Lab 3: Advanced MapReduce Concepts

Objective:

After completing this lab, you will be familiar with writing custom input formats, counters and joins using MapReduce.

Requirements:

1. Hortonworks Sandbox 2.3
2. Eclipse
3. Maven
4. M2E Plugin (A Maven Plugin for Eclipse)

Due Date: Friday 10/2 11:55 PM

Things to Include in your Submission:

Your submission should be a zip file named username_lab3.zip that comprises of 3 folders:

1. CustomInputFormat - This folder will contain all the .java files needed to accomplish Task 1 and 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.

2. ProjectHours - This folder will contain all the .java files needed to accomplish Task 3 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.

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

Please upload the username_lab3.zip file to the Lab 3 Drop Box on Moodle

Rubric:

1. **Question 1 (5 Points):** Explain the functionality of TextOutputFormat with an example. (2.5 Points for Explanation, 2.5 Points for Example).
2. **Question 2 (10 Points):** Explain the functionality of MultipleOutputFormat with an example. (5 points for explanation, 5 Points for Example).
3. **Task 1 and 2 (50 Points)**

    a. CustomWordCountInputFormat class with corresponding custom RecordReader. Both classes implemented the correct logic and produce the described Key Value pair (just one per file) - 30 Points.
    b. Custom Mapper class and No Reducer - 5 Points
    c. Custom Counters have been created and produced the correct output - 10 points
    d. Driver class prints valid number of MAP_INPUT_BYTES - 2.5 Points
    e. All submission instructions have been followed. 2.5 Points
4. **Task 3 (50 Points)**
    a. IntPair - A custom writable to handle a pair of ints with correct logic - 5 Points
    b. Custom Mapper class for dealing with Sprint and Work Details with correct tagging and correct logic. 8 *2 - 16 Points
    c. Custom Reducer Class that does the join - 7 Points
    d. Driver and Custom Comparator and Partitioner Class with correct logic - 20 Points
    e. All submission instructions have been followed. 2 Points
5. **Question 3 (10 Points)**. Explain the idea behind distributed cache with an example. 5 points for explanation, 5 points for example.

Feedback:

Please complete the anonymous feedback for the lab under Feedback → Lab Anonymous Feedback → Lab 3 Anonymous Feedback.

1. Custom Input Formats:

As we saw in class, an input split is a chunk of the input that is processed by a single map. Each map processes a single split. Each split is divided into records, and the map processes each record—a key-value pair—in turn. Splits and records are logical: there is nothing that requires them to be tied to files, for example, although in their most common incarnations, they are. Input splits are represented by the Java class InputSplit.

```
public abstract class InputSplit {

        public abstract long getLength() throws IOException, InterruptedException;
        public abstract String[] getLocations() throws IOException, InterruptedException;

}
```

An InputSplit has a length in bytes and a set of storage locations, which are just hostname strings. Notice that a split doesn't contain the input data; it is just a reference to the data. The storage locations are used by the MapReduce system to place map tasks as close to the split's data as possible, and the size is used to order the splits so that the largest get processed first, in an attempt to minimize the job runtime (this is an instance of a greedy approximation algorithm).

As a MapReduce application writer, you don't need to deal with InputSplits directly, as they are created by an InputFormat. An InputFormat is responsible for creating the input splits and dividing them into records. Before we see some concrete examples of InputFormat, let's briefly examine how it is used in MapReduce. Here's the interface:

```
public abstract class InputFormat<K, V> {

        public abstract List<InputSplit> getSplits(JobContext context) throws IOException,
        InterruptedException;
        public abstract RecordReader<K, V> createRecordReader(InputSplit
        split,TaskAttemptContext context) throws IOException, InterruptedException;

}
```

The client running the job calculates the splits for the job by calling getSplits(), then sends them to the jobtracker, which uses their storage locations to schedule map tasks that will process them on the tasktrackers. On a tasktracker, the map task passes the split to the createRecordReader() method on InputFormat to obtain a RecordReader for that split. A RecordReader is little more than an iterator over records, and the map task uses one to generate record key-value pairs, which it passes to the map function. We can see this by looking at the Mapper's run() method:

```
public void run(Context context) throws IOException, InterruptedException {

        setup(context);
        while (context.nextKeyValue()) {
                map(context.getCurrentKey(), context.getCurrentValue(), context);
                }
        cleanup(context);
}
```

After running setup(), the nextKeyValue() is called repeatedly on the Context (which delegates to the identically-named method on the RecordReader) to populate the key and value objects for the mapper. The key and value are retrieved from the Record Reader by way of the Context and are passed to the map() method for it to do its work. When the reader gets to the end of the stream, the nextKeyValue() method returns false, and the map task runs its cleanup() method and then completes.

