UNIVERSITÉ BISHOP'S UNIVERSITY

Bishop's University

CS 450 – Elements of Big Data / CS 550 – Big Data Management and Analytics

Assignment 2: MapReduce with Hadoop

1. Introduction

The objective of this assignment is to learn how to design a distributed solution of a Big Data problem with help of MapReduce and Hadoop. In fact, MapReduce is a software framework for spreading a single computing job across multiple computers. It is assumed that these jobs take too long to run on a single computer, so you run them on multiple computers to shorten the time. Some of these jobs are summaries of daily statistics where running them on a single machine would take longer than one day. MapReduce is done on a cluster, and the cluster is made up of nodes. MapReduce works like this: a single job is broken down into small sections, and the input data is chopped up and distributed to each node. Each node operates on only its data. The code that's run on each node is called the mapper, and this is known as the map step. The output from the individual mappers is combined in some way, usually sorted. The sorted data is then broken into smaller portions and distributed to the nodes for further processing. This second processing step is known as the reduce step, and the code run is known as the reducer. The output of the reducer is the final answer you're looking for.

The advantage of MapReduce is that it allows programs to be executed in parallel. If our cluster has 10 nodes and our original job took 10 hours to process, a MapReduce version of the job can reach the same result in a little more than 1 hour. For example, say we want to know the maximum temperature in Canada from the last 100 years. Assume we have valid data from each province for each day over this period. The data looks like <province> <date> <temp>. We could break up the data by the number of nodes we have, and each node could look for the maximum among its data. Each mapper would emit one

temperature, <”max”><temp>. All mappers would produce the same key, which is a string “max.” We’d then need just one reducer to compare the outputs of the mappers and get our global maximum temperature.

At no point do the individual mappers or reducers communicate with each other. Each node minds its own business and computes the data it has been assigned.

Depending on the type of job, we may require different numbers of reducers. If we revisit the Canadian temperature example and say we want the maximum temperature for each year, the mappers would now have to find the maximum temperature for each year and emit that, so the intermediate data would look like <year> <temp>. Now we’d need to make sure all values with the same year go to the same reducer. This is done in the sort between the map and reduce steps. This example illustrates another point: the way data is passed around. Data is passed in key/value pairs. In our second Canadian temperature example, the year was the key and the temperature was the value. We sorted by the year, so we correctly combined similar years; each reducer would receive common key (year) values.

2. Distributed mean and variance mapper

We’re going to create a MapReduce job that calculates the mean and variance of a bunch of numbers. This is for demonstration purposes, so we’ll use a small amount of data. Create a file called mrMeanMapper.py in your favorite editor and add in the code from the following listing.

```
import sys
from numpy import mat, mean, power
def read_input(file):
     for line in file:
          yield line.rstrip()

input = read_input(sys.stdin)
input = [float(line) for line in input]
numInputs = len(input)
input = mat(input)
sqInput = power(input,2)
print "%d\t%f\t%f" % (numInputs, mean(input),
mean(sqInput))

print >> sys.stderr, "report: still alive"
```

This is a straightforward example. You loop over all the input lines and first create a list of floats. Next, you get the length of that list and then create a NumPy matrix from that list. You can then quickly square all the values. Finally, you send out the mean and the mean of the squared values. These values will be used to calculate the global mean and variance.

Let's see the code from listing in action. There's an input file with 100 numbers in a file called inputFile.txt in the source code download. You can experiment with the mapper before you even touch Hadoop by typing the following command in a Linux window:

```
cat inputFile.txt | python mrMeanMapper.py
```

You can also do this on Windows by typing the following in a DOS terminal:

```
python mrMeanMapper.py < inputFile.txt
```

You should see something like this:

```
100 0.509570 0.344439
```

report: still alive

The first line is the standard output, which we'll feed into the reducer. The second line, which was written to standard error, will be sent to the master node to report to the master that the node is still alive.

