CIS 4930 Introduction to Hadoop and Big Data
Lab Homework 6
Julie Reyes | U76631122

Project A: Writing a Partitioner

I. Introduction

The intent of this project is to write a MapReduce job will invoke multiple Reducers that are specified by a custom partitioner. The custom partitioner will take keys from the Mapper output and determine which specific Reducer the key value pairs are sent to. This project deviates from the normal execution of a MapReduce job as the values sent to the Reducers are not based on the intermediate key from the Mapper. More specifically, the partitioner in this project determines the Reducer by the month found in a log file for each line of input. For this project the file *weblogs* will be analysed and the unique IP address hits will be summed per month.

II. Driver Description

The diver for this project is named *ProcessLogs.java*. The job name is defined by the driver as *"Process Logs"*. The Mapper class is set to *LogMonthMapper*. The Reducer class is set to *countReducer.* The Mapper output key and value are set to be of type *Text*. The final output key is set to be of type *Text* and the value to be of type *IntWritable.* The number of Reducers is set in the driver by setting the *sumNumReduceTasks* to 12, which corresponds with the total number of months. The driver also set the partitioner class as *MonthPartitioner*.

III. Mapper Description

The Mapper *LogMonthMapper* extends the Mapper class and takes in four parameters, *LongWritable, Text, Text, Text.* The Mapper is overridden by the *map()* function, which takes in three parameters *LongWritable* for the key, *Text* for the value, and *Context.* The input line is extracted by the *toString()* function and converted a String variable named *line*. The IP address length is determined by getting the index of the first space in the *line*. Next, the Mapper employs the Pattern and Matcher objects to extract a specific section from the input *line*, through the use of a regular expression. In this Mapper, *"\\/(\\D*?)\\/"* was used as regular expression to extract the month from the *line* string best on the given format. Lastly, the map function emits the *Text* key as the substring of *line* that corresponds to the IP address, and emits the *Text* value as the string extracted from the *line* that contains the month. Note that in the code a substring is generated to exclude the "/" found by the regular expression for the month value.

IV. Reducer Description

The Reducer for this project is named *CountReducer* and accepts four parameters *Text, Text, Text,* and *IntWritable.* The Reducer is overridden by the reduce function which takes in three parameters. *Text* as the key, *Iterable<Text>* as an iterable set of values,and *Context.* The variable int *count* keeps a running total of all of the hits on the same IP address. The final output is the key, which corresponds to the IP address, and the value *count.* The key value pairs are written by the *write()* function of the Context object.

V. Partitioner Description

The class that extends the partitioner is named *MonthPartitioner*. Next, a configuration named *configurations* and a HashMap *months* are initialized. In the *setConfig*() method the parameter *configuration* is passed to the and the HashMap is populated. The HashMap maps the three letter word of a month to the number of month it corresponds to. Next, the *getConf()* method in implemented and returns *configuration.* Lastly, the *getPartition()* method is implemented. In this method four parameters are passed, the *Text* key, *Text* value, and the *int* numReduceTasks. The number of month is represented by the integer number that corresponds to that month. For this to integer to be returned, the hashMap *months* is accessed with the *months.get(value.toString())* and then cast to an integer and is the return value of the *getPartition*() method in the partitioner. This integer return number determines the Reducer to which the data is sent.

VI. Data Flow Description

The data flow for this project starts at the input file, in this case the *weblogs* file. This file is initially on the hadoop file system as it was uploaded to the cluster in a previous lab. To submit the job to the cluster the the driver *ProcessLogs,* is run on the local machine and then submits the MapReduce job to the hadoop cluster. The *weblogs* HDFS file on the cluster is read by the record reader to be used by the Mapper. In the Mapper, the data is manipulated based in a determined manner by implemented java code. Next, the data path is determined by the *Month Partitioner*, which is a custom partitioner. This partitioner sends the data to specific reducers based on the value of the month in the log file, which means all IP address hits in the month of January will all go to the same reducer 0. Next, once the data is in the reducer all the same IP address hits are counted up. Lastly, the IP address and hits key value pairs from each month are output by the 12 separate Reducers that mirror each month.

VII. Compile Procedure Description

The four files *CountReducer, LogMonthMapper, MonthPartitioner,* and *ProcessLogs* where exported from the stubs package on eclipse. When exporting a jar file named *partitioner.jar* was created. Next to submit the job the following was typed in the command line:

$ *hadoop jar partitioner.jar stubs.ProcessLogs weblog IPMonth*

VIII. Test data and Results

Initially the MapReduce job was tested on a smaller subset of the data to ensure the job was working properly. The following are results obtained for the trail run:

part - r - 00007
….
10.130.195.163 21
10.150.176.47 12
10.211.47.159 2860
10.216.113.172 828
10.82.30.199 26
...