Some applications don't want files to be split, as this allows a single mapper to process each input file in its entirety. For example, a simple way to check if all the records in a file are sorted is to go through the records in order, checking whether each record is not less than the preceding one. Implemented as a map task, this algorithm will work only if one map processes the whole file.

There are a couple of ways to ensure that an existing file is not split. The first (quick and dirty) way is to increase the minimum split size to be larger than the largest file in your system. Setting it to its maximum value, Long.MAX_VALUE, has this effect. The second is to subclass the concrete subclass of FileInputFormat that you want to use, to override the isSplitable() method to return false. For example, here's a non-splittable TextInputFormat:

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

public class NonSplittableTextInputFormat extends TextInputFormat {

        @Override
        protected boolean isSplitable(JobContext context, Path file) {
                return false;
        }

}
```

Question 1 (5 Points): Explain the functionality of TextOutputFormat with an example. (2.5 Points for Explanation, 2.5 Points for Example).

Question 2 (10 Points): Explain the functionality of MultipleOutputFormat with an example. (5 points for explanation, 5 Points for Example).

Task 1: Writing a Custom Input Format.

The input to this MapReduce program will be a directory that contains several input files. Your task is to write a mapreduce program that accepts a search word as input and outputs the number of times this specific search word appears in every file. For instance, consider that you have an input directory /tmp/input that contains files text1.txt, text2.txt and text3.txt. Let us assume that the contents of the files are as follows:

test1.txt: Mary had a little lamb. Mary did not have a little lamb.
test2.txt: Why the obsession with Mary?
test3.txt: I don't want to create sample input files anymore

Let us assume that the user enters Mary as the search argument when submitting the MapReduce job. The output of this program should be

