Lab 1: Introduction to MapReduce & HDFS

Objective:

After completing this lab, you will be familiar with the MapReduce paradigm, implementing solutions through MapReduce in Hadoop via the Java API and running your solutions in a local JVM in Eclipse as well as on your sandbox. Finally, the lab will also introduce you to the HDFS command line tool utility.

Requirements:

1. Hortonworks Sandbox 2.6
2. Eclipse Neon or newer
3. Maven
4. Java 8

Due Date: Fri 9/15 11:55 PM

Things to Include in your Submission:

Your submission should be a zip file named username_lab1.zip that comprises of 3 folders:

1. MinTemperature - This folder will contain all the .java files needed to accomplish Task 1 and the .jar file that you built using Maven. The folder should also contain a text file that includes the command for executing your code on a test cluster.

2. FriendList - This folder will contain all the .java files needed to accomplish Task 2 and the .jar file that you build using Maven. The folder should also contain a text file that includes the command for executing your code on a test cluster.

3. Script - This folder will contain the script(s) you wrote to accomplish Task 3. The folder should also contain a text file that includes the command for executing your code on a test cluster.

4. A .pdf file that contains answers to the various questions you were asked during the lab.

Please upload the username_lab1.zip file to the Lab 1 Drop Box on Moodle

Rubric:

1. **Question 1 (10 Points):** Explain the MapReduce concept in your own words using a simple example. 5 Points for explanation, 5 points for example.

2. **Task 1 (30 Points)**
    a. Working Map class with correct logic - 10 Points
    b. Working Reduce Class with correct logic - 10 Points
    c. Working Main Driver class - 5 Points
    d. Following submission instructions - 5 Points
3. **Task 2 (50 Points)**
    a. Working Map class with correct logic - 20 Points
    b. Working Reduce Class with correct logic - 20 Points
    c. Working Main Driver class - 5 Points
    d. Following submission instructions - 5 Points
4. **Task 3 (15 Points)**
    a. Script can copy the files to HDFS - 5 Points
    b. Script can copy the files from HDFS - 5 Points
    c. The names of the input directory, target location on HDFS and target location on the local file system are specified as input arguments - 3 Points
    d. All submission instructions have been followed - 2 Points
5. **Question 2 (15 Points)**. Data flow during a read - 7.5 points. Data flow during a write 7.5 points
6. **Question 3 (5 Points).** Each command is worth a point

Feedback:

Please complete the anonymous feedback for the lab under Feedback → Lab Anonymous Feedback → Lab 1 Anonymous Feedback.

1. What is MapReduce?

MapReduce is a programming framework introduced by Google in the early 2000's. It is targeted at solving problems that have to work on huge datasets. Rather than devising an algorithm that works on the entire dataset, the mapreduce framework works on several chunks of the same dataset in parallel during the map phase and combines the results together during the reduce phase. MapReduce can take advantage of locality of data, processing data on or near the storage assets to decrease transmission of data.

"Map" step: The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. A worker node may do this again in turn, leading to a multi-level tree structure. The worker node processes the smaller problem, and passes the answer back to its master node.

"Reduce" step: The master node then collects the answers to all the sub-problems and combines them in some way to form the output – the answer to the problem it was originally trying to solve.

MapReduce allows for scalable processing of the map and reduction operations. If each of the map operations can be done independently, then all maps can be performed in parallel. In practice, the number of parallel maps is controlled by the number of input splits (think independent data chunks) and/or the number of mapping tasks (CPU cores dedicated to mapping) available in the cluster. Similarly, a set of reducers can operate on the output of the mapping tasks in parallel as long as the outputs of map operations that share the same key(think - the results that need to be merged together) are sent to the same reducer.

MapReduce can be inefficient on small datasets when compared to sequential algorithms. The killer application for the MapReduce paradigm is large datasets.

1.1. Programming model and execution framework

A Map/Reduce computation has two phases as mentioned above, a *map* phase and a *reduce* phase. The input to the computation is a data set of key/value pairs. The map and reduce functions in Hadoop MapReduce have the following general form:

```
map: (K1, V1) → list(K2, V2)
reduce: (K2, list(V2)) → list(K3, V3)
```

Table 1 MapReduce

In general, the map input key and value types (K1 and V1) are different from the map output types (K2 and V2). However, the reduce input must have the same types as the map output, although the reduce output types may be different again (K3 and V3).