These values represent the hits for the month of August. The number 7 in the file

name part - r - 00007, corresponds to the month which is 1 less the convention as the months in the part names start at 00000. Once the job was confirmed to be working properly, the job was submitted to run on the *weblogs* file, which is much larger dataset. This is a small sample of the results in January:

```
part - r - 00000
...
10.1.171.161    21
10.1.181.142    415
10.1.183.134    1
10.1.186.241    2
10.1.187.27     15
10.1.190.237    18
...
```

Project B: Implementing a Custom WritableComparable

I. Introduction

The goal of this project is to count the number of times a lastname and firstname pair occur in a data set. This is accomplished by writing a custom WritableComparable type which will hold two strings, the first and last name strings. Since, WritableComparable defines a sorted order for the keys, the program will count the number of times the key occurs in the dataset.

II. WritableComparable Description

In this project, a custom WritableComparable is defined by the *StringPairWritable.java* program. The *StringPairWritable* implements *WriteComparable* and is defined as a <*StringPairWritable*> object. Two strings named *right* and *left* are initialized. First, an empty constructor is defined for serialization. Next, a constructor with two String objects parameters is created. In this constructor named *StringPairWritable*, the *left* and *right* String objects are defined by the input parameters. Next, the *write()* method, which serializes the fields of the object to out, is defined by beth the *left* and *right* parameters with the *out.write()* function. The *readsFields()* method in the *StringPairWritable* deserializes the fields of this object from *in*. This method accepts one parameter the *dataInput in.* The left and right strings are defined by the *this.left = in.readUTF()* and *this.right = in.readUTF()* inside the method *readFields()*. Next, the *compareTo()* method compares the *WriteComparable* object to another of the same type by method *compareTo()*. This method accepts one parameter of type *StringPairWritable* and returns an *int* value. This method is defined by comparing the left parameter of *this StringPairWritable* object to the *left* parameter of the *StringPairWritable* object passed to the method. If they are equal then the integer return is the comparison of *this.right.compareTo(other.right).* If they *left* parameters are not equal then the integer value of the comparaison *this.left.compareTo(other.left)* is returned. Next, a custom method *toString()* is defined. This method has no parameters and returns the two strings in the *StringPairWritable* object inside parentheses and separated by a comma. This will be the format of the key when the MapReduce job is outputted. Next, the *equals()* method is defined. The *equals()* method is passed one parameter of type *Object.* This method compares the two *StringPairWritable* objects to measure their equality and returns a boolean value. The *hashcode()* method implements a hash code

for a *StringPairWritable* object. The *hashCode()* uses the variables *final int prime = 31* and *int result = 1*, to return the integer hashcode for each parameter *left* and *right* of a *StringPairWritable* object by using the formula *result = prime * result + ((left == null) ? 0 : left.hashCode());* and *result = prime * result + ((right == null) ? 0 : right.hashCode());*

III. Driver Description

The *StringPairTestDriver* driver is responsible for configuring and implementing the MapReduce job named *"Custom Writable Comparable"*. This driver then specifies the input directory from which to read and an output to write to by the command line inputs given by the user. The driver sets the Mapper and Reducer class as *LongSumReducer* and *StringPairMapper* respectively . The Reducer *LongSumReducer* for this project is imported from the Hadoop library class.
Next, the driver defines the intermediate key and values from the Mapper as type *StringPairWritable* and *LongWritable* for this project. This driver also has ToolRunner functionality defined in the *main()* method.

III. Mapper Description

The *StringPairMapper* extends the Mapper class and accepts four parameters *LongWritable*, *Text*, *StringPairWritable*, and *LongWritable*. The *map()* method overrides the Mapper and takes in three arguments as *LongWritable key*, *Text value*, and *Context context*. In the *map()* method, the input line is split into words. The *StringPairWritable* object is the key constructed from the first two strings in the line input. The value is set by the Mapper to be of type *LongWritable* and is initialized to *one*, of value 1. The *Context* object emits the *StringPairWritable* object as the key, and the *LongWritable* one as the value.

IV. Data Flow Description

The data flow for this MapReduce job starts with the local input file named *nameyeartestdata* provided by the course. This file was then uploaded to the hadoop cluster through the *put* command. Once the file was on the cluster, the diver, *StringPairMapper,* running locally submits the MapReduce job to the cluster. The data input splits are sent to the Mapper by the record reader. The Mapper then converts the provided line of input by the determined manner explained in the Mapper Description section of this report. Once the *StringPairWritable* key and value *LongWritable 1* are generated, the data is sent to the Reducer. In the Reducer, the total count of distinct last name, first name pairs are emitted to the command line defined output directory.

V. Test data and Results

In the lab pdf for this project the file to test the MapReduce job was provided at the path, *~/training_materials/developer/data/nameyeartestdata*. The lab description for this project suggested two methods for testing the MapReduce job. The first method was to test it locally. The second method was to test the job by submitting both the test file to the HDFS and the MapReduce job the cluster. For my project I tested the job on the cluster. The following is a slice of the results obtained:

...
(Braatz,Kyra) 1

(Brown,Brianna) 1
(Brown,John) 3
(Brunet,Thomas) 2
...

The command line used to obtain these results was the following:

$ hadoop jar writables.jar stubs.StringPairTestDriver nameyeartestdata zzz_writable