Homework 2 Summary
Si Chang

This index generation system is build in java, and there are two parts of this system.

One is for decompress data files, extracting web pages, parsing web pages, splitting words, stemming words, remove illegal words and stop words, and generating lexicon, URL table, id, word mapping table and intermediate posting files.

One is for IO efficient merge function and generating final byte format inverted index structure.

Statistics

All the following data is for nz10 data set.

It takes 2892 seconds to generate partially sorted postings.
It takes 187 seconds to merge postings and generate final index file.

The index file is 368MB

The final index format like this, [4 bytes for length of the index of each word][3 bytes for docid][1 byte for frequency][3 bytes for position(20 bits) and context(4 bits)]

Limits

The parser sometimes does not work well, and there are some data files I can not parse, which are “data_154”, “data_294”, “ data_295”, “data_345”.

How to run it?

There are two Java projects, one is named “indexing” and another is named “Merge”.

1. Open eclipse and import “indexing” project.

2. Set the number of thread to generate posting files. Open “PostingGenerator.java” and find the following lines.

```
//                  define the size of the thread pool
ThreadPool pool = new ThreadPool(13);
```

13 is the number of thread and you can change it to increase the execution speed.

3. open “PostingGenerationThread.java” and find this line

```
String index_file = "D:\\web search engine\\homework2\\nz10\\nz10_merged\\"
                                        + this.filename + "_index";
String data_file = "D:\\web search engine\\homework2\\nz10\\nz10_merged\\"+this.filename +
"_data";
```

to modify the path of the zipped index files and data files.

4. it takes 2892 seconds to finish this phase and the generated files include posting files, such as “postings_1”, lexicon file, such as “lexicon.seri”, word id mapping table, such as “wordid_generator.seri”. The posting files' format is like “1,4780,78,A”. And the first number is word id, second number is doc id, thrid number is position, and the fourth item is word context.

5. Open eclipse and import “Merge” project.

6. Open the “IOEfficientMerge.java” file, and find the following lines,

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream(
        "D:\\workshop\\indexing\\lexicon.seri"));
Lexicon lexicon = (Lexicon) in.readObject();
in.close();

in = new ObjectInputStream(new FileInputStream(
        "D:\\workshop\\indexing\\wordid_generator.seri"));
WordIDGenerator wordid_generator = (WordIDGenerator) in
        .readObject();
in.close();

String filename = "D:\\workshop\\indexing\\postings_";
ArrayList<String> filenames = new ArrayList<String>();
for (int i = 0; i < 414; i++) {
    if (i != 154 && i != 294 && i != 295 && i != 345) {
        filenames.add(filename + String.valueOf(i));
    }
}
```

modify the according path of the “lexicon.seri”, “wordid_generator.seri”, and “postings_” files.

7. go to the following lines to configure the IO efficient functions.

```
IOEfficientMerge merge = new IOEfficientMerge(filenames.size(),
        filenames, 1000000, 400000000, "index ");
```

The first parameter is the number of the input buffers. The second parameter is the input posting files' names. The third one is the size of input buffer.(byte) the fourth one is the size of the output buffer. (byte) The fifth one is the prefix of output index file's name.

8. run “IOEfficientMerge.java” file to generator the final inverted index files and it is the byte format. Also save the name of the index file and it's position to corresponding word in lexicon.
9. It takes 187 seconds to finish this process.

How it works?

Unzip class

I employ java api java.util.zip.GZIPInputStream to finish this job, and pass the unzipped index and data files to next step operation.

Split_pages class

Based on the index file I extract the URLs and corresponding pages from data files and return.

HTMLParseLister class

I employ HTMLEditorKit.Parser class to parse the HTML pages, and I implement the HTMLParseLister to process different tags and text between tags.

TokenizerNew class

For each text between tags I split text into words based on the delimiters which I defined " \t\n\r\f~!@#$%^&*()_+|`-=\\{}®[]:\";'<>?,./'—'0123456789". Also I split compound words into single words, such as split "TableTennis" into table and tennis. I also make use of org.tartarus.snowball.SnowballProgram to stem words into normal style.

StopWords class

I employ this class to remove stop words and I define the stop words in "stopwords.txt" file.

Lexicon class

This class maintains a lexicon for all words, include word, doc number, index page and position. I employ ConcurrentHashMap to implement the lexicon to increase the query speed. Also because I make use of multi-thread programming, this data structure can also control concurrent modify well.

WordIDGenerator class

In this class, I build a word and id hash table, and a concurrent control for generating wordid.

DocidGenerator class

In this class, I build a concurrent control for generating docid.

URLTable class

In this class, I build a URL table. The data structure includes three fields: docid, URL, size.

TreeSet class

I also make use of TreeSet to store postings. Because it implements a binary tree, and the root of the tree is always the smallest one. When I finish inserting postings, it is already sorted. This can greatly speed up the sorting process.

InputBuffer class

This class maintain a input buffer for one posting file. You can define the size of it and when it is empty, extract next chunk from file, until process all postings in this file.

OutputBuffer class

This class maintain a output buffer for merged postings. This write bytes into files when it is full. For each new word, it returns file name and position mappings.

PriorityQueue class

I make use of Java API PriorityQueue to maintain a priority queue for the heads of input buffers during merge operation. The inner implementation of PriorityQueue is a binomial heap. And each time extract the minimum item.