CS6240 – Parallel Data Processing in MapReduce

Basic MapReduce API

Ruinan Hu
hu.ru@husky.neu.edu
Northeastern University,
College of Engineering.

Sujith Narayan
rudrapatnaprakash.s@husky.neu.edu
Northeastern University,
College of Computer and Information Science

Karthik Chandranna
chandranna.k@husky.neu.edu
Northeastern University,
College of Computer and Information Science

Naveen Aswathanarayana
aswathanarayana.n@husky.neu.edu
Northeastern University,
College of Computer and Information Science

1 Introduction:

This project aims to provide a basic MapReduce framework for writing applications that can process Big Data in parallel on multiple nodes. The Master node instructs the Slave nodes to perform map and reduce tasks which are provided by the user. The Map phase takes a set of data and converts the required individual elements into key-value pairs. This is followed by the shuffle phase where the framework shuffles the output from the mappers and groups the equivalent keys together. The Reduce phase processes this intermediate data to aggregate them into key associated all the related values. Programs written using this API can be parallelized and executed on large clusters provided by Amazon known as AWS clusters. Many phases are involved in run-time and our system takes care of these phases, mainly creation of clusters, details of partitioning and distribution of input data to all clusters, program execution across the clusters and managing the inter-machine communication between master and slave nodes. Programmers can easily use this framework to build their own map and reduce functionality upon their data.

2 Implementation:

2.1 MR Job:

The MR Job class has methods that initialize the user's mapper and reducer classes along with the input and output folders. The methods use generic classnames and datatypes as arguments. These allow the framework to handle the different key and value types that the user provides.

On each node, including the master, the **run()** method kick starts the process. The IP details of all the nodes is read from S3 and these are stored in the Configuration object along with the number of instances in the cluster. Once the identity of the master is established, the master and slave nodes proceed to run their respective functions.

2.2 The Master Node:

The master node creates and distributes map and reduce jobs among the different slave nodes. It fetches the details of the input files from S3. This file list is sorted and a map of slave node number to a list of files is created. Sorting the files by size is a way to achieve equal load among the different slave nodes.

Each slave node is assigned a list of files and this list is further divided into four sub-lists of equal size. This is done to target the four cores of the EC2 machines. The four cores of the slave node get a Job object each. The job object contains a unique identifier and a list of files that are processed on that core. These jobs are called map jobs. The map jobs are preceded by a "map" header as a form of identifying the job. It is important to keep track of the number of map jobs that have been created.

Once the master has sent map jobs to all the slave nodes, it listens on a ServerSocket for responses. Upon receiving a "shuffled" response from all the map jobs on all the slave nodes, the master node proceeds to create reduce jobs.

The master node reads folder names (keys) from the intermediate shuffled dataset. Each node is assigned a list of keys. This list of keys is further divide into four sub-lists of equal size. The four cores of the slave node get a JobReduce object each. The JobReduce object contains a job id and a list of keys that are processed on that core. These jobs are called reduce jobs. The reduce jobs are preceded by a "reduce" header. As with map jobs, it is important to keep track of the number of reduce jobs.

Once the master has sent reduce jobs to all the slave nodes, it listens for responses. Upon receiving a "reduced" response from all the reduce jobs on all the slave nodes, the master node proceeds to kill all the slave nodes.

2.3 The Slave or Worker Nodes:

The Slave nodes listen to and act upon the commands from the master node. Each slave node has a ServerSocket listening to messages from the master. Upon arrival of a connection, a separate thread is spawned to handle the request. So, in effect, one thread handles one job. Depending on the header in the request, whether it is map or reduce, different functions are executed.

2.3.1 Map:

If the header contains “map”, the map job is executed. The list of files is retrieved from the received Job object. Each file is read line by line, and the map function of the user’s mapper class is executed. The line number is used as the key and the record as the value. A context object is also sent as an argument. The user’s map function writes to the local file system using the write methods in the context object. The naming of the files is important as it plays a huge role in the shuffling process. This output file is then uploaded to S3 by the use of a bash script and a “shuffled” message is sent to the master node.