In the map phase, the framework splits the input data set into a large number of fragments and assigns each fragment to a *map task*. The framework also distributes the many map tasks across the cluster of nodes on which it operates. Each map task consumes key/value pairs from its assigned fragment and produces a set of intermediate key/value pairs. For each input key/value pair *(K1,V1)*, the map task invokes a user defined *map function* that transmutes the input into a different key/value pair *(K2,V2)*. Following the map phase the framework sorts the intermediate data set by key and produces a list (K2, V2) tuples so that all the values associated with a particular key appear together. It also partitions the set of tuples into a number of fragments equal to the number of reduce tasks.

The Java API for the mapper resembles this general from:

```
public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
```

```
	public class Context extends MapContext<KEYIN, VALUEIN, KEYOUT,
VALUEOUT> {


		// ...

	}

	protected void map(KEYIN key, VALUEIN value,Context context) throws
IOException, InterruptedException {


		// ...

	}

}
```

Table 2 Mapper

In the reduce phase, each *reduce task* consumes the fragment of *(K2,list(V2))* tuples assigned to it. For each such tuple it invokes a user-defined *reduce function* that transmutes the tuple into an output key/value pair *(K3,V3)*. Once again, the framework distributes the many reduce tasks across the cluster of nodes and deals with shipping the appropriate fragment of intermediate data to each reduce task.

The Java API for the reducer resembles this general form:

```
public class Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

	public class Context extends ReducerContext<KEYIN, VALUEIN, KEYOUT,
VALUEOUT> {
			// ...

	}



	protected void reduce(KEYIN key, Iterable<VALUEIN> values, Context
context) throws IOException, InterruptedException {
			// ...

	}

}
```

Table 3 Reducer

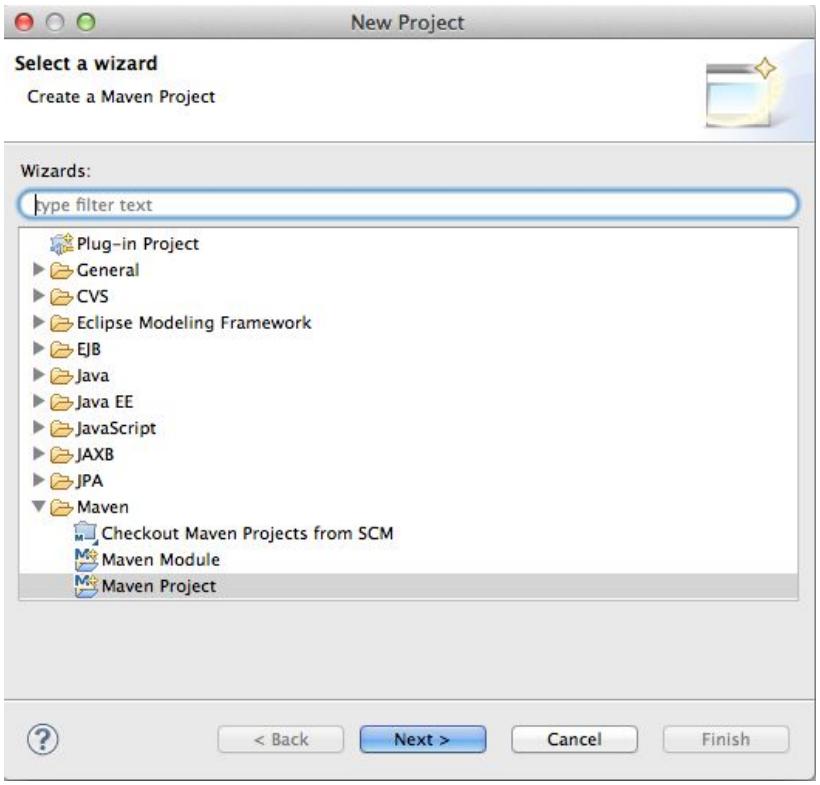
Tasks in each phase are executed in a fault-tolerant manner, if node(s) fail in the middle of a computation the tasks assigned to them are re-distributed among the remaining nodes. Having many map and reduce tasks enables good load balancing and allows failed tasks to be rerun with small runtime overhead.

