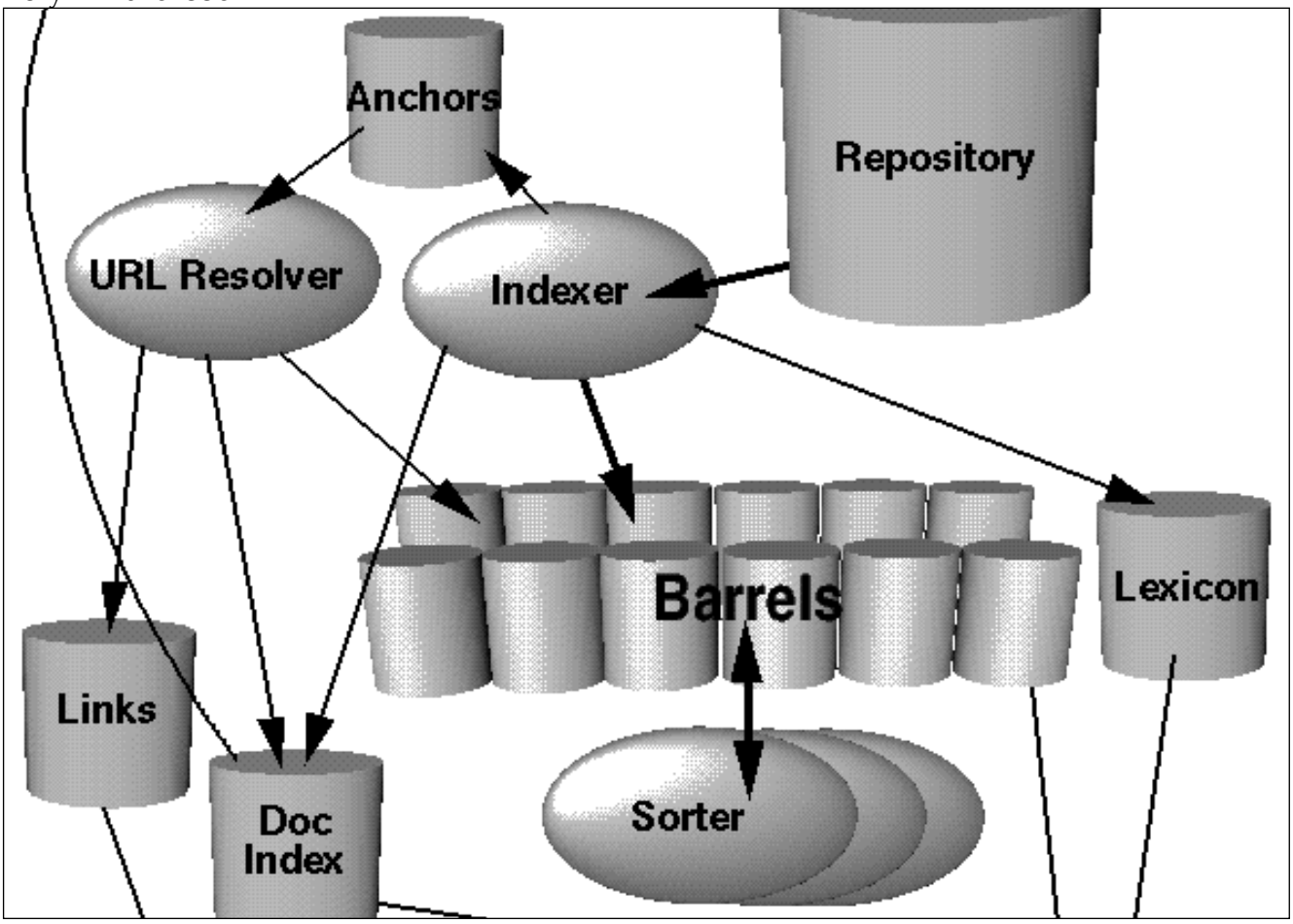
Indexer and query processor

Web Search Engines Homework 2&3

Hongzheng Shi - March 26, 2014
PolyID : 0484338



Introduction

This project consists of two parts: a indexer generator written in C++, which can parse downloaded web pages, generate inverted lists, url table and lexicon table in compressed form; and a query processor written in Python, which takes in user queries, consults index files generated from the first part, then returns ranked list of urls related to the query keywords and snippets around them.

