 score and also the query time: 1 second.

```
Enter the Query: penguin
2012-04-06 00:27:43.832000
['penguin']
www.penguin.net.nz/news/archive/0509.html 10.8572368585
www.conservation.govt.nz/Conservation/001~Plants-and-Animals/004~Seabirds/009~Pe
nguins-in-Canterbury.asp 10.7412550288
www.yellow-eyedpenguin.org.nz/about/index.html 10.6303762177
www.penguin.co.nz/static/html/nz/permission/index.html 10.4625780442
www.penguin.net.nz/cons/conservation.html 10.3879878764
www.penguin.net.nz/faq/faq.html 10.3751044871
www.yellow-eyedpenguin.org.nz/news/news.html 10.328735559
www.penguin.net.nz/viewing/wild.html 10.3098566573
www.yellow-eyedpenguin.org.nz/index.html 10.0716016623
neon.otago.ac.nz/chemistry/research/sab/dunedin/dun22.htm 10.0660321923
2012-04-06 00:27:44.867000
```

Figure 1. Query results for “penguin”

5. HOW THE PROGRAM WORKS INTERNALLY:

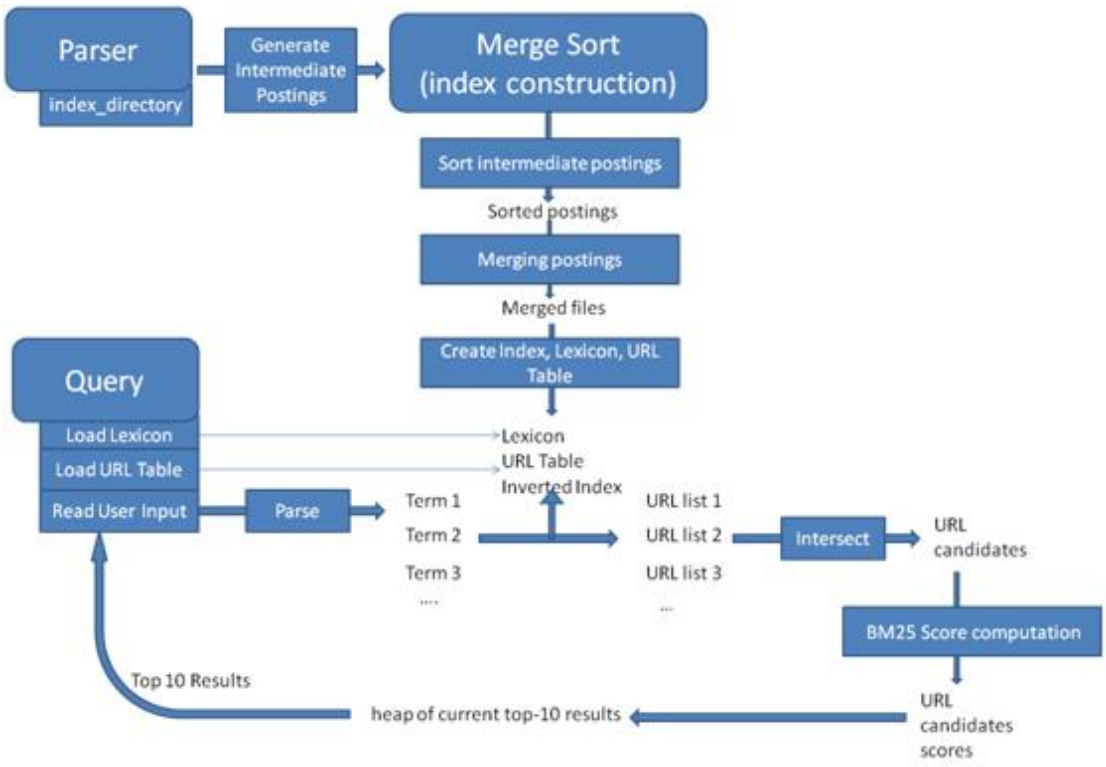


Figure 2: how program works internally

As illustrated in Figure 2, in Parser module, index_directory function is called to generate intermediate postings. In mergesort module, three main parts are called to build the inverted index: sort intermediate postings, merge posting files, and create index structures and write into disks. Finally, in query module, lexicon and URL table files are loaded into memory upon program starts. Then user's query is parsed into terms, geturllist function is called for each term in terms and the URL list for each term is returned. With Python's Set intersection operation, we have URL candidates, each of which is computed with a BM25 score. Finally, the top 10 ranked results in the heap are returned and displayed.

