Introduction to Hadoop – Part Two

Contents

[1 Working with Found Datasets 1](#_Toc3579908)

[2 Hadoop and Comma Separated Values (CSV) Files 1](#_Toc3579909)

[2.1 Sample Data 1](#_Toc3579910)

[2.2 Population Data: Population.java 2](#_Toc3579911)

[2.3 Running the Program 2](#_Toc3579912)

[2.3.1 Exercises to Do 3](#_Toc3579913)

[2.4 Joining the DataSets – CountyJoin.java 3](#_Toc3579914)

[2.4.1 Exercises to Do 5](#_Toc3579915)

[3 Apache Spark 5](#_Toc3579916)

[3.1 pyspark 5](#_Toc3579917)

[3.2 Student.json 5](#_Toc3579918)

[3.2.1 Using Data Frames 5](#_Toc3579919)

[3.2.2 Using SQL Queries 6](#_Toc3579920)

[3.2.3 Exercises to Do 6](#_Toc3579921)

[3.3 Exiting Apache Spark 6](#_Toc3579922)

[3.4 Weather.json 7](#_Toc3579923)

[3.4.1 Using Data Frames 7](#_Toc3579924)

[3.4.2 Using SQL Queries 8](#_Toc3579925)

[4 Spark and CSV Files 9](#_Toc3579926)

[4.1 Spark Queries 9](#_Toc3579927)

[4.2 Using SQL 9](#_Toc3579928)

[4.2.1 Exercises to Do 10](#_Toc3579929)

[4.3 HDFS and Apache Spark 10](#_Toc3579930)

[4.4 Final Word Count 11](#_Toc3579931)

[5 Streaming 11](#_Toc3579932)

[6 Summary 13](#_Toc3579933)

[Appendices 14](#_Toc3579934)

[Appendix One: Population.java 14](#_Toc3579935)

[Appendix Two: CountyJoin.java 16](#_Toc3579936)

The following workshop introduces the Hadoop server.

Command for you to try out are in blue text:

runThis command

Comments or sample results will be in purple:

Sample results….

Do note, the Hadoop server is a Linux box and so the commands below are case sensitive. Most Linux and Hadoop commands should be typed in lower case.

Be aware if you copy and paste from this document, Word sometimes converts the quotes into smart quotes automatically. For example, instead of a double quote: " it shows: “ and would generate invalid syntax.

# Working with Found Datasets

This workbook assumes you have completed Workshop 8 – Hadoop Part 1 and are familiar with how to access and run Java programs using Hadoop.

So far the examples have worked with simple text files with no particular structure. To be a meaningful tool Hadoop and its associated projects, need to be able to work with more complex data, such as that found in the Excel and JSON data files used in previous workbooks.

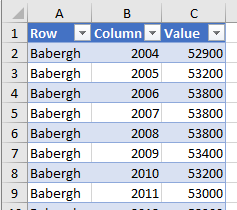
# Hadoop and Comma Separated Values (CSV) Files

Back in the early workbooks we used Excel and CSV files and loaded them into Oracle. CSV files can also be used in Hadoop.

## Sample Data

The following examples will use reshaped versions of the information found in the Data worksheets of the Population Estimates (aged 16-64) and Residents Pay spreadsheets seen in Week 1 and Week 2.

The following uses the reshaped versions of the data, for example, where the data looks similar to this:



Some of the Country names have been renamed from say *“Bristol, City of”* to *Bristol – City of*. This is because in the following program we want to treat the comma as a word separator and this avoids any confusion.

CSV versions of the files can be found on the hpd-srv server and copied to your own directory:

cp /home/6cs030/csv/pop.csv .

cp /home/6cs030/csv/pay.csv .

also copy these for later use:

cp /home/6cs030/csv/pop-header.csv .

cp /home/6cs030/csv/pay-header.csv .

The data has been reshaped in that it has been through the Pivot Table conversion seen in Workbook 2. Note, the data has been through some cleansing, such as removing rows which have no values at all, or replacing missing values with the average.

## Population Data: Population.java

The following program will work with the pop.csv file. The Word Count examples looked for spaces and punctuation to separate words, whereas in the CSV files, the data will be separated by commas.

The program is available from here to copy:

cp /home/6cs030/csv/Population.java .

The process is similar to the Word Count programs in that it imports some data and splits the data based on some criteria. Instead of just counting the words found, this program will total up the population values for each County and count how many rows there were. The County Names are the key in this case.