```
/tmp/input/test1.txt 2
/tmp/input/test2.txt 1
/tmp/input/test3.txt 0
```

This is fairly straightforward to do in MapReduce. But we will explore the development of a custom input format to solve the above problem. Rather than using the default TextInputFormat, you will be required to develop your own custom input format -CustomWordCountInputFormat. This input format will have the following features:

1. The input file will not be splittable - That is rather than each line generating a key value pair, one key value pair must be generated for the entire file.
2. The key and the value generated for each file must be as follows. The key will the path to the file (inclusive of file name) and the value will be the number of times the search word appears in the file.

If implemented correctly, the above code will not need any reducers. The program will accept 3 input arguments in the following order

1. Path to the input directory
2. Path to the output directory
3. Search Word

Hint: You will find examples 8-2,8-3 and 8-4 in the book and the sample code from the class lecture on custom input formats to be useful with this activity.

Turn in:

1. You will be adding to the code you produced above in Task 2. Turn-in instructions are provided for both tasks together.

Rubric:

1. As mentioned above, Tasks 1 and 2 are related. Rubric for Task 1 is included along with Task 2's rubric.

2. Counters

Counters are a useful channel for gathering statistics about the job: for quality control or for application-level statistics. They are also useful for problem diagnosis. If you are tempted to put a log message into your map or reduce task, it is often better to see whether you can use a counter instead to record that a particular condition occurred. In addition to counter values being much easier to retrieve than log output for large distributed jobs, you get a record of the number of times that condition occurred, which is more work to obtain from a set of log files. Hadoop maintains some built-in counters for every job, and these report various metrics. For example, there are counters for the number of bytes and records processed, which allows you to confirm that the expected amount of input was consumed and the expected amount of output was produced. Counters are divided into groups, and there are several groups for the built-in counters. A comprehensive listing of counters is available in chapter 8.

MapReduce allows user code to define a set of counters, which are then incremented as desired in the mapper or reducer. Counters are defined by a Java enum, which serves to group related counters. A job may define an arbitrary number of enums, each with an arbitrary number of fields. The name of the enum is the group name, and the enum's fields are the counter names. Counters are global: the MapReduce framework aggregates them across all maps and reduces to produce a grand total at the end of the job.

Task 2 - Writing Custom Counters

You will supplement the mapreduce program you wrote in the previous task with custom counters. You will define 3 user counters that belong to the counter group WordCount:

1. EqualToTwo
2. LessThanTwo
3. GreaterThanTwo

The counters will track the following statistic. They will track the number of times the search word appeared in a file and increment the corresponding counter. For instance, if a file has the search word "Mary" twice, the counter EqualToTwo should be incremented. If a file has the search word "Mary" thrice, the counter GreaterThanTwo should be incremented and so on.

In addition to the above 3 counters, your code should also print the number of bytes of uncompressed output produced by all the maps in the job. You will find the pre-existing counter MAP_OUTPUT_BYTES useful here.

Hint: You will find examples 8-1 in the book and the sample code from the class lecture on custom counters to be useful with this activity.

Turn in:

1. Create a folder called CustomInputFormat.
2. This folder will contain all the .java files needed to accomplish Task 1 and 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.

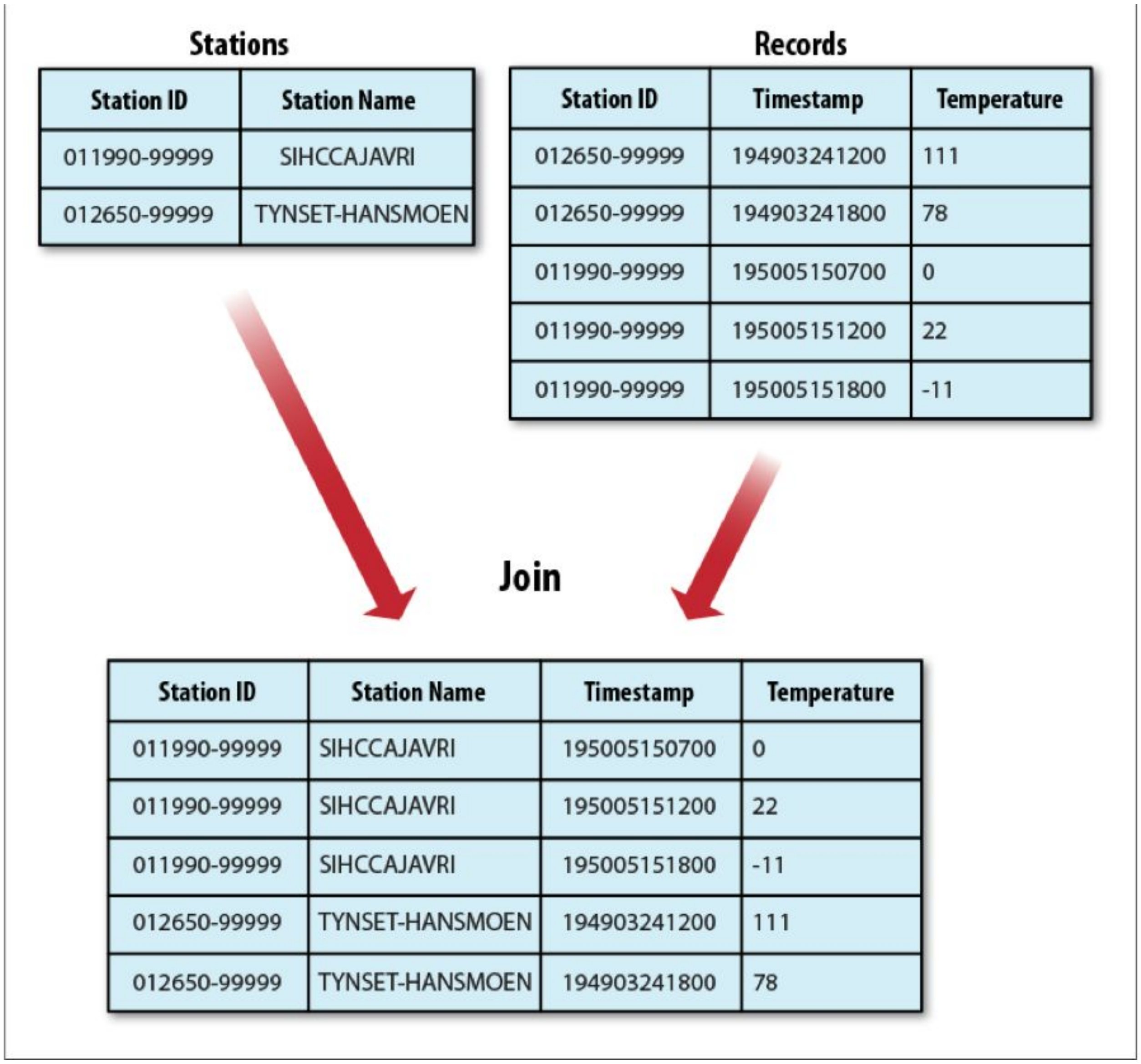
Rubric:

a. CustomWordCountInputFormat class with corresponding custom RecordReader. Both classes implemented the correct logic and produce the described Key Value pair (just one per file) - 30 Points.
b. Custom Mapper class and No Reducer - 5 Points
c. Custom Counters have been create and produce the correct output - 10 points
d. Driver class prints valid number of MAP_OUTPUT_BYTES - 2.5 Points
e. All submission instructions have been followed. 2.5 Points

3. Joins

MapReduce can perform joins between large datasets, but writing the code to do joins from scratch is fairly involved. Typically, rather than writing MapReduce programs, you might consider using a higher-level framework such as Pig, Hive, or Cascading, in which join operations are a core part of the implementation. However, it is still useful to know how joins can be implemented with a MapReduce program from scratch.

For example, Let's briefly consider the problem from the book. We have two datasets—for example, the weather stations database and the weather records—and we want to reconcile the two. Let's say we want to see each station's history, with the station's metadata inlined in each output row. If the join is performed by the mapper, we have a map-side join, if the join is performed by a reducer, we have a reduce side join. The desired result of the join is shown in the figure below.