This project is part of a very primitive web search engine project. It implemented basic parts and realized necessary functionalities, but most of the methodologies are not optimal, hence the performance is limited. Detailed analysis will be discussed in conclusion part as follows.

Modules

The two main modules are Indexer and Query processor, the latter consumes index files generated from the former. Detailed system design is shown in the following Figure1:

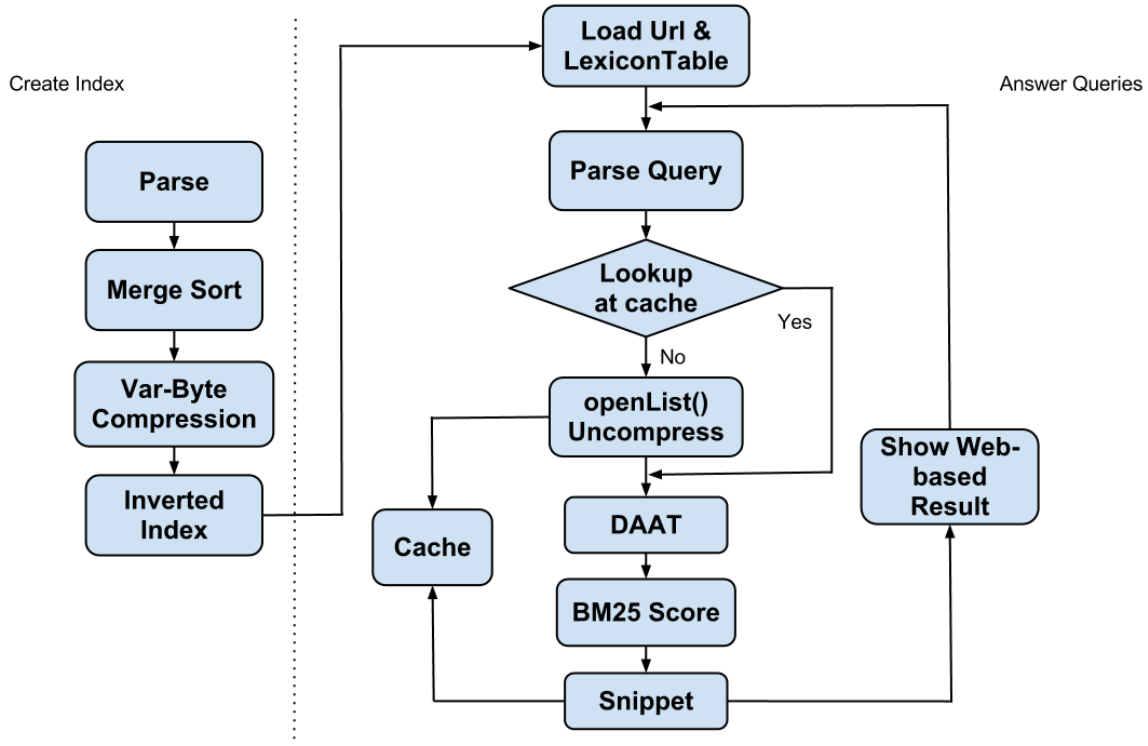


Figure 1 : Module diagram

1. Indexer

Left part of figure 1 indicates how the indexer work.

The indexer will generate:

1. inverted-list files
form: {docID,freq,pos,cont}
2. lexicon file
form: {word,num,invertedListFileID,pointerInIVListFile,sizeOfList}
3. URL table file
from: {url, invertedListFileID, pointerInIVListFile, numWords} where docId is array index from compressed web page files and corresponding index files.

Main components of the indexer:

1. ResourceLoader:
load one %d_data and corresponding %d_index file into memory, return contents as vectors
2. Parser:
This class will parse page file, write postings to vector and return word count for given page
3. RawPage
This class defines RawPage data structure
4. UrlTable
This class defines URLtable structrue and its functions
5. LexiconTable
This class defines LexiconTable structrue and its functions
6. InvertedIndexGenerater
This class calls other components to implement following functions:
    1) reads source files into mem, parse them into postings, aggregate and sort before writing to temp files to disk
    2) merge-sorts temp generated files into one file
    3) scan merged file, aggregate freq, compose final index for each word
    4) in the mean time, generate lexicon table, url table
7. vbyte.h
header file containing macros for vbyte compression
8. Config.h
define global configurations
9. main.cpp
driver class