To Do: Question 1 (10 Points): Explain the MapReduce concept in your own words using a simple example. 5 Points for explanation, 5 points for example.

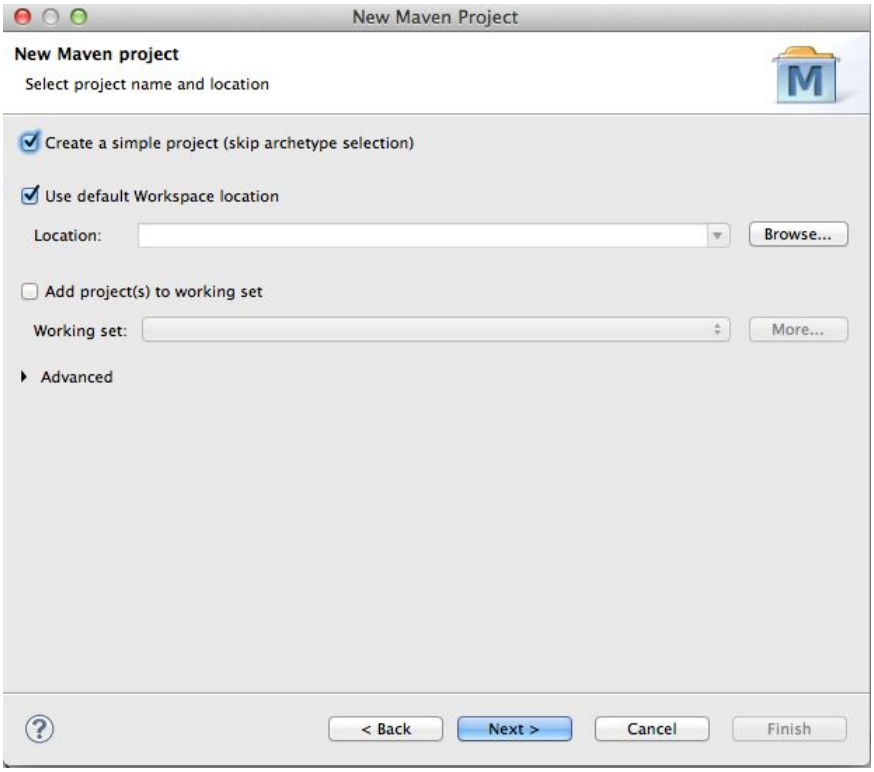
Creating a Project in Eclipse with Maven

These instructions assume that you have maven successfully installed and you installed Eclipse Neon.

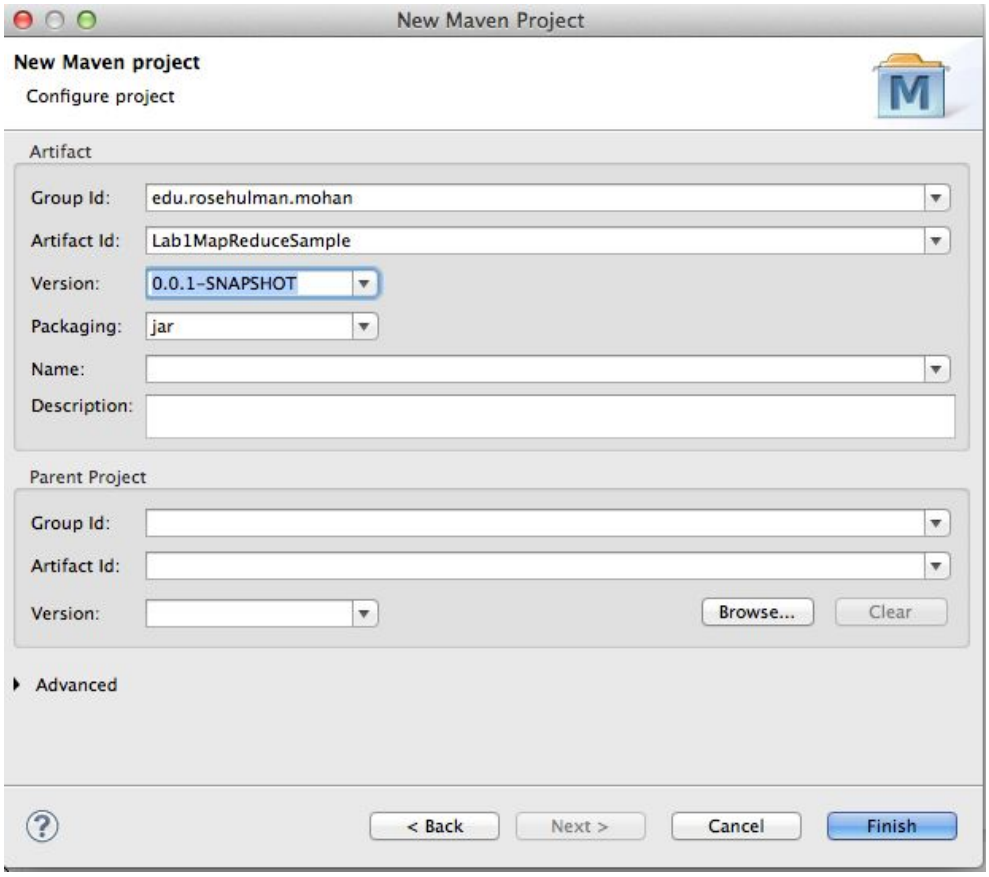
1. Create a new Maven Project in Eclipse. Click on File → Project → New Project
2. Choose a Maven Project in the dialog (as shown below)



