COMP 330 Assignment #3

1 Description

The goal of this assignment is to implement two MapReduce programs in Java (using Apache Hadoop). Specifically, your MapReduce jobs will analyzing a data set consisting of New York City Taxi trip reports in the Year 2013.

2 Taxi Dataset

The data set itself is a set of simple text files. Each taxi trip report is a different line in a file. The attributes present on each line of the files are, in order:

attribute	description
`medallion`	an md5sum of the identifier of the taxi - vehicle bound (Taxi ID)
`hack_license`	an md5sum of the identifier for the taxi license (driver ID)
`vendor_id`	identifies the vendor
`pickup_datetime`	time when the passenger(s) were picked up
`payment_type`	the payment method -credit card or cash
`fare_amount`	fare amount in dollars
`surcharge`	surcharge in dollars
`mta_tax`	tax in dollars
`tip_amount`	tip in dollars
`tolls_amount`	bridge and tunnel tolls in dollars
`total_amount`	total paid amount in dollars

The data files are in comma separated values (CSV) format. I've uploaded the data to Amazon's S3, which is Amazon's storage service in the cloud. It is stored as twelve different files located at the location:

```
s3://chrisjermainebucket/comp330_A3/
```

This is about 20 GB of data in all. Important note: be aware that the above address may not copy-and-paste from this PDF correctly, as you may lose the underscore characters. This same problem may happen with the other commands below.

Using Amazon EMR, you can use those files as an input to a Hadoop MapReduce job in exactly the same way that you use files stored in HDFS as input to a Hadoop MapReduce job. For example, if you wanted to run a word count, you could just use the command:

```
[hadoop-10-182-96-202]$ hadoop jar myWordCount.jar -r 8 s3://chrisjermainebucket/comp330_A3/ outputfile
```

You can list the files from your master node using:

```
[hadoop-10-182-96-202]$ aws s3 ls s3://chrisjermainebucket/comp330_A3/
```

For testing and development, you can run your Hadoop MapReduce jobs on just one of the twelve files. For example, from your master node:

```
[hadoop-10-182-96-202]$ hadoop jar myWordCount.jar -r 8 s3://chrisjermainebucket/comp330_A3/trip_fare_1.csv
outputfile
```

A super-small subset of the first file (only 1000 lines) is available for download (see Canvas). If you want, you can use this file for testing and debugging by loading it into HDFS (just like you did for the first two labs) and then running your MapReduce program over it.

3 The Tasks

Analyzing log data like this is exactly what tools such as Spark and Hadoop are frequently used for. There are two separate programming tasks associated with this assignment; both involve analyzing this log data.

3.1 Task 1

Write a MapReduce program that checks all of the files and computes the total amount of revenue (total dollars) for each date present in the data set.

As you do this, be aware that this data (like all real data) can be quite noisy and dirty. The first line in the file might describe the schema, and so it doesn't have any valid data, just a bunch of text. You may or may not find lines that do not have enough entries on them, or where an entry is of the wrong type (for example, the dollars cannot be converted into a Java `double`). If you find a bad line, simply discard the line. Some people go crazy doing error checking, but all I am asking for is that you write robust code that won't crash with any of the data I have provided.

3.2 Task 2

Write a MapReduce program that computes 5 taxi drivers who had the most revenue in the data set.

Computing this is going to require you to write a sequence of at least a couple of MapReduce jobs. One job is going to need to compute the amount of money for each taxi driver, so the result will be a file containing a number of `(taxi, revenue)` pairs. One common question that comes up is: What is the correct format for the pairs? You can try to output a binary file that uses Java classes to store the pairs, but I would just change the reducer to produce `Text`, `DoubleWritable` pairs. These will be put into a text file, and then your second job will read the lines of text and re-parse each line into a `Text`, `DoubleWritable` pair.

The second job is going to need to compute the five most prosperous drivers using the first data set. There are a lot of ways to do this, but the most efficient (and a rather simple way) is to have a priority queue as a private member within your `Mapper` class. Every call to `map` just inserts the next taxi into the priority queue, organized by revenue. Whenever the queue has more than five entries in it, throw out the worst entries so that you only have five. But your `map` method *will not actually emit any data*. Then, your `Mapper` class will implement the `cleanup` method:

```
public void cleanup(Context context) throws IOException, InterruptedException ...
```

This is a method that any `Mapper` class is free to implement, that is automatically called when the mapper is about to be finished. Here, you'll simply emit the contents of the priority queue, writing them to the `Context`. In this way, your `Mapper` class will only write out the five best taxis processed by that mapper. Essentially, we are using the mapper as a filter. The reducer then computes the top five out of the top five found by every mapper. To do this, the key emitted by your `Mapper` class should just be the same value, like `1`. Then all of the taxis output by all mappers will go to that one reducer. That reducer can collect all of the results and emit the five best at the end.

Note that if you do this, you need to be careful and you cannot insert the Hadoop mutable data types into your queue, since they will be reused. Insert the standard Java mutable types instead. In the end, we are interested in computing a set of `(taxi, revenue)` pairs.

Another common problem on this task is blindly using your reducer class as a combiner (the word count implementation that I provided uses the `WordCountReducer` class as a combiner, which means that Hadoop may use the class during the shuffle to eagerly attempt to reduce the amount of data). Make sure that you understand the implication of doing this, or else just remove the line in the `main` method that

provides a combiner class.

4 Important Considerations

4.1 Machines to Use

One thing to be aware of is that you can choose virtually any configuration for your EMR cluster—you can choose different numbers of machines, and different configurations of those machines. And each is going to cost you differently! Pricing information is available at:

```
http://aws.amazon.com/elasticmapreduce/pricing/
```

Since this is real money, it makes sense to develop your code and run your jobs on a small fraction of the data. Once things are working, you'll then use the entire data set. We are going to ask you to run your Hadoop jobs over the "real" data using two machines with at least four cores each as workers. You'll have 8 cores total, so you'll want to run 8 reducers (if you use more cores, use more reducers).

Be very careful, and shut down your cluster as soon as you are done working. You can always create a new one easily when you begin your work again.

4.2 Monitoring Your Cluster

When you are running your jobs, you'll frequently have questions: what is my job doing? How many mappers are running? How many reducers? Fortunately, Hadoop has a very nice web interface that will allow you to monitor your job. Basically, Hadoop uses your master node as a web server and allows you to connect to display the status of the cluster.

Unfortunately, for security reasons, it's not as easy as simply going to a web address when you want to monitor your EMR Hadoop cluster. Amazon does not allow you to connect directly. Instead, you need to use an SSH tunnel. Fortunately, this should just take a few minutes to set up. See the instructions at:

```
http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-connect-ui-console.html
```

Once you set up the tunnel, you can view the status of your cluster by going to the web page:

```
http://ec2-107-21-183-54.compute-1.amazonaws.com:8088/
```

(you will replace the `ec2-107-21-183-54` with the address of your own master node).

5 Turnin

Create a single document that has results for all three tasks. For each task, copy and paste the result that your last MapReduce job wrote out, and include the textual output that you got from Hadoop when you ran the job.

Please zip up all of your code and your document (use .gz or .zip only, please!), or else attach each piece of code as well as your document to your submission individually.

6 Grading

Each problem is worth one half of the overall grade. If you get the right answer and your code is correct, you get all of the points. If you don't get the right answer or your code is not correct, you won't get all of the points; partial credit may be given at the discretion of the grader.