Detailed workflow of indexer:

1. ResourceLoader provides an API vector<RawPage>* getPages(), encapsulates details about how the program get data from source, several

problems addressed:
a. maintain a list of files in data source folder, and a pointer recording current file number
b. on each call to getPages(), ResourceLoader reads next index file to mem, uncompress it, get a list of pointers to corresponding data file
c. load corresponding data file into mem, uncompress it, using index records to read individual pages, return them as a vector

2. How does the program generate temp index files?
a. ask ResourceLoader for next bunch of pages
b. for each page, parse it to get raw postings, append postings to buffer
c. if buffer size is large enough, call WriteRawPostingToFile() to sort by (word,docID), then write to a new temp file
d. repeat above steps until no more page to load

3. How does the program merge temp index files?
Let unix sort do the job, this step will merge all temp files into one intermediate file

4. When to generate UrlTable?
While generating temp files, keep it in mem, write to file only at last.

5. How final invertedList is generated?
Scan merged file from former steps, now postings on same word appear consecutively. Iterate through the whole file
and merge postings, compress and write into file

6. What happens also while generating inv-list?
generate lexicon file.

Discussions:

Indexer is the most compute oriented part in web search engines, efficiency is extremely import to completing indexing millions of pages on my little laptop within a reasonable time. In this case, I decided to implement it with C++.

For compression, I used varbyte, the simplest byte-wise compression mentioned in class, it worked well in the project: shrinking inverted index files size from about 7GB to only 3GB.

Performance:

(1). nz2.tar:
Size of inverted index: 275MB
Time used: 180 seconds

(2).about 1GB data from NZ_ALL:
Size of inverted index: 1.37GB
Time used: about 50 minutes

(3).NZ_ALL:

size of inverted index: 2.75GB
Time used: 9190 seconds = 153 minutes = 2.5 hours

2. Query processor

Overview

The query processor gets queries from web server, parsing queries into separated key words, using space as delimiter. I'm implementing only AND semantic, which means only urls with page content containing ALL of the key words will be considered as search results.

Components:

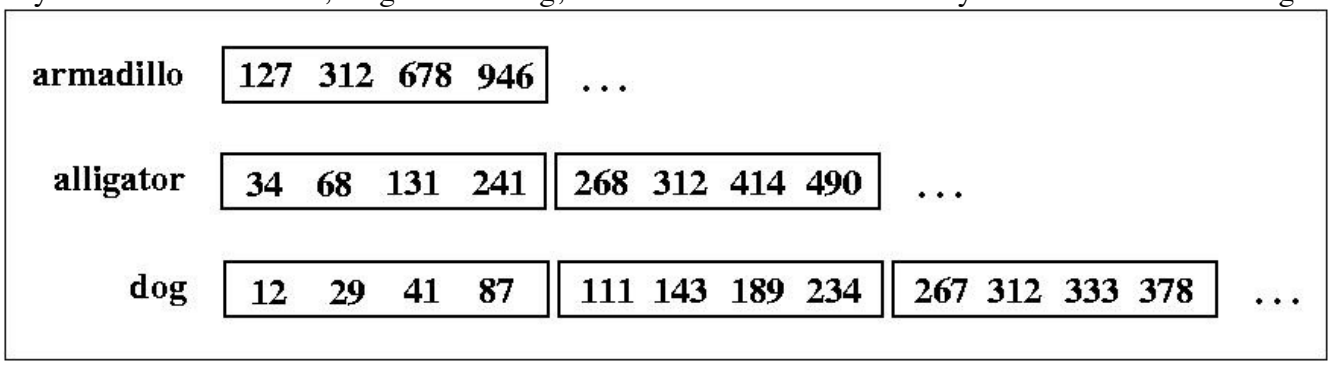
1. queryhander.py: defines query handler for the whole program.
   This module parses incoming keywords, call backend to get results, and present them as html pages
2. query.py: defines Query class which implements queryWords(query,start,limit) API, as well as daatQuery(words) function, daat query process will be explained soon. Also, snippets for each result url are generated.
3. readpage.py: given docID, read from raw files the corresponding page.
4. index.py: work together with lexicon table to get inverted list from given word, load lists into memory, and cache them.
5. lexicon.py: load lexicon table from disk.
6. urltable.py: load urltable from disk.
7. vbyte.py: implements decode function to get data objects from binary stream
8. bm25.py: get bm25 score of given result for ranking. details of bm25 see following
9. singleton.py: defines a python class decorator for singleton. Singleton is needed a lot in this project since many classes like Lexicontable, Query, should be instantiated only once to merit from cache.
10. config.py: global configurations.
11. main.py: driver, start web server loop.