6. LIMITATIONS OF THE PROGRAM:

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

1. 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.
2. This project does not support Chunkwise encoding. In real systems, compression is done in chunks. And the main factor that affects the query time is the size of index file. Because when we read the inverted index of a term, we need to perform seek(offset) first. We will show how the size of index file affects the query execution time in section 8.
3. As suggested in the lecture slides, all inverted lists in the query should be traversed simultaneously from left to right. This can be implemented using four simple primitives:

   - openList(t) and closeList(lp): open and close the inverted list for term t for reading.
   - nextGEQ(lp,k): find the next posting in list lp with docID >= k and return its docID.
   - getFreq(lp): get the frequency of the current posting in list lp

   However, in this project, we used Python Set intersection operation to retrieve the matching documents.

7. MAJOR FUNCTIONS AND MODULES USED IN THE PROGRAM:

parser.py: The first module that parses the NZ10 data.

- index_directory(directory): parses out the pages provided in NZ dataset with the format of *data and *index files and returns [term, term position, urlID] tuples.

Mergesort.py: The second module that creates the inverted index.

- sort(w,hashlist): For each intermediate posting file, the items sorted by <word docId postion> returned
- decode7bit(bytes): integer decoded function
- encode7bit(value): integer encoded function, which is similar to variable byte encoding
- createIndex(input_path, output_path, lexicon): To create inverted index: build index structure dict(term) = [[doc_1, frequency], [doc_2,frequency], [doc_3, frequency]]. In this function, we used efficient coding for docIDs, to build the inverted index by first taking differences, and then encoding those smaller numbers.
- stopwords(w,hashlist): Filter out and not to index the word in stopwords list
- decorated_file(f, key): To yields an easily sortable tuple.
- standard_keyfunc(line): The standard key function in my application.
- saveUrlTable(paths, output_path): Merge the urltables and save to file

- mergeSortedFiles(paths, output_path, dedup=True, keyfunc=standard_keyfunc): merge multiple sorted files

query_new.py: The third module that read user's query, execute the query and return the ranked results

- class SimpleQuery: Simple query class
- query_op(self, s, t): Operate the query condition. In this program, we used Python Set intersection operation to compute the common documents in which each term in the query appears.
- loadUrls(self, urlFile): Read the url table file and load it into memory, use a hash table to store it
- loadLexicon(self, lexiconFile):: Read the lexicon file and load it into memory, use a hash table to store it
- query(self, q, op='and'): Query the condition and return the results.
- rank(self, res, words): For each word, first compute the url list res that contains such word using geturllist. Then use Python set intersection to get the final url list res
- bm25_relevance(self, terms, docItem, b=0.75, k=1.2): Given multiple inputs, performs a BM25 relevance calculation for a given document. Terms should be a list of terms. total_docs should be an integer of the total docs in the index. Optionally accepts a b parameter, the default is 0.75. Optionally accepts a k parameter with the default of 1.2
- geturllist(self, q): Get [urlID, frequency] list given a word
- decode7bit(self,bytes): integer decoded function
- encode7bit(value): integer encoded function, which is similar to variable byte encoding

8. Experiment Results:

HOW LONG DOES IT TAKE ON THE PROVIDED DATA SET?

In this project, we used NZ10 dataset, which is 570MB containing 270,000 pages. The index building time is 1 hour 3 minutes. For the NZ2 dataset, the indexing time is 12 minutes.

HOW LARGE THE RESULTING INDEX FILES ARE?