The overall code is available in Appendix One. Some things to note:

**Mapping Stage**

In CSV files, the data will be separated by commas, so we need to tell Hadoop to split using this:

String record = value.toString();

String[] parts = record.split(",");

**Reducer Stage**

By default, Hadoop uses tab (\t) characters to separate words when outputting the results. In this case we are going to output the final results also using commas, so the results could be treated as another CSV file. The output includes both the key and value, the following separates the values using a comma:

for (Text t : values) {

String parts[] = t.toString().split("\t");

popCount++;

popTotal += Integer.parseInt(parts[0]);

} // for loop

String str = String.format("**%d,%d"**, popCount, popTotal);

For the key we need to change a configuration parameter to change the default:

Configuration conf = new Configuration();

conf.set("mapreduce.output.textoutputformat.separator", ",");

**Delete the Output File**

In the Word Count examples, we had to ensure we always deleted the hdfs output directory before running the program.

This program is a bit more sophisticated in that it will delete the folder for you:

Path outputPath = new Path(args[1]);

FileOutputFormat.setOutputPath(job, outputPath);

outputPath.getFileSystem(conf).delete(outputPath, true);

## Running the Program

As before the program needs to be compiled first and a jar file created:

javac -classpath $(hadoop classpath) Pay.java

jar cf Pay.jar Pay\*.class

Create a new input called input\_csv directory and store the pop.csv file there:

hdfs dfs -mkdir input\_csv

hdfs dfs -put pay.csv input\_csvls

Run the file:

hadoop jar Pay.jar Pay input\_csv/pay.csv output\_csv

Check if the output file has been created:

hdfs dfs -ls output\_csv

To view the results:

hdfs dfs -cat output\_csv/part-r-00000

To retrieve the results to a csv file:

hdfs dfs -get output\_csv/part-r-00000 results.csv

The results.csv can be viewed using normal operating system commands such as:

more results.csv

One thing to note is the first line containing the column names no longer exists, so if you wanted to import this data into something that expected this, such as Oracle, you would need to add this information first.

### Exercises to Do

* Pick some of the counties and check that the totals add up!
* Produce a Java file that handles the Pay data instead (pay.csv). One thing to note is that the pay totals are currency values, so the numbers will be a float rather than the whole numbers seen in the Population file. You will need to amend the code that handles the figures to account for this.

## Joining the DataSets – CountyJoin.java

The above example only works with one of the csv files. The next example joins the two files, similar to joining two tables in a relational database.

The code can be seen in Appendix Two, to copy the file to your own directory:

cp /home/6cs030/csv/CountyJoin.java .

Some things to note:

**Two Mappers**

The two csv files are very similar, but have subtle differences in the type of data they contain. The figures in the population file are whole numbers, whilst the figures in the pay file contain currency information, so need to be treated slightly differently.

This means we need two Mapper classes to treat the input data accordingly. To help the reducer distinguish between the two, we will also prefix the value either with “pay” or “pop”.

For example:

public static class **PayMapper** extends Mapper <Object, Text, Text, Text> {

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

String record = value.toString();

String[] parts = record.split(",");

// 0: Key (county) 1: Year 2: figure

// default to 0 if null value

if (parts.length == 3 )

context.write(new Text(parts[0]), new Text(**"pay\t"** + parts[2]));

else

context.write(new Text(parts[0]), new Text(**"pay\t"** + 0));

} // map method

} // PayMapper

The PopMapper class is similar to before, except it also prefixes the values with “pop”:

context.write(new Text(parts[0]), new Text("**pop\t**" + parts[2]));

**Reducer Class**

The reducer class will use the prefix to then treat the data differently:

if (parts[0].equals("pop")) {

popCount++;

popTotal += Integer.parseInt(parts[1]);

} // if

else if (parts[0].equals("pay")) {

payCount++;

payTotal += Float.parseFloat(parts[1]);

} // else

**Main**

Because the input files are slightly different we want to control the order that Hadoop reads them and allocate the correct Mapper class. The following will assume the first file will be pay.csv and the second pop.csv:

MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class, PayMapper.class);

MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class, PopMapper.class);

To run the CountyJoin file:

Compile the code and produce a Jar file called CountyJoin.jar

javac -classpath $(hadoop classpath) SR\_DP.java

jar cf SR\_DP.jar SR\*.classhdfs

Put the pay.csv into the hdfs:

hdfs dfs -put pay.csv input\_csv

Run the code by providing the two input files:

hadoop jar SR\_DP.jar SR\_DP SR\_DP\_input/male\_2020.csv SR\_DP\_output

Retrieve the results file from hdfs:

hdfs dfs -get output\_csv/part-r-00000 join-results.txt

If you view the results you will notice this time the results are using the default tab separators:

more join-results.txt

### Exercises to Do

* Amend the code so that it produces the output using commas rather than tabs.