Station ID	Station Name
011990-99999	SIHCCAJAVRI
012650-99999	TYNSET-HANSMOEN

Station ID	Timestamp	Temperature
012650-99999	194903241200	111
012650-99999	194903241800	78
011990-99999	195005150700	0
011990-99999	195005151200	22
011990-99999	195005151800	-11

Station ID	Station Name	Timestamp	Temperature
011990-99999	SIHCCAJAVRI	195005150700	0
011990-99999	SIHCCAJAVRI	195005151200	22
011990-99999	SIHCCAJAVRI	195005151800	-11
012650-99999	TYNSET-HANSMOEN	194903241200	111
012650-99999	TYNSET-HANSMOEN	194903241800	78

Reduce - Side Joins

A reduce-side join is more general than a map-side join, in that the input datasets don't have to be structured in any particular way, but it is less efficient because both datasets have to go through the MapReduce shuffle. The basic idea is that the mapper tags each record with its source and uses the join key as the map output key, so that the records with the same key are brought together in the reducer. We use several ingredients to make this work in practice:

Multiple inputs

The input sources for the datasets generally have different formats, so it is very convenient to use the MultipleInputs class (see “Multiple Inputs” on page 250 in the book) to separate the logic for parsing and tagging each source.

Secondary sort

As described, the reducer will see the records from both sources that have the same key, but they are not guaranteed to be in any particular order. However, to perform the join, it is important to have the data from one source before another. For the weather data join, the station record must be the first of the values seen for each key, so the reducer can fill in the weather records with the station name and emit them straightaway. Of course, it would be possible to receive the records in any order if we buffered them in memory, but this should be avoided because the number of records in any group may be very large and exceed the amount of memory available to the reducer.

The MapReduce framework sorts the records by key before they reach the reducers. For any particular key, however, the values are *not* sorted. The order in which the values appear is not even stable from one run to the next, because they come from different map tasks, which may finish at different times from run to run. Generally speaking, most MapReduce programs are written so as not to depend on the order in which the
values appear to the reduce function. However, it is possible to impose an order on the values by sorting and grouping the keys in a particular way.

To illustrate the idea, consider the MapReduce program for calculating the maximum temperature for each year. If we arranged for the values (temperatures) to be sorted in descending order, we wouldn’t have to iterate through them to find the maximum; instead, we could take the first for each year and ignore the rest. (This approach isn’t the most efficient way to solve this particular problem, but it illustrates how secondary sort works in general.)

To achieve this, we change our keys to be composite: a combination of year and temperature. We want the sort order for keys to be by year (ascending) and then by temperature (descending):

1900 35°C
1900 34°C

1900 34°C ...

1901 36°C
1901 35°C

If all we did was change the key, this wouldn't help, because then records for the same year would have different keys and therefore would not (in general) go to the same reducer. For example, (1900, 35°C) and (1900, 34°C) could go to different reducers. By setting a partitioner to partition by the year part of the key, we can guarantee that records for the same year go to the same reducer. This still isn't enough to achieve our goal, however. A partitioner ensures only that one reducer receives all the records for a year; it doesn't change the fact that the reducer groups by key within the partition:

The final piece of the puzzle is the setting to control the grouping. If we group values in the reducer by the year part of the key, we will see all the records for the same year in one reduce group. And because they are sorted by temperature in descending order, the first is the maximum temperature:

To summarize, there is a recipe here to get the effect of sorting by value:

- Make the key a composite of the natural key and the natural value.
- The sort comparator should order by the composite key, that is, the natural key *and* natural value.
- The partitioner and grouping comparator for the composite key should consider only the natural key for partitioning and grouping.

In our fictional join example, we can use the notion of secondary sort described above. To tag each record, we use TextPair from Chapter 4 for the keys (to store the station ID) and the tag. The only requirement for the tag values is that they sort in such a way that the station records come before the weather records. This can be achieved by tagging station records as 0 and weather records as 1.

Full details of the above along with the corresponding code is available in Chapter 9.

Task 3 - Custom Reduce Side Join

Let us assume you have two input files titled workdetails.txt and sprintdetails.txt. Sprint details contains the following information (sprint number and sprint name)