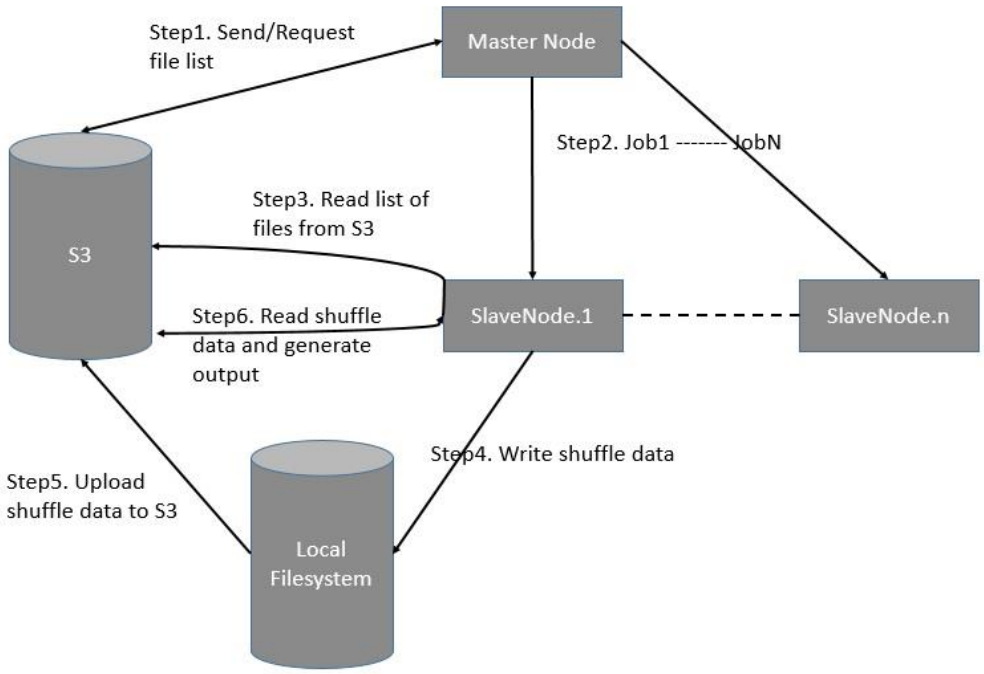
2.3.2 Shuffle:

The framework uses Local filesystem and S3 as our main data storage. The input and output folders which contains the dataset are configured to be in S3. During the shuffle phase, the framework uploads each key file from local file system to appropriate folder representing key on S3

2.3.3 Reduce:

If the header contains “reduce”, the reduce job is executed. The list of keys is retrieved from the received JobReduce object. These keys are folder names on S3. Each slave node accesses the assigned folder and reads each file. An ArrayList of values is then created and the user’s reduce function is invoked. Note that the key is of type Writable and the list of values is of type List<Writable>, so the user has to cast the writable objects to the specified datatype for the Reducer Input Key and Reducer Input Value. The context is also passed as an argument and this is used to write the output to the local file system. This output file is then uploaded to S3 by the use of a bash script and a “reduced” message is sent to the master node.