DaatQuery process:

Daat query processing is used in this project. The big idea in this methodology is to encapsulate lower level operations such as reading binary and GZIPed files, searching and comparing docIds in different inverted lists into several simple API, so that higher level operations, in this project query processing, can use directly. This separation of logic can simplify programming procedure and end up with a multiple tiered system that can be maintained separately.
The main APIs of daat query processing are:
- *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. Return value > MAXDID if none exists.
- *getFreq(lp)* get the frequency of the *current* posting in list *lp*

Simple example how to process a query:
key words are: armadillo, alligator and dog, assume inverted lists of the keywords starts as following:



- open all 3 inverted lists for reading using *open()*
- this returns 3 pointers *lp0*, *lp1*, and *lp2* to the starts of the lists
- call *d0 = nextGEQ(lp0, 0)* to get docID of first posting in *lp0*
- call *d1 = nextGEQ(lp1, d0)* to check for matching docID in *lp1*
- if *(d1 > d0)*, start again at first list and call *d0 = nextGEQ(lp0, d1)* • if *(d1 = d0)*, call *d2 = nextGEQ(lp2, d0)* to see if *d0* also in *lp2*
- if *(d2 > d0)*, start again at first list and call *d0 = nextGEQ(lp0, d2)*
- if *(d2 = d0)*, then *d0* is in all three lists; add it to result set; then continue

at first list and call *d0 = nextGEQ(lp0, d2+1)*

Note in my implementation, all results are added into results set, scores are calculated separately later, this works fine for the project since the total amount of data is relatively small compared to a real world web search engine.

bm25 ranking

Daat query processing will return a long list of urls satisfying input query words, not all of them are of equally value to user. Based on this observation, most web search engines , if not all of them, will sort the results by some kind of ranking score, and present urls with better scores to users with higher priority.
In this project, bm25 is used to calculate score for each result that got through.
Okapi BM25 is a ranking function used by search engines to rank matching documents according to their relevance to a given search query.

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

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

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 $b = 0.75$.

Snippets

Snippets refers to a short piece of words surrounding query keywords in result pages. Offering snippets when representing search results can help user determine which urls are more helpful without even clicking any url.

In this project, we get position of keywords from inverted index, 50 characters before and 100 characters after first occurrence of query words are provided as snippet. This is not optimal since ideally some ranking based on context should help make better snippets.

Caching

Caching is used in this project: Index class keeps inverted list cache for recently loaded lists; snippets for given words are cached; pageReader class will cache pages by docIds so further request on the same page will be fast.

A limitation in this project is there is no expiration scheme for caches, everything visited will stay in the cache, so in the long run the cache size will just explode. However this works fine for simple demos.

About Pypy

When the first time I run the project with the whole index data, the performance was frustrating, a query like "hello world" took more than 20 seconds to return even after all tables are loaded into memory and inverted lists are cached!!

Deeply look into execution time I noticed the nextGEQ() methods are too slow even though the whole inverted list is already loaded in memory. I blame the poor efficiency of python as an interpreting language for the performance.

Pypy is an execution environment for python codes which uses JIT compilation to make python code run real fast. By applying pypy, the running time became hundreds time faster, without modifying a line of code.

Performance
For query processor, I used index from NZTOTAL.

Loading Lexicon table and url table takes about 10 seconds, this happens only once during first query.

