CS 660 - Data Science at Scale

Lab 4: Designing and Tuning MapReduce Algorithms

Group: Eric Jiang, Joonsoo Park, Franco Pettigrosso, McWelling Todman

1 Read and summarize chapter 4 on Inverted Indexing for Text Retrieval

The purpose of this chapter is to explain the use and technical advantages of Inverted indices. The preliminary example used to illustrate how inverted indices can be applied in practice is the robust process of crawling the web in order to construct an inverted index that will facilitate web search on a platform such as Google. The chapter provides a brief overview of web crawling, emphasizing that a proper web crawler should not overload web servers with high frequency requests, visit the same page multiple times, and avoid collecting redundant content.

An inverted Index, as defined in this book, consists of a term (unique item) and a postings list. The postings list contains postings, which are comprised of a document ID and a Payload. The payload is typically the number of times a particular term appeared in a document (term frequency), or in some cases it can be nothing. More advanced indices often include information such as the various positions in which the term appeared within the document, and whether or not that position is in a heading. Next, we are presented with the logic for a systematic approach to collecting terms from documents, then sorting them out for aggregation and indexing. The authors then explains how the logic is implemented in the Map Reduce framework.

Without getting into too much detail, the logic of the pseudo code calls a sort on the postings for each term. This is done in order to ensure the the user gets the best possible matches to their query. In order for a computer to execute the sort, it needs to bring all of the postings into memory, since there is no guarantee the data from the mapper is going to come in order. As you might recognize, this places a hard cap on index size, as a reducer will fail if the machine runs out of memory. In order to overcome this cap, we can once again Invert the Index by making a tuple of the Term and the doc id. The Map reducer run time will then implicitly sort our data during the shuffle/sort phase prior to reduction.

Storing such amount of data is not an easy task. While the HDFS can handle a lot of data, more efficiency is needed when performing operations with the enormous data sets often encountered at the enterprise level. If we were to utilize normal data types such as int 32, it is more than likely we would run into spatial constraints. The solution to this problem presented by the authors is compression, which saves space at the expense of time. Compressing the data reduces space, however the decompression process creates an additional step, which means additional time must be spent. One important technique touched on by the authors is "D-gaps" which involves subtracting an appropriately sized constant from each number starting from one point and moving down a line. This makes the numbers smaller, thus requiring fewer bits to store each one. It is important to note the technique only works with numbers greater than zero.

There are two schools of thought when it comes to incorporating compression into your inverse index. The two approaches presented are byte/word alignment and bit alignment. Word/Byte alignment is similar to counting every 8 bits as a number and the last bit is the delimiter. Thus, 1-127 can be represented via 1 byte, then 128-16,383 is represented in 2 bytes. Doing this with words is essentially the same idea, however we are stuffing a bunch of numbers in a 16-64-bit words. Word/Byte alignment is not the best way to conserve space because there is a lot of space is wasted due to the requirement that buckets size be a multiple of 8.

The next way, and noted the best way, is to do bit alignment. The tricky part is doing it in such a way that we can consistently distinguish one number from another. For example, if we want to compress $0 \geq x < 3$ into a couple of bits using a bit aligned scheme, we need a mechanism to delimit our code representations. We can easily do 0,1,01, however that will not work because 0 prefixes 01 violating our need for a reliable delimiter scheme. We could instead do 00, 01, 1, which works out perfectly because it results in a single unambiguous segmentation in all cases: if we get a 0, then we can of see a 0 or a 1 then new number. If we get a 1 then we can assume its a 2 and move to the next number. The book shows use three ways how to accomplish this. The methods are Unary, Y, and the Golomb method. For our purposes, the best way to compress the data is to use D-gaps and the Golomb method.

2 Implement, using MRJob, the algorithms presented for inverted indexing

```
\usr\bin\env python3.7
'''
InvertedIndexingGoogle.py
reporduce the logic in chapter
eric, park, tod, franco
'''
from mrjob.job import MRJob
from mrjob.step import MRStep
import os #needed to get the name of the file
import re #to get words

WORD_RE = re.compile(r"[a-z|A-Z]+")

class MRMostUsedWord(MRJob):
    '''
    get the name of the file and takes the words
    of each line given and and assigns a record
    and a count
    '''
    def get_files_and_contents(self, _, line):
        book = str(os.environ['mapreduce_map_input_file']).split('/')[-1]
        for word in WORD_RE.findall(line):
            yield [word.lower(), book], 1

'''
adds the counts together so make things easier
on the reducer
'''
def combine_word_terms(self, term_book, count):
    yield term_book ,sum(count)

'''
same thing as the combiner
'''
def put_counts (self, term_book, count):
    yield term_book[0], [term_book[1], sum(count)]

'''
gets are the same term and puts
them in a list after, yields the
the term and the frequencies per
file
'''
def reducer(self, term, postings):
    P = []
    for post in postings:
        P.append(post)
    yield term, P

'''
we want to know the order of the
terms by frequency, or the number
of times they appear, in all the
documents. That happens here.
'''
def get_frequency(self, term, postings):
    string_builder = ''
    frequency = 0
    records = []
    for r in postings:
        records.append(r)
    for record in records:
        for book, count in record:
            frequency += count
    string_builder += "{0} {1} ".format(frequency,term)
    for record in records:
        for book, count in record:
            string_builder += "{0} {1}".format(book,count)
        yield None, (frequency, string_builder)

    '''
    we want to show the results of
    all of our hard work. The information
    gets sorted at runtime.
    '''
    def show_results(self, _, freq_term ):
        for number, letter in sorted(freq_term, reverse = True):
            yield int(number), letter

    def steps(self):
        return [
            MRStep(
                mapper=self.get_files_and_contents,
                combiner=self.combine_word_terms,
                reducer = self.put_counts
            ),
            MRStep(reducer=self.reducer),
            MRStep(reducer=self.get_frequency),
            MRStep(reducer=self.show_results)
        ]

if __name__ == '__main__':
    MRMostUsedWord.run()
```

3 Run and measure performance on Google Cloud with varying number of nodes

Tests:

1. python3.7 InvertedIndexingGoogle.py -r dataproc –num-task-instances 5 gs://cs600/Test/*.txt ... 8 mins
2. python3.7 InvertedIndexingGoogle.py -r dataproc –num-task-instances 3 gs://cs600/Test/*.txt ... 10 mins
3. python3.7 InvertedIndexingGoogle.py -r dataproc –num-task-instances 2 gs://cs600/Test/*.txt ... 12 mins