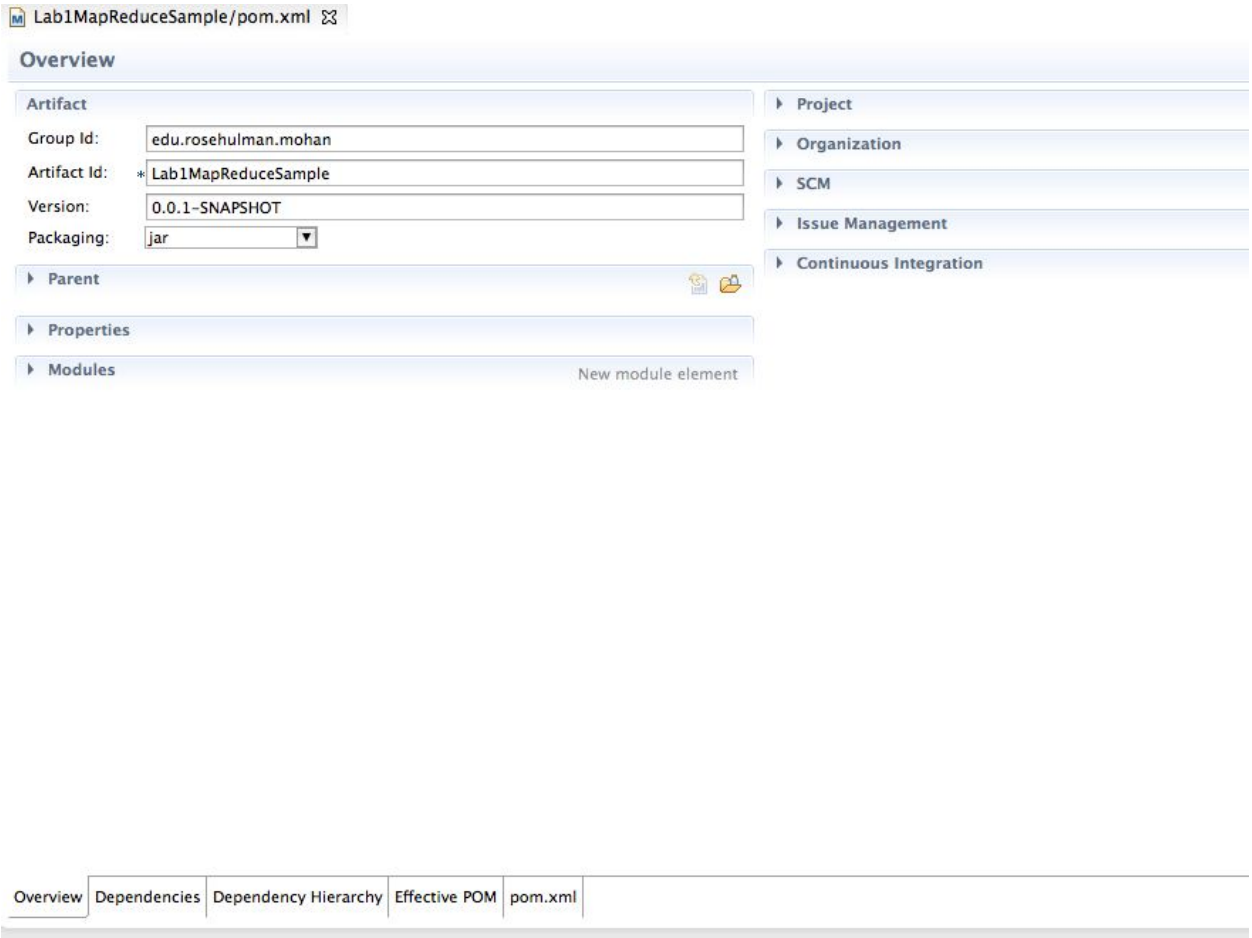
3. Choose Create a Simple Project(Skip Archetype Selection) as shown below