Figure 1: Architecture Diagram

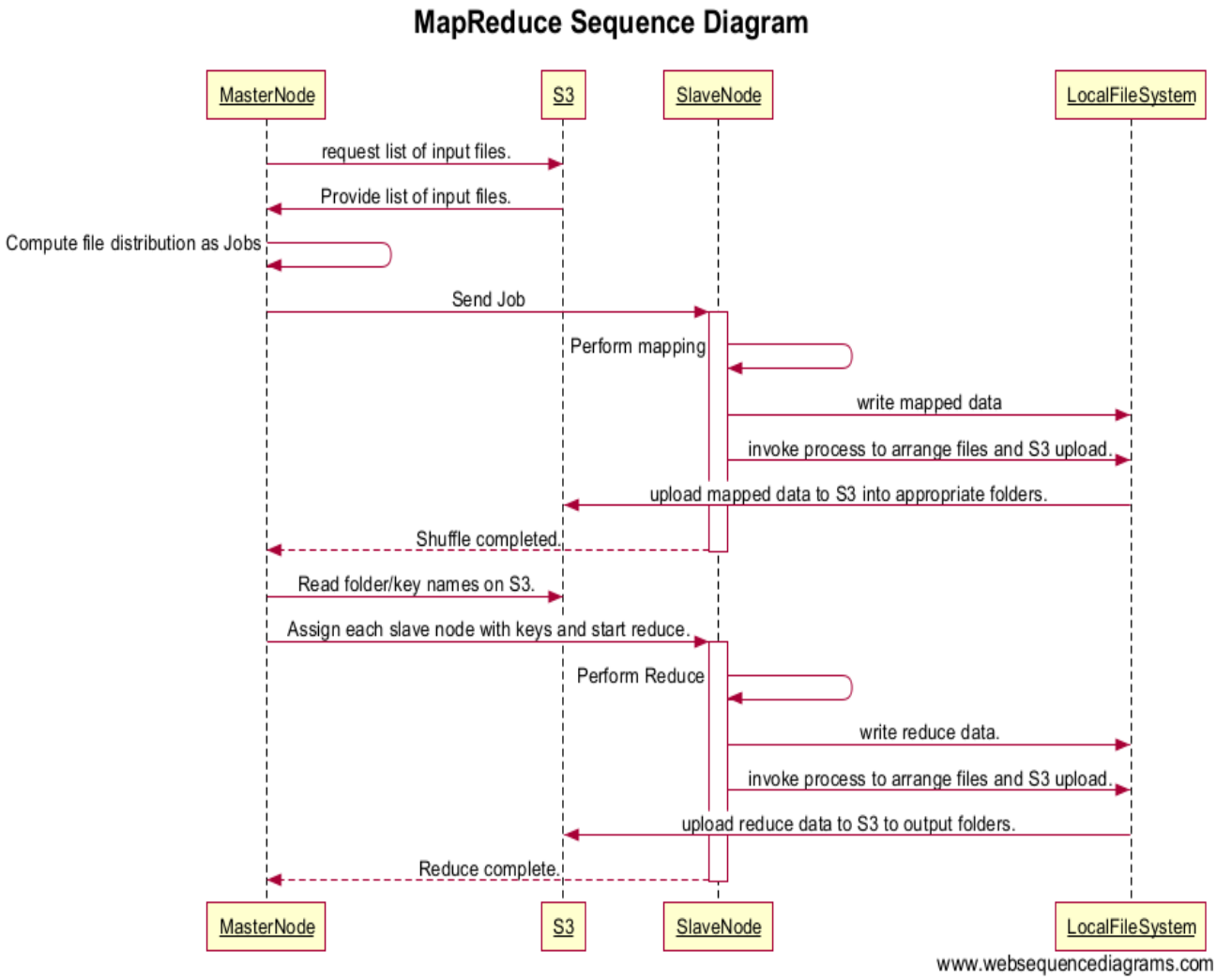


3 Network Model

The java.net.Socket class is used to communicate between the master and the slave nodes. It is important to note that the framework does not involve communication between the slave nodes, thereby avoiding a full-mesh network architecture. The IP details of all the nodes is stored on a file on S3. The master node uses this list to communicate with the slave nodes.

At the start of program execution, the slave nodes listen on a ServerSocket for commands from the master. When the master wants to communicate with the slave nodes, it creates a client socket by using the IP and port of the slave node. The ServerSocket is multithreaded. When a request is received, a separate thread is spawned to handle that request, and the ServerSocket continues listening to other requests. This networking design lets the framework target all four cores of the EC2 machines. The master sends four jobs to the four cores, by using four client sockets.

Once the master sends messages to the slave nodes, it creates a ServerSocket and listens for responses from the slave nodes. This ServerSocket is again multithreaded. This lets the framework listen to responses from all the slave nodes. The slave nodes communicate with the master by creating client sockets by using the master's IP and port.



4 Jobs/Tasks:

In order to efficiently use the computation power of nodes and to efficiently distribute the load among the nodes, we have defined Jobs. A job is a logical task that each core performs. Instead of sending a huge task to the slave node, the huge task is split into four sub-tasks and each sub-task is sent to a core in the node. The slave node creates a separate thread to handle each sub-task. This approach does a great deal to increase the computation time and mitigate the risk of running into out-of-memory issues.

5 File System:

The framework uses Local filesystem and S3 as our main data storage. The input and output folders which contains the dataset are configured to be in S3. Amazon SDK S3 objects are used to extract the information about the input folders. Master Node requests summary details of the input folder and computes and distributes the file. Files are distributed in such a way that each Slave node gets equal amount of data. Each Slave node read data file from S3. Since slave node contains the list of file it has to read and map to user specified format, we provide a way to read each file line by line. Mapped output is first written to local file system. Each job the slave node receives consists of list of files. Slave node write each folder for each job it receives to process during map stage. Each job folder contains file for each key the slave node is mapping.