# Apache Spark

There are many ways to access Apache Spark:

* pyspark – uses Python
* spark-shell – uses Scala
* spark-sql – to run SQL queries
* spark-submit – to run a program file, such as Python

Or you can access it via a Java program.

See this webpage for further details and examples:

<https://spark.apache.org/docs/latest/sql-programming-guide.html>

When using JSON the above mentions that: “*Note that the file that is offered as* a json file *is not a typical JSON file. Each line must contain a separate, self-contained valid JSON object”.* For more information on the JSON Lines text format, see: <http://jsonlines.org/>

To run the following ensure you are logged onto hpd-srv.wlv.ac.uk first.

## pyspark

The following examples will use pyspark, which allows you to use Python for any programming tasks.

The weather.json dataset from the MongoDB tutorial will also be used for some of the examples, plus an extended version of the student.json file seen the lecture. A copy of these files are available on hpd-srv.wlv.ac.uk. To copy into your directory type:

cp /home/6cs030/data/weather.json .

cp /home/6cs030/data/student.json .

To access Spark from the operating system type:

pyspark

## Student.json

The following example shows how a JSON file can be imported. A Spark session is automatically available using the spark variable.

### Using Data Frames

The first examples will manipulate the data using a Spark DataFrame, this is equivalent to a relational table in SQL. See this link for further details:

<https://spark.apache.org/docs/1.3.0/api/scala/index.html#org.apache.spark.sql.DataFrame>

The Spark DataFrame is similar to the pandas DataFrame found in Python, but some of the methods and what they expect/return may differ slightly.

To create a DataFrame object based on the student.json file:

df = spark.read.json("student.json")

This stores the results in the df variable, which is a DataFrame. show() can be used to list the results:

df.show()

DataFrames are structured into columns and rows, to check what the schema is:

df.printSchema()

df represents a DataFrame object, so can be manipulated using methods associated with this data type. For example, to show just the name field:

df.select("name").show()

SQL-like operations can now be easily expressed:

df.select(df['name'], df['age'] + 1).show()

filter can be used to show only certain rows. To show students older than 21:

df.filter(df['age'] > 21).show()

groupBy is similar to the SQL GROUP BY command. To count how many students are on each course:

df.groupBy("course").count().show()

### Using SQL Queries

Spark implements a full SQL query engine which can convert SQL statements to a series of Resilient Distributed Dataset (RDD) transformations and actions. This second set of examples will use SQL to query the DataFrame.

First the DataFrame can be registered as a SQL temporary view:

df.createOrReplaceTempView("student")

This means that “student” can be queried as if it was a SQL table:

sqlDF = spark.sql("SELECT name, age, course FROM student WHERE age > 21")

sqlDF.show()

### Exercises to Do

Using both forms of syntax (Data Frame and SQL) write code to:

* Show just the name and lives fields
* Count how many people live at each place.

## Exiting Apache Spark

To leave the pyspark environment either type:

exit()

or press Ctrl-d

The latter can be used in any of the Spark systems.

## Weather.json

This example will use a larger dataset, such as the weather.json seen when using MongoDB.

Assuming you left pyspark in the previous section, restart it:

pyspark

### Using Data Frames

This time load the weather.json file into a Data Frame and view some data:

df = spark.read.json("weather.json")

df.show()

By default show() only lists the first 20 rows. To show more rows, include a number to represent how many rows should be listed. For example, to show 40 rows:

df.show(40)

This time the schema will be a lot more complex, to view it:

df.printSchema()

For reference, a text version of the schema can be found on Canvas, called weather-schema.txt

How many records are there?

df.count()

Show the first row:

df.first()

Or the first two rows:

df.take(2)

Show the summary statistics:

df.describe().show()

This produces descriptive statistics for numerical columns, such as the count, mean and standard deviation. For example, the first row contains the counts:

df.describe().first()

The dot notation can be used to view sub-documents. To show 40 of the user’s screen names:

df.select("user.screen\_name").show(40)

In a dataframe the *field* name must be in quotes. Either single or double quotes can be used, but you must be consistent.

