Assignment #2 Inverted Index

Chaoqun Huang ch2958

1 Introduction

In this assignment, I write a program in java that creates an inverted index structure from a set of downloaded web pages provided by common crawl. The program will first read the text from the pages asynchronously and generate intermediate files and then combine and sort them by using unix sort. Finally build an inverted index and lexicon index on top of the sorted postings. There will be four expected outputs from my program:

- Inverted index
- Lexicon index
- Url table
- Word table

2 How my program work

My program can be divided into three parts:

- Generate postings
- Sort the postings
- Build the lexicon and inverted index

2.1 Generate postings

The first part of this program is to generate intermediate postings. My source data is from common crawl in wet data format. Wet data format contains only the metadata and extracted plain text data. Therefore, it saves me the trouble to write my own html parser.

Although the wet format contains only the plain text, there are text in different languages. Languages like Chinese are really hard to deal with, so in this assignment, only pages containing only English (latin alphabet) will be processed. The “CommonCrawlReader” class will first be called to combine the text belongs to the same html page, any line that cannot be encoded in ASCII will be filtered out. The way to check if it is ASCII or not is to call the google Guava api. Then, the “posting” class will start to generate postings by split the html page into

words and do an iteration to get the wordList which is a hash map with word as key and frequency as value. Finally write the postings (Word, DocId, Frequency) into temp files.

CommonCrawlReader will have a fixed thread pool, which can assigned by the user. By default, 20 threads are initialized to parser the files asynchronously. However, since there is only one intermediate posting file, the “write to file” operation is done synchronously by a lock on the file when writing.

Given sufficient memory and computation power, the size of the thread pool should be the number of files. In this way, one thread handles one file, which should be the most efficient way to parser. However, limited by the IO write/read rate, 20 threads or less on my mac can lead to 100% cpu usage.

```
ExecutorService es = Executors.newFixedThreadPool(20);
CommonCrawlReader commonCrawlReader = new CommonCrawlReader();
CompletableFuture<Void> runAsync = new CompletableFuture<>();
for (String wet : FilePath.WETS) {
   runAsync = CompletableFuture.runAsync(() -> commonCrawlReader.startParser(wet), es);
}
while (true) {
   if (runAsync.isDone()) break;
}
```

Code 1-1: Parsing Asynchronously

The Url table is generated in this step as well. A url table includes an Id and url in String Url. The url table is considered to fit in main memory, so it is stored in ASCII format which makes it easier to read.

2.2 Sort the postings

After generating the intermediate posting, the next step is to sort these postings by the word id, so that lexicon and inverted index can be built on top of that.

The sorting is done by calling unix sort. Unix sort is an I/O efficient sorting utility provided by GNU. By default, the Unix sort will use as many processors as it can when sorting. So by command:

```
Sort -k1,1 -k2n,2 postings_intermediate > postings_sorted
```

Code 1-2: Unix Sort

“SortUtil” class is used to sort postings given the postings file path.

2.3 Building the lexicon and inverted index

The lexicon and inverted index is built on top of the sorted postings. A lexicon consists of a word Id, its start pointer, its length, and how many docs contains the word. As for inverted index, basically they are pair of numbers, a doc id and frequency. A word table mapping the String word to word id will also be generated in this step.

For example, if a user wants to search document contains word “dog”. First, the word id for dog will be found in word table and then using this word id, a start pointer and length can be read from the lexicon index. Finally, the doc id and its frequency can be read by starting pointer and length in inverted list.

There are three output in this step, a lexicon index, an inverted index and word table. The program will scan the intermediate postings. Since, the intermediate postings are sorted by word id, the same word id will be next to each other. It will keeps reading these postings until find a new word id. The start pointer, length and count will be written to lexicon file.

Inverted index is considered not to fit in the main memory, so all the doc id and frequency is compressed using Variable byte (VB) encoding(VByte). VByte 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. To decode a variable byte code, we read a sequence of bytes with continuation bit 0 terminated by a byte with continuation bit 1.

```
public static byte[] encode(int value) {
  List<Byte> res = new ArrayList<>();
  while(value > 127) {
    res.add((byte)(value & 127));
    value>>>=7;
  }
  res.add((byte)(value | 0x80));
  Byte[] bytes = res.toArray(new Byte[res.size()]);
  return ArrayUtils.toPrimitive(bytes);
}
```

Code 1-3: VByte encoding

The “VByteCompress” class to encode and decode the inverted index.

3 Design decisions

- DocId: doc IDs are assigned in the order the pages are parsed. The wet format has already sort the pages in it by url. Therefore, assigned doc id in this way will still keep the documents “clustered” (Url in the same domain is next to each other)
- Word Id: The word is assigned by the alphabetically, so that if the words "dog" and "dogs" are next to each other in the sorted list of all words, then their inverted lists are also next to each other in the inverted index. So query like “dog*” will be much more efficient.
- Sorting: the intermediate postings are using string word and is sorted by string word to keep it clustered.

4 Conclusion

To run the program:

- Import the dependencies (Google Guava, Sun langs) using maven
- Write the wet files path in File Path class
- Run the main function

The first Run:

30 Wet files:

CC-MAIN-20170919112242-20170919132242-00000.warc.wet.gz to
CC-MAIN-20170919112242-20170919132242-00030.warc.wet.gz

Wet files	10 G
Pages	1274330
Running time	80 Minutes
Postings size	5.07 G
Lexicon	199 M
Inverted Index	1.27 G
Url Table	100 M

Word Table	167 M
Pages per Second	265

The second run:
60 Wet files:

CC-MAIN-20170919112242-20170919132242-00000.warc.wet.gz to
CC-MAIN-20170919112242-20170919132242-00060.warc.wet.gz

Wet files	20 G
Pages	2607930
Running time	180 Minutes
Postings size	10.13 G
Lexicon	401M
Inverted Index	2.6G
Url Table	201 M
Word Table	258 M
Pages per Second	241