The framework provides a simple bash script to shuffle the data from local file system to S3. S3 contains folder for each key. During shuffle phase framework upload each key file from local file system to appropriate folder representing key on S3. This way reduce job aggregate all the key file generated from various jobs on many nodes into one key folder on S3. The mapped data is primary written as binary using java ObjectStream. In order to write and read the file we are using java ObjectStream libraries casting into appropriate objects and then retrieving the elements of objects. During reduce phase each node is given a set of keys to aggregate. Master node creates jobs which contains the list of keys the node aggregates. The slave node looks up the folder with the key name and loads them to aggregate. The final output from the reducer depending upon the reduce logic provided by the user is written to S3 output folder.

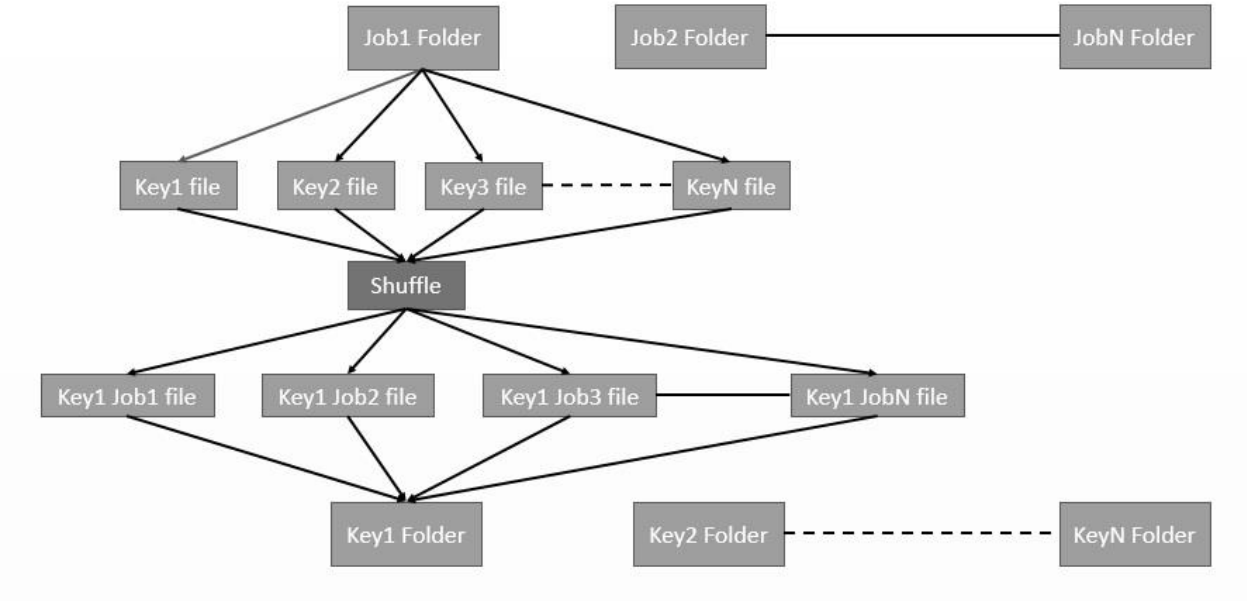
6) Configuration:

The Configuration class is used to set the input directory, output directory, mapper class, reducer class, datatypes of the map output key, map output value, reducer output key and reducer output value. The master and slave nodes access the required parameters through the Configuration class.

7) Context:

The Context class has methods to write output data from the mapper and reducer to the file system. A hashmap of key to opened stream is maintained. This hashmap lets us keep open streams to write output data, in the absence of which, new streams would have to be created for each write operation.

Figure 2: Shuffle Phase



6. Data Types

The framework currently supports the following datatypes: IntWritable, FloatWritable, DoubleWritable and Text. All the datatypes implement the Writable interface. The framework invokes the reduce function by passing a Writable as key and a List of Writables as value. The user, in the reduce function, must typecast the Writable to the datatype mentioned for MapperOutputKey and MapperOutputValue in the main function.

7 Assumptions:

Few assumptions considered while implementing the MapReduce framework were. First the storage service used must be S3. Amazon provides reliable and low cost commodity servers and appropriate infrastructure to on large scale servers. The MapReduce system assumes to run on the AWS services. Second, permissions to user S3 bucket should be provided. Write permissions has to be provided to users bucket on S3. Third all the nodes are considered to be running at all times until they complete the tasks. Finally when user implements the reduce method, cast Writable key and List<Writable> values to the MapperOutputKey and MapperOutputValue datatypes mentioned in the main method.