3. Distributed mean and variance reducer

Now that we have the mapper working, let work on the reducer. The mapper took raw numbers and collected them into intermediate values for our reducer. We'll have many of these mappers doing this in parallel, and we'll need to combine all those outputs into one value. We now need to write the reducer so we can combine the intermediate key value pairs. Open your text editor and create a file called mrMeanReducer.py; then enter the code from the following listing.

```
import sys
from numpy import mat, mean, power
def read_input(file):
for line in file:
```

```
yield line.rstrip()
input = read_input(sys.stdin)
mapperOut = [line.split('\t') for line in input]
cumVal=0.0
cumSumSq=0.0
cumN=0.0
for instance in mapperOut:
     nj = float(instance[0])
     cumN += nj
     cumVal += nj*float(instance[1])
     cumSumSq += nj*float(instance[2])
mean = cumVal/cumN
varSum = (cumSumSq - 2*mean*cumVal + cumN*mean*mean)/cumN
print "%d\t%f\t%f" % (cumN, mean, varSum)

print >> sys.stderr, "report: still alive"
```

The code in listing is the reducer, and this receives the output of the values from listing of the mapper. These values are then combined to form a global mean and variance, which was the goal of this exercise. You can practice with this on your local machine by typing the following at the command prompt:

```
%cat inputFile.txt | python mrMeanMapper.py | python
mrMeanReducer.py
```

In DOS, enter the following command:

```
%python mrMeanMapper.py < inputFile.txt | python
mrMeanReducer.py
```

You will see in the next section how the MapReduce can be automated in Python.

4. Using mrjob to automate MapReduce in Python

mrjob is the easiest route to writing Python programs that run on Hadoop. If you use **mrjob**, you'll be able to test your code locally without installing Hadoop or run it on a cluster of your choice.

Additionally, **mrjob** has extensive integration with Amazon Elastic MapReduce. Once you're set up, it's as easy to run your job in the cloud as it is to run it on your laptop.

Here are several features of mrjob that make writing MapReduce jobs easier:

- Keep all MapReduce code for one job in a single class
- Easily upload and install code and data dependencies at runtime
- Switch input and output formats with a single line of code
- Automatically download and parse error logs for Python tracebacks
- Put command line filters before or after your Python code

If you don't want to be a Hadoop expert but need the computing power of MapReduce, **mrjob** might be just the thing for you.

mrjob lets you write MapReduce jobs in Python 2.7/3.4+ and run them on several platforms. You can:

- Write multi-step MapReduce jobs in pure Python
- Test on your local machine
- Run on a Hadoop cluster
- Run in the cloud using Amazon Elastic MapReduce (EMR)
- Run in the cloud using Google Cloud Dataproc (Dataproc)
- Easily run *Spark* jobs on EMR or your own Hadoop cluster

mrjob is licensed under the Apache License, Version 2.0.

To get started, install with `pip`:

```
pip install mrjob
```

5. The anatomy of a MapReduce script in mrjob

You can do a lot of things with mrjob, but to get started we will go over a typical MapReduce job. The best way to explain this is with an example. We will solve the same mean/variance problem so that we can focus on the subtleties of the framework. The code in the following listing serves the same purpose as the described codes for the mapper and the reducer. Open a text editor, create a new file called `mrMean.py`, and enter the code from the following listing.

```
from mrjob.job import MRJob

class MRmean(MRJob):
      def __init__(self, *args, **kwargs):
            super(MRmean, self).__init__(*args, **kwargs)
            self.inCount = 0
            self.inSum = 0
            self.inSqSum = 0

      def map(self, key, val):
            if False: yield
            inVal = float(val)
            self.inCount += 1
            self.inSum += inVal
            self.inSqSum += inVal*inVal
            def map_final(self):
            mn = self.inSum/self.inCount
            mnSq = self.inSqSum/self.inCount
            yield (1, [self.inCount, mn, mnSq])

      def reduce(self, key, packedValues):
            cumVal=0.0; cumSumSq=0.0; cumN=0.0
            for valArr in packedValues:
            nj = float(valArr[0])
            cumN += nj

            cumVal += nj*float(valArr[1])

            cumSumSq += nj*float(valArr[2])
            mean = cumVal/cumN
            var = (cumSumSq - 2*mean*cumVal + cumN*mean*mean)/cumN
            yield (mean, var)

      def steps(self):
            return ([self.mr(mapper=self.map, reducer=self.reduce,\
            mapper_final=self.map_final)])
if __name__ == '__main__':

   MRmean.run()
```