4. Specify groupID, artifactID as shown below.
    a. GroupID - edu.rosehulman.username - For instance, edu.rosehulman.mohan
    b. ArtifactID - Lab1MapReduceSample
    c. Packaging: jar
    d. Version: 0.0.1-SNAPSHOT



5. Click on Finish
6. Eclipse will create the project successfully.
7. Expand the project tree in the package explorer and select the pom.xml file
8. Eclipse will open up a nice GUI that looks like the screen shown below.



9. Click on the dependencies tab and fill the entries as shown below. We are instructing Maven, that during compile time, we will need access to the jars that are part of the org.apache.hadoop group and have the artifact id hadoop-client and version 2.8.1. In plain english, we are asking Maven to include all the hadoop-client libraries

10. Now you are ready to start development.

Task 1: Map Reduce Sample (30 Points)

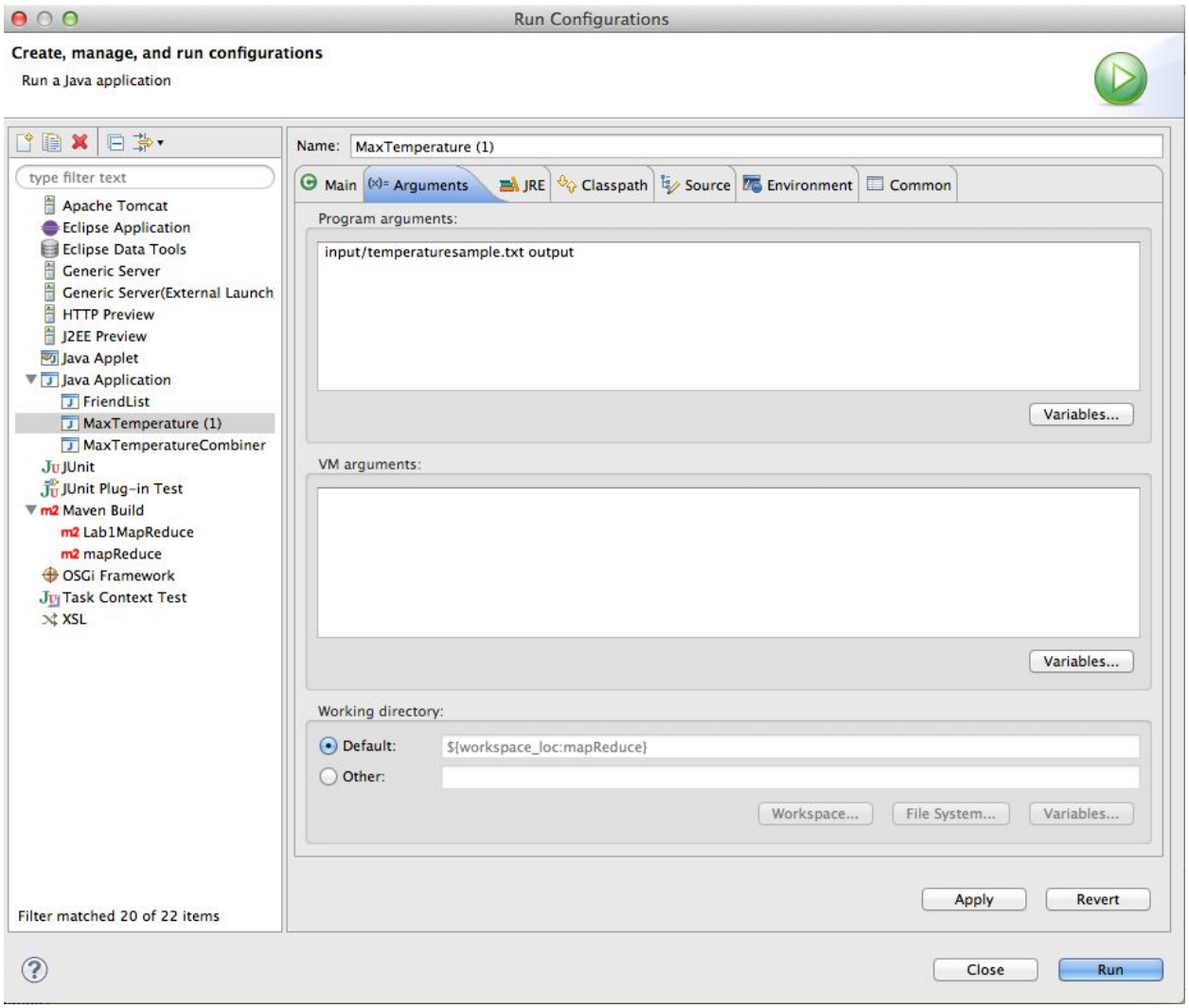
We will work on the dataset used during the in-class demonstration of MapReduce.