8 Example:

Main:

```
MrJob mrjob = new MrJob();
mrjob.setInputDirPath(<INPUT_DIR_PATH>);
mrjob.setOutputDirPath(<OUTPUT_DIR_PATH>);
mrjob.setMapperClass(AirlinePriceMapper.class);
mrjob.setReducerClass(AirlinePriceReducer.class);
mrjob.setMapperOutputKey(Text.class);
mrjob.setMapperOutputValue(DoubleWritable.class);
mrjob.setReducerOutputKey(Text.class);
mrjob.setReducerOutputValue(DoubleWritable.class);
mrjob.run();
```

Mapper:

```
class AirlinePriceMapper extends Mapper{
      void map(Writable key, Writable value, Context context)
      {
            AirlineParser FParser = new AirlineParser();
            if (!FParser.map(value.get())){
                  return;
            }
            context.writeMapper(new Text(FParser.Carrier),
            new DoubleWritable(FParser.Price));
      }
}
```

Reducer:

```
class AirlinePriceReducer extends Reducer{
 void reduce(Writable key, List<Writable> values, Context context)
 {
  int count =0;
  Double sum = 0.0D;
  ArrayList<Double> price = new ArrayList<Double>();
  for(Writable value : values) {
      count ++;
      DoubleWritable f = (DoubleWritable) value;
      sum+= f.getDouble();
      price.add(f.getDouble());
  }

  context.writeReducer((Text)key, new DoubleWritable(sum/count));

  }
}
```

9 Challenges:

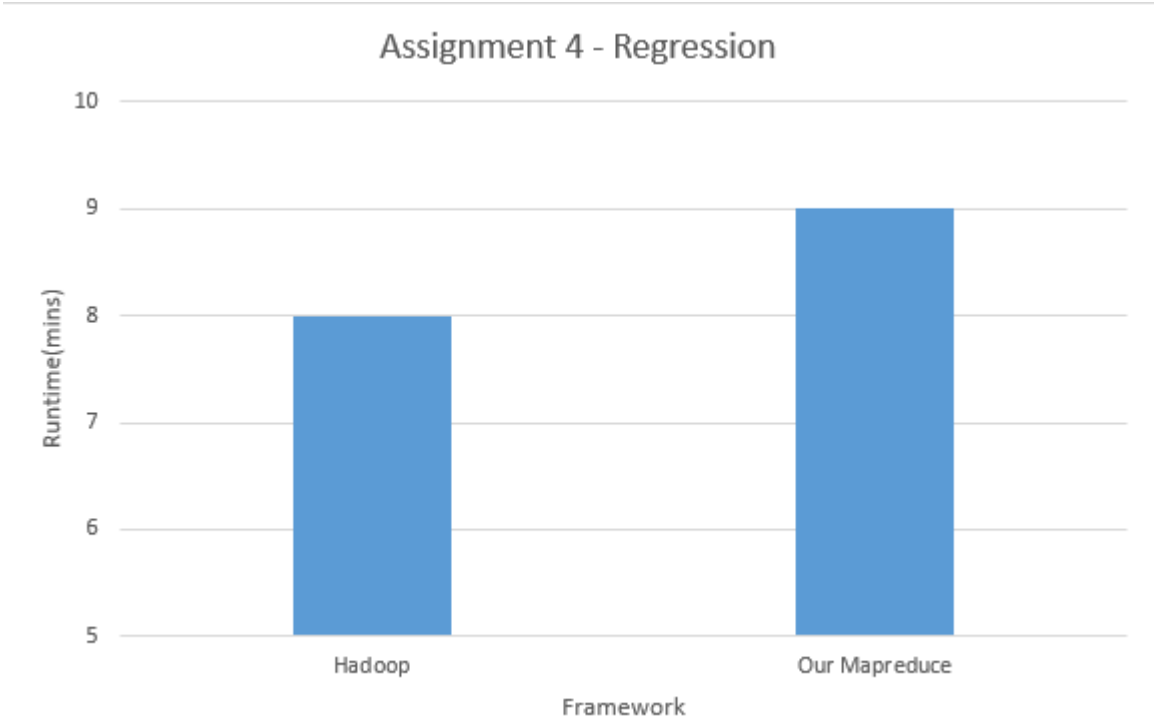
Main challenges faced during implementation of the MapReduce framework were. First reading and writing data to and from the object input and object output streams. Second achieving maximum performance by targeting all four core of the EC2 nodes available such as m3xlarge.