Search for “test”, for the first time, loading tables time included, 17301.70ms:

```
server started.
loading urltable...
urltable size: 2635521
num of lists: 1
loading lexicon...
lexicon size: 3005481
time to openlist: 11.743501
time sorting list: 5.09999999991e-05
total time nextGEQ: 0.0673429999999
total time bm25:  0.102862000001
total time to query: 16.748451
result set size: 122465
time to sort by BM25: 0.150204
start index & end index: 0 10
time to get snippets: 0.299581
[I 140330 18:44:29 web:1728] 200 GET /?query=test (::1) 17301.70ms
```

Search for “test” again, only 576.53ms

```
num of lists: 1
time to openlist: 4.6999999995e-05
time sorting list: 7.50000000025e-05
total time nextGEQ: 0.072783000001
total time bm25:  0.0842709999988
total time to query: 0.404985
result set size: 122465
time to sort by BM25: 0.152933
start index & end index: 0 10
time to get snippets: 0.000118999999998
[I 140330 18:45:12 web:1728] 200 GET /?query=test (::1) 576.53ms
```

Search for “armadilo”, without cache, 230.14ms

```
num of lists: 1
time to openlist: 0.000343999999998
time sorting list: 4.50000000001e-05
```

```
total time nextGEQ: 1.40000000002e-05
total time bm25:  6.49999999993e-05
total time to query: 0.000669000000002
result set size: 1
time to sort by BM25: 1.89999999947e-05
start index & end index: 0 1
time to get snippets: 0.140562
[I 140330 18:46:55 web:1728] 200 GET /?query=armadilo (::1) 230.14ms
```

“armadilo” with cache, 1.09ms

```
num of lists: 1
time to openlist: 1.99999999992e-05
time sorting list: 5.80000000028e-05
total time nextGEQ: 1.19999999981e-05
total time bm25:  3.9000000001e-05
total time to query: 0.000261000000002
result set size: 1
time to sort by BM25: 1.60000000022e-05
start index & end index: 0 1
time to get snippets: 2.40000000034e-05
[I 140330 18:47:53 web:1728] 200 GET /?query=armadilo (::1) 1.09ms
```

“cat dog”, no cache, 530ms:

```
num of lists: 2
time to openlist: 0.189903
time sorting list: 5.30000000012e-05
total time nextGEQ: 0.0835030000004
total time bm25:  0.0649380000001
total time to query: 0.435976
result set size: 9082
time to sort by BM25: 0.008174
start index & end index: 0 10
time to get snippets: 0.07739
[I 140330 18:48:49 web:1728] 200 GET /?query=cat+dog (::1) 530.65ms
```

“cat dog”, cached, 104ms:

```
num of lists: 2
time to openlist: 4.50000000001e-05
time sorting list: 8.10000000016e-05
total time nextGEQ: 0.0421440000001
total time bm25:  0.0151189999997
total time to query: 0.097046
result set size: 9082
time to sort by BM25: 0.006286
start index & end index: 0 10
time to get snippets: 0.000118999999998
```

```
[I 140330 18:49:25 web:1728] 200 GET /?query=cat+dog (::1) 104.85ms
```

“to be or not to be”, without cache,20s:

```
num of lists: 6
time to openlist: 14.963412
time sorting list: 6.70000000014e-05
total time nextGEQ: 2.023691
total time bm25:  0.438990999996
total time to query: 19.181914
result set size: 294431
time to sort by BM25: 0.414292
start index & end index: 0 10
time to get snippets: 0.499935
[I 140330 18:50:47 web:1728] 200 GET /?query=to+be+or+not+to+be (::
1) 20866.41ms
```

“to be or not to be”, with cache, 4s:

```
num of lists: 6
time to openlist: 6.60000000039e-05
time sorting list: 9.40000000043e-05
total time nextGEQ: 2.128238
total time bm25:  0.460525999999
total time to query: 4.347938
result set size: 294431
time to sort by BM25: 0.400129
start index & end index: 0 10
time to get snippets: 0.000343000000001
[I 140330 18:51:39 web:1728] 200 GET /?query=to+be+or+not+to+be (::
1) 4787.28ms
```

From above results, cache worked well on saving openList(), and getSnippets() time, which together consumed most of query execution time. Lexicon list and url table are sitting int the memory. Other operations are mostly CPU bounded, nothing much to cache. In all, the caching scheme worked well.

How to run

Indexer:
Follow these steps:
1. Open JIndexer.xcodeproj with Xcode
2. Copy source data folder into path specified by "Config.h", or change "Config.h" according to source data path
3. Run from XCode
4. Check output files in path specified by “Config.h"

Query processor:

1. install pypy, add pypy into PATH
2. In the project folder, user command line “pypy main.py” to start server
3. visit localhost:8080

Limitations

Limitations of each part are discussed in component descriptions.