To Do 1: Create a program in MapReduce using Java to find the minimum temperature of each given year in the data set. Since we are running this program on our sandbox, please use the small dataset titled temperaturesample.txt as your input data set. Please note that you should be able to reuse majority of the code available in the book. Please refer to examples 2-3, 2-4, and 2-5. Please note that your java files should be in src/main/java in the package edu.rosehulman.username

How do you test your program from within Eclipse?[Only for Mac Users/Linux]

Once your code is complete and you are ready to test it, your first step should be test and see if it runs successfully within the Eclipse JVM. To do this, you need to create a run configuration as shown below:

1. Right click on the .java file that contains the main method and select Run as → Run Configurations.
2. In the window that opens up, use the tool tips to find the new run configuration option.
3. Choose the arguments tab and provide a space separated list of input arguments. For instance, if your input arguments were input/temperaturesample.txt and output, the window should look like this:



4. Click on Run
5. To run your program again, you need to right click on the .java file that contains the main method and choose Run as → Run Configurations. In the window, choose the run configuration you previously created and click Run.

If the code has been successfully tested on the eclipse, JVM, you are now ready to test it on the sandbox.

How do you test your program on the cluster(aka sandbox)?

We will be using Maven to build a .jar file that we can use to run the code.

1. Right click on the Pom.xml file and choose Run as → Maven Install. You will see Maven run through it is build life cycle in the console and if all things are successful, you will see a new .jar file has been created under the target sub-folder.
2. Open up a terminal window or use putty to scp the file to the sandbox. For instance, assuming you are already in the target directory, you can use the following command:
   scp -P 2222 *.jar root@127.0.0.1:
3. SSH to the sandbox
4. Once you are in the sandbox terminal, we will be using the hadoop command to launch the jar. The command is pretty straight forward. It follows the following pattern:

c

   yarn jar “the jar file you want to run” “the class file that contains the main method” “input arguments if any”

   For instance $yarn jar MapReduce.jar edu.rosehulman.mohan.MinTemperature /tmp/input/temperatureSample.txt /tmp/temperatureOutput
5. Please note that the input and output directories that are mentioned above are on HDFS and not on your file system.
6. You can use the file browser on hue to create any directories and files that are needed.

Turn in:

1. Create a folder called MinTemperature
2. This folder will contain all the .java files needed to accomplish Task 1 and the .jar file that you built using Maven The folder should also contain a text file that includes the command for executing your code on a test cluster.