10 Performance:

A key step in achieving good performance was targeting all four cores of each EC2 instance. We used our framework to run assignments 3, 4 and 5. We found both the results and computation time to be consistent with their corresponding Hadoop implementations.

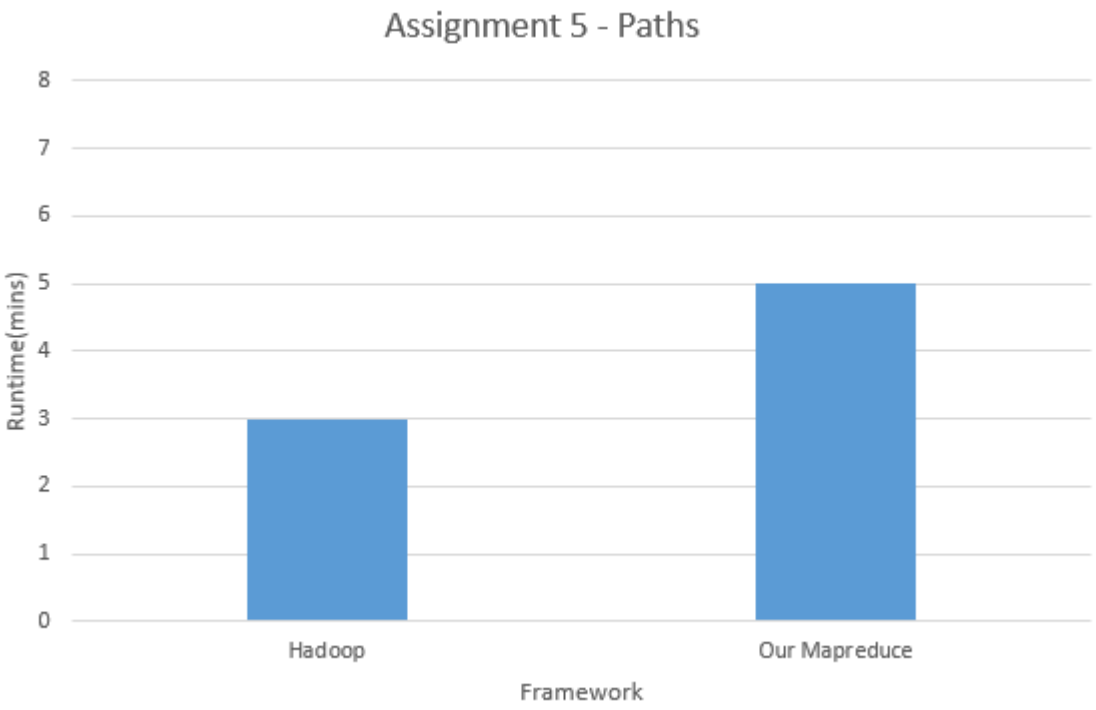
Configuration: 1 t2.micro EC2 instance (Master node), 10 m3.xLarge EC2 instances (Slave nodes)

1). Regression: Write a MapReduce job that ranks carriers and plots the evolution of prices of the least expensive carrier over time



1 t2.micro EC2 instance (Master), 10 m3.xLarge EC2 instances (Slave)
Full Dataset. Fig 3

2). Paths: Compute missed connections for all two-hop paths



1 t2.micro EC2 instance (Master), 10 m3.xLarge EC2 instances (Slave)
2 year Dataset. Fig 4

11 Future Enhancements

1) Log collection to enable better debugging
2) Fault tolerance – Error handling when nodes fail
3) A more efficient shuffling mechanism i.e. try not to upload all the intermediate data to S3
4) Customizable datatypes for keys and values
5) A web analytics tool to track the progress of the tasks

11 References

(1) http://stackoverflow.com/
(2) Hadoop, The Definitive Guide – Tom White
(3) Basic Hadoop framework http://hadoop.apache.org/docs/r2.6.0/api/
(4) MapReduce: Simplified Data Processing on Large Clusters
(5) https://www.gnu.org/software/bash/manual/bashref.html
(6) AWS command line manual.