This code calculates a distributed mean and variance. The input text is broken up into multiple mappers and these calculate intermediate values, which are accumulated in the reducer to give a global mean and variance. You need to create a new class that inherits from the class MRjob. In this example, we have called that class MRmean. Your mapper and reducer are methods of this class. There is another method called steps(), which defines the steps taken. In the steps() method you tell mrjob the names of your mapper and reducer. If you don't specify anything, it will look for methods called mapper and reducer. Let talk about the behavior of mapper. The mapper acts like the inside of a for loop and will get called for every line of input. If you want to do something after you have received all the

lines of the input, you can do that in `mapper_final`. This may seem strange at first, but it's convenient in practice. You can share state between `mapper()` and `mapper_final)`. So in our example we accumulate the input values in `mapper()`, and when we have all the values we compute the mean and the mean of the squared values and send these out. Values are sent out of the mapper via the yield statement. Values are represented as key/value pairs. If you want to send out multiple values, a good idea is to pack them up in a list. Values will be sorted after the map step by the key. Hadoop has options for changing how things are sorted, but the default sort should work for most applications. Values with the same key value will be sent to the same reducer. You need to think through what you use for the key so that similar values will be collected together after the sort phase. We used the key 1 for all the mapper outputs because we want one reducer, and we want all of the mapper outputs to wind up at the same reducer.

The reducer in mrjob behaves differently than the mapper. At the reducer, the inputs are presented as iterable objects. To iterate over these, you need to use something like a for loop. You can't share state between the mapper or mapper_final and the reducer. The reason for this is that the Python script isn't kept alive from the map and reduce steps. If you want to communicate anything between the mapper and reducer, it should be done through key/value pairs. At the bottom of the reducer, we added a yield without a key because these values are destined for output. If they were going to another mapper, we would put in a key value.

To run the mapper only, enter the following commands in your Linux/DOS window, not in your Python shell. The file inputFile.txt is in the Ch15 code download.

```
%python mrMean.py --mapper < inputFile.txt
```

You should get an output like this:

```
1 [100, 0.50956970000000001, 0.34443931307935999]
```

To run the full function, remove the `--mapper` option.

```
%python mrMean.py < inputFile.txt
```

You'll get a lot of text describing the intermediate steps, and finally the output will be displayed to the screen:

.

.

.

```
streaming final output from c:\users\ayoub\appdata\local
\temp\mrMean.Peter.20110228.172656.279000\output\part-00000
0.50956970000000001 0.34443931307935999
removing tmp directory c:\users\ayoub\appdata\local\
temp\mrMean.20110228.172656.279000
```

To stream the valid output into a file, enter the following command:

```
%python mrMean.py < inputFile.txt > outFile.txt
```

6. **Tasks**
   a. Write a program in MapReduce using mrjob to compute the following formula

$$\hat{\alpha} = \frac{n}{\sum_{i=1}^{n} \ln x_i}$$

   (same formula in Assignment 1, Task I.B)

   b. Follow the instructions in the file `Hadoop_on_Colab.ipynb` to install Hadoop on Google Colab and run your code in a) on Google Colab.
   c. Generate random numbers (100, 200, …, 5000) in python, run your code in a) on Google Colab with Hadoop, compute computation time for each size, and draw a curve where the abscissa axis contains the sizes (100, 200, …,5000) and the ordinates axis contains the execution time.

7. **Submission**

You must submit the *pdf* file of your report. In fact, it must contain the responses in form of outputs for the tasks a), b), and c). In addition, you must submit all your implementations

.py and `Hadoop_on_Colab.ipynb` that contains the output of your code on Google Colab. ***Please, do not submit your assignment in .zip or .rar files.***

8. Useful external references

- NumPy documentation
- SciPy documentation
- mrjob v0.7.4 documentation
- Python programming language