Rubric:

1. Working Map class with correct logic - 10 Points
2. Working Reduce Class with correct logic - 10 Points
3. Working Main Driver class - 5 Points
4. Following submission instructions - 5 Points

Task 2 - Finding Common Friends

Facebook has a list of friends (note that friends are a bi-directional thing on Facebook. If I'm your friend, you're mine). They also have lots of disk space and they serve hundreds of millions of requests everyday. They've decided to pre-compute calculations when they can to reduce the processing time of requests. One common processing request is the "You and Joe have 230 friends in common" feature. When you visit someone's profile, you see a list of friends that you have in common. This list doesn't change frequently so it'd be wasteful to recalculate it every time you visited the profile (sure you could use a decent caching strategy, but then I wouldn't be able to continue writing about mapreduce for this problem). We're going to use mapreduce so that we can calculate everyone's common friends once a day and store those results. Later on it's just a quick lookup. We've got lots of disk, it's cheap.

Assume the friends are stored as Person,[List of Friends], our friends list is then:

A ,B, C,D
B,C,A
C,A,B

The above should be interpreted as A is friends with B,C,D and B is friends with C,A and so on.

Each line will be an argument to a mapper. For every friend in the list of friends, the mapper will output a key-value pair. The key will be a friend along with the person. The value will be the list of friends. The key will be sorted so that the friends are in order, causing all pairs of friends to go to the same reducer. This is hard to explain with text, so let's just do it and see if you can see the pattern. A Map task operating on the first line would produce something like this:
Key -> Value
(A , B) -> C,D
(A , C) -> B,D
(A , D) -> B C.

Similarly, the map task operating on the second line would produce

(A , B) -> C
(B , C) -> A

Now all the (A , B) keys will end up with the same reducer and that reducer will operate on something that looks like this:

Key → {Values}
(A, B) → {[C,D], [C]}

The reducer will now output that (A,B) have C as a common friend.

Turn in:

1. Create a folder called FriendList
2. This folder will contain all the .java files needed to accomplish Task 2 and the .jar file that you built using Maven The folder should also contain a text file that includes the command for executing your code on a test cluster.

Rubric:

1. Working Map class with correct logic - 20 Points
2. Working Reduce Class with correct logic - 20 Points
3. Working Main Driver class - 5 Points
4. Following submission instructions - 5 Points

Task 3 - Simple HDFS Script (15 Points)

As a hadoop developer, you will often find yourself in the position of writing a simple script to move files between the local filesystem (on your Sandbox) and HDFS. Write a simple script (in shell script, java or python) that copies the contents of an input directory in your local file system to a specified directory location on HDFS and then downloads the files from HDFS to a specified location on your local filesystem (on your Sandbox). The names of the input directory, target location on HDFS and target location on the local filesystem should be specified as input arguments.

Hint: You will find the HDFS shell commands put and get very useful.

Turn in:

1. Create a folder called Script.
2. This folder will contain the script(s) you wrote to accomplish Task 3.The folder should also contain a text file that includes the command for executing your code on a test cluster.

Rubric:

1. Script can copy the files to HDFS - 5 Points
2. Script can copy the files from HDFS - 5 Points
3. The names of the input directory, target location on HDFS and target location on the local file system are specified as input arguments - 3 Points
4. All submission instructions have been followed - 2 Points

To Do: Question 2 (15 Points): Explain in your own words, the data flow during a read operation on HDFS and data flow during a write operation on HDFS.

To Do: Question 3 (5 Points): Explain the following HDFS Shell commands with an example. Each command is worth a point.

1. setrep
2. chown
3. touchz
4. mv
5. **mkdir**

Things to Include in your Submission:

Your submission should be a zip file named username_lab1.zip that comprises of 3 folders:

1. MinTemperature - This folder will contain all the .java files needed to accomplish Task 1 and the .jar file that you built using Maven. The folder should also contain a text file that includes the command for executing your code on a test cluster.

2. FriendList - This folder will contain all the .java files needed to accomplish Task 2 and the .jar file that you build using Maven. The folder should also contain a text file that includes the command for executing your code on a test cluster.

3. Script - This folder will contain the script(s) you wrote to accomplish Task 3.The folder should also contain a text file that includes the command for executing your code on a test cluster.

4. A .pdf file that contains answers to the various questions you were asked during the lab.

Please upload the username_lab1.zip file to the Lab 1 Drop Box on Moodle

Feedback:

Please complete the anonymous feedback for the lab under Feedback → Lab Anonymous Feedback → Lab 1 Anonymous Feedback