For the NZ10 dataset, we only store URL ID and frequency in index. The index file is approximately 110MB in size without efficient coding for docIDs in index by taking differences, which means:

- if postings with docID and frequency:

(34,1) (68,3) (131,1) (241,2) … -> (34,1) (68,3) (131,1) (241,2) …

However, after encoding the URL IDs in index by taking differences:

- if postings with docID and frequency:

(34,1) (68,3) (131,1) (241,2) … -> (34,1) (34,3) (43,1) (110,2) …

The index file size becomes 50MB.

Some results of the One word queries:

Enter the Query: **penguin**
2012-04-02 16:59:45.971000
['penguin']
www.penguin.net.nz/news/archive/0509.html 10.8572368585
www.conservation.govt.nz/Conservation/001~Plants-and-Animals/004~Seabirds/009~Penguins-in-Canterbury.asp 10.7412550288
www.yellow-eyedpenguin.org.nz/about/index.html 10.6303762177
www.penguin.co.nz/static/html/nz/permission/index.html 10.4625780442
www.penguin.net.nz/cons/conservation.html 10.3879878764

www.penguin.net.nz/faq/faq.html 10.3751044871
www.yellow-eyedpenguin.org.nz/news/news.html 10.328735559
www.penguin.net.nz/viewing/wild.html 10.3098566573
www.yellow-eyedpenguin.org.nz/index.html 10.0716016623
neon.otago.ac.nz/chemistry/research/sab/dunedin/dun22.htm 10.0660321923
2012-04-02 16:59:46.935000
Query time: approximately 1 second

Enter the Query: **kiwi**
2012-04-02 16:59:51.093000
['kiwi']
extranet.doc.govt.nz/content/frontpage/2004/kiwi-chicks-rock.htm 7.36877386088
www.conservation**.co.nz**/Community/002~Events/Conservation-Week/010~2001/Unique-New-Zealand-Education-Resource/005~Kiwi.asp 7.357335928
www.conservation**.org.nz**/Community/002~Events/Conservation-Week/010~2001/Unique-New-Zealand-Education-Resource/005~Kiwi.asp 7.357335928
www**.doc.govt.nz**/Community/002~Events/Conservation-Week/010~2001/Unique-New-Zealand-Education-Resource/005~Kiwi.asp 7.357335928
www.wwf.org.nz/features/05-03-kiwi.cfm 7.33540350197
www.aroha.net.nz/kiwi_agm.html 7.30773181301
doc.govt.nz/Regional-Info/007~Tongariro-Taupo/004~Conservation/Biodiversity/001~Tongariro-Kiwi-Sanctuary.asp 7.30720071293
www.greatwalks.co.nz/Conservation/001~Plants-and-Animals/001~Native-Animals/Maori/001~Kiwi.asp 7.2834529994
www.coolrunning.co.nz/results/2001r200.html 7.28100091833
interim.internetnz.net.nz/issues/kiwishare990617schedule1.html 7.23771962889
2012-04-02 16:59:52.935000
Query time: approximately 1 second

Enter the Query: **wellington**
2012-04-02 17:30:07.222000
['wellington']
www.converge.org.nz/pma/fknionind.htm 4.36389318401
athletics.org.nz/r060702.html 4.35547633096
homepages.paradise.net.nz/%7Edchamber/eq1890s.htm 4.346005775
www.justice.govt.nz/pubs/reports/2003/conviction-sentencing-2002/appendix-1.html 4.31294453364
www.justice.govt.nz/pubs/reports/2000/convict_sentence_2000/appendix_1.html 4.30640701882
www.valleys.org.nz/results/2001/2001r022.html 4.30153942293
socialreport.msd.govt.nz/notes-references/ 4.295042099
www.coolrunning.co.nz/results/2004/2004r128.html 4.29222442845
www.justice.govt.nz/pubs/reports/1999/convict_sentence/appendix_1.html 4.28653040132
athletics.org.nz/nr110801.html 4.27358638223
2012-04-02 17:30:10.490000
Query time: approximately 3 seconds

Some results of the Free text queries (AND strategy):