```
1,Databases
2,Hadoop
3,Graphics
4,CompArch
5,Compilers
```

workdetails.txt contains the following information. Name of a person and sprint that they worked on

```
Sriram,Mohan,1
Sriram,Mohan,2
Cary,Laxer,3
Micah,Taylor,3
Micah,Taylor,4
Claude,Anderson,5
Curt,Cliftion,5
Matt,Boutell,1
Chandan,Rupakheti,5
Sriram,Mohan,4
```

Please implement a reduce side join that accepts the following input arguments

1. Path to work details
2. Path to sprint details
3. Path of output directory

and produces the following output.

```
1       Databases     Matt Boutell
1       Databases     Sriram Mohan
2       Hadoop        Sriram Mohan
3       Graphics      Micah Taylor
3       Graphics      Cary Laxer
4       CompArch      Sriram Mohan
4       CompArch      Micah Taylor
5       Compilers     Chandan Rupakheti
5       Compilers     Curt Cliftion
```

5 Compilers Claude Anderson

Hint: You will need to use MultipleInputFormats and Secondary Sort. My solution has the following class

1. JoinSprintMapper - Custom Mapper to handle sprint details file that produces a custom IntPair as the key and the name of the sprint as one Text Value. The IntPair contains the project number and a tag value to ensure that the Sprint name is loaded before the names of the people in the reducer
2. JoinWorkMapper - Custom Mapper to handle work details file that produces a custom IntPair as the key and the name of the person as one Text Value. The IntPair contains the project number and a tag value
3. IntPair - A custom writable to handle a pair of ints.
4. JoinReducer - The reducer knows that it will receive the station record first, so it extracts its name from the value and writes it out as a part of every output record
5. Driver and other custom classes: Tying the job together is the driver class. The essential point is that we partition and group on the first part of the key, the project ID which we do with a custom Partitioner (KeyPartitioner) and a custom group comparator, First Comparator which again groups reduer records only by the first part of the key.

Turn in:

1. Create a folder called Project Hours.
2. This folder will contain all the .java files needed to accomplish Task 3 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:

a. IntPair - A custom writable to handle a pair of ints with correct logic - 5 Points
b. Custom Mapper class for dealing with Sprint and Work Details with correct tagging and correct logic. 8 *2 - 16 Points
c. Custom Reducer Class that does the join - 7 Points
d. Driver and Custom Comparator and Partitioner Class with correct logic - 20 Points
e. All submission instructions have been followed. 2 Points

Question 3 (10 Points). Explain the idea behind distributed cache with an example. 5 points for explanation, 5 points for example.

Things to Include in your Submission:

Your submission should be a zip file named username_lab3.zip that comprises of 3 folders:

1. CustomInputFormat - This folder will contain all the .java files needed to accomplish Task 1 and 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.

2. ProjectHours - This folder will contain all the .java files needed to accomplish Task 3 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.

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

Please upload the username_lab3.zip file to the Lab 3 Drop Box on Moodle

Feedback:

Please complete the anonymous feedback for the lab under Feedback → Lab Anonymous Feedback → Lab 3 Anonymous Feedback.