Enter the Query: **room host**
2012-04-02 03:48:19.745000
['room', 'host']
www.afw.co.nz/hotel/bonus_code_poker_room.html 7.69671965737
room-air-cleaners.interstudent.co.nz/ 7.56759268018
conference-room-table.interstudent.co.nz/ 7.46464861937
gaytravel.net.nz/nz/AverleighCottage.html 7.39596180394
www.explorenewzealand.net.nz/RuapehuHomestead.html 7.26225705934
www.afw.co.nz/hotel/room_steam_twinks_video_voyeur.html 7.24000595155
www.afw.co.nz/hotel/brunson_doyle_poker_room.html 7.20807791899
las-vegas-hotel-room-dancers.thewall.co.nz/ 7.18093943629
travelink.co.nz/nz/DesignerCottage.html 7.12965975824
www.explorenewzealand.net.nz/DesignerCottage.html 7.12950622054
2012-04-02 03:48:22.698000
Query time: approximately 4 seconds

==========

Enter the Query: **penguin kiwi**
2012-04-02 03:52:14.806000
['penguin', 'kiwi']
www.hrobinson.co.nz/products.html 15.5352197463
lists.otago.ac.nz/pipermail/otmc-discuss/2003-November/000238.html 14.7286044888

homepages.ihug.co.nz/~l.orman/autobiographical.htm 14.4637120883
library.auckland.ac.nz/subjects/nzp/nzlit2/hunt.htm 13.9401719976
doc.govt.nz/Publications/004~Science-and-Research/Biodiversity-Recovery-Unit/Rar
e-Bits/PDF/090%7ERare-Bits-No-51-Dec-2003.asp 13.6568419724
www.folksong.org.nz/opo/index.html 13.2520636039
www.ecoworks.co.nz/kiwi&wildlife.html 13.1632334259
www.conservation.org.nz/Community/002~Events/Conservation-Week/011~2000/002~Prog
ramme.asp 12.7362540683
www.kcc.org.nz/species/threatened.asp 12.6781456225
www.penguin.co.nz:8000/nf/Book/BookDisplay/0,,0_0140279946,00.html 12.6309691512

2012-04-02 03:52:17.919000
Query time: approximately 3 seconds

==========

Enter the Query: **meta yoga seek**
2012-04-02 03:55:31.999000
['meta', 'yoga', 'seek']
www.hknet.org.nz/3modesbhagavdG.htm 13.5110938334
2012-04-02 03:55:36.638000
Query time: approximately 4.5 seconds

==========

Enter the Query: **penguin be sick kiwi**

2012-04-02 04:03:09.839000

['penguin', 'be', 'sick', 'kiwi']

no search result.

2012-04-02 04:03:10.757000

Query time: approximately 0.9 second

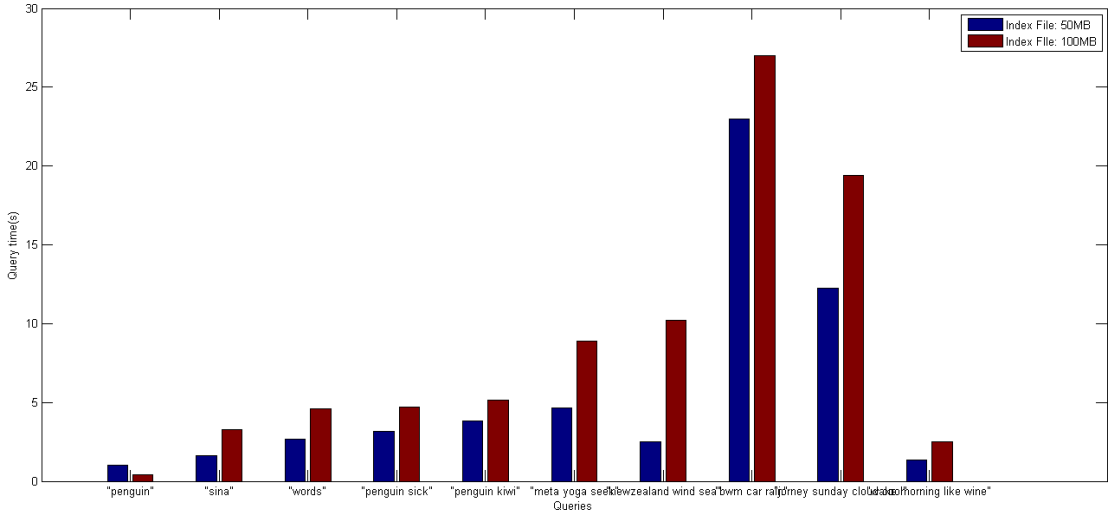


Figure 3: how index file size affects query time

As figure 3 shows, we did 10 queries based on two index files: one with size 50MB with efficient coding for docIDs in index by taking differences while the other index file with size 100MB does not take differences for docIDs. From the experiment result, it can be concluded that the query time drops greatly if the index file is compressed into smaller size.

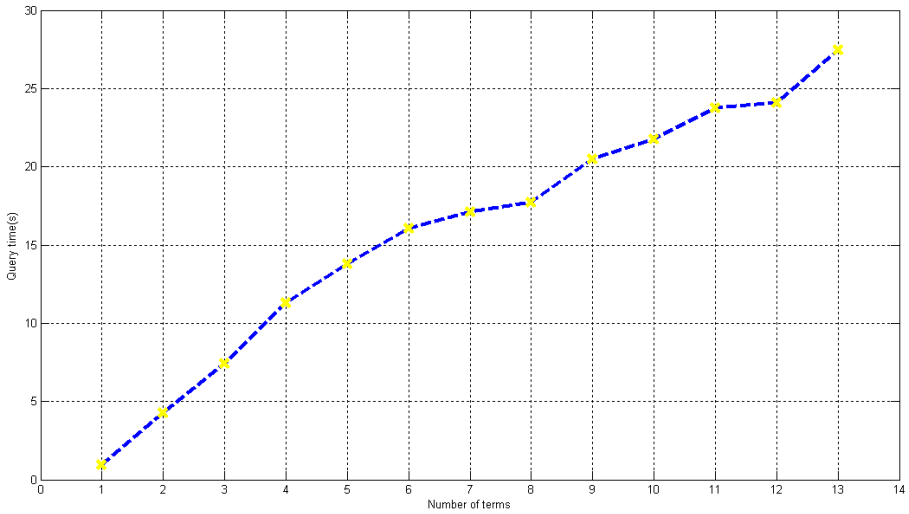


Figure 4: how number of terms affects query time

As demonstrated in Figure 4, we did 13 queries starting from 1 term to 13 terms. The corresponding query time increased nearly linear but proportional with the length of the inverted

index. The queries were:
penguin
penguin kiwi
penguin kiwi summer
penguin kiwi summer auckland
penguin kiwi summer auckland beautiful
penguin kiwi summer auckland beautiful sun
penguin kiwi summer auckland beautiful sun wine
penguin kiwi summer auckland beautiful sun wine green
penguin kiwi summer auckland beautiful sun wine green cool
penguin kiwi summer auckland beautiful sun wine green cool journey
penguin kiwi summer auckland beautiful sun wine green cool journey sea
penguin kiwi summer auckland beautiful sun wine green cool journey sea cloud
penguin kiwi summer auckland beautiful sun wine green cool journey sea cloud pink

9. Conclusion

In this project, I implemented a lightweight search engine that embraces three modules: parser, index builder, and query execution. All modules are implemented in Python. This program, which has met all the minimal requirements of assignment 2 and 3, is maintained and extended easily. Also there is still much work should be done to make it more robust and faster, such as chunkwise index compression, phased query execution, and other ranking functions.

Reference:

[1] http://en.wikipedia.org/wiki/Inverted_index
[2] http://www.nltk.org/
[3] David Carmel, Einat Amitay, Juru at TREC 2006: TAAT versus DAAT in the Terabyte Track, TREC 2006.
[4] Hao Yan, Shuai Ding, Torsten Suel, Inverted Index Compression and Query Processing with Optimized Document Ordering, WWW 2009, April 20–24, 2009, Madrid, Spain.
[5] http://en.wikipedia.org/wiki/Okapi_BM25