The following would be the same as above:

df.select ('user.screen\_name').show(40)

but: df.select ('user.screen\_name").show(40)

will generate an error message!

When testing for equality using a data frame, use double equals (==) for the test. For example, show the tweets where the language is English (en):

df.filter(df['user.lang'] == "en").show()

To restrict both the rows and columns, a pipeline can be set up to pass one command to another:

df.filter(df['user.lang'] == "en").select('user.screen\_name', 'user.location').show(40)

This will retrieve the tweets where the language is English, then output just the screen name and location of the user.

By default Spark truncates long text fields, to avoid this use the Boolean False as a second parameter to show(). In this case you will also have to specify how many rows to show too.

Pattern matching can be done using the contains method. For example, find texts containing *sun*  and show the full text field:

df.select('text').filter(df['text'].contains("sun")).show(10, False)

### Using SQL Queries

Create a temporary view containing the weather data:

df.createOrReplaceTempView("weather")

The equivalent of the last data frame query is as follows (type on one line):

sqlDF = spark.sql("SELECT user.screen\_name, user.location FROM weather WHERE user.lang = 'en' ")

This time the field names do not need to be in quotes and the test for equality is a single equal sign. spark.sql takes a SQL string as a parameter which must be in either single or double quotes. The query includes a test for a string (en), which must be in different quotes to the SQL query – only single (') or double quotes (") should be used.

For example, the above used double quotes for the SQL query, then single quotes for the value. The following will also work, where single quotes are use for the SQL query and double quotes for the value test:

sqlDF = spark.sql('SELECT user.screen\_name, user.location FROM weather WHERE user.lang = "en" ')

What you cannot do is the use the same sort of quotes for both, so the following will not work:

sqlDF = spark.sql('SELECT user.screen\_name, user.location FROM weather WHERE user.lang = 'en' ')

To show 50 rows from the results:

sqlDF.show(50)

Most standard SQL commands will work. For example, count how many records there are:

sqlDF = spark.sql("SELECT count(\*) AS weather\_count FROM weather").show()

Count by language:

sqlDF = spark.sql("SELECT user.lang, count(\*) AS language\_count FROM weather GROUP BY user.lang").show()

The totals should add up to match the previous count!

The SQL LIKE command can be used for pattern matching. In MongoDB we listed just the Tweets containing sun in the text. The equivalent in SQL would be:

sqlDF = spark.sql("SELECT text FROM weather WHERE text LIKE '%sun%' ").show()

To make the query case insensitive by forcing the text field into upper case characters:

sqlDF = spark.sql("SELECT text FROM weather WHERE UPPER(text) LIKE '%SUN%' ").show(20, False)

# Spark and CSV Files

Spark can also read CSV files, as well as JSON format. The following examples will use the pay.csv and pop.csv files seen in Section 2, however, we will use a version of the files that includes a header in the first row, which can be used for the column headings (pay-header.csv and pop-header.csv). This is the only difference from the files above.

The following assumes you copied these files earlier (see Section 2.1).

Load the two CSV files into separate data frames:

dfPay = spark.read.format("csv").option("header", "true").load("pay-header.csv")

dfPop = spark.read.format("csv").option("header", "true").load("pop-header.csv")

option("header", "true") tells Hadoop to use the first line as the column headings.

## Spark Queries

Show some data:

dfPop.show()

dfPay.show()

As you can see it has used the headers for the column names. So could just show the county:

dfPop.select("county").show()

Or just Wolverhampton:

dfPop.filter(dfPop['county'] == 'Wolverhampton').show()

## Using SQL

To manipulate the dataframes using SQL syntax, covert the data frames to views:

Male2020.createOrReplaceTempView("Male2020")

dfPay.createOrReplaceTempView("pay")

Then we can join them as if they were two SQL tables:

df2020 = spark.sql("SELECT Male2020.\_c0, Female2020.\_c2, FROM Male2020, Female2020 WHERE Male2020.\_c0 = Female2020.\_c0 and Male2020.\_c1 = all\_ages and Female2020.\_c2 <1000").show(20, False)

Note, we need to join both on the County name and Year; otherwise we would get a semi Cartesian product.

Table aliases can be used to simplify the query. This query will just show the results for Wolverhampton and the year:

sqlDF = spark.sql("SELECT p.county, p.year, p.population, pa.annual\_pay FROM pay pa, pop p WHERE p.county = pa.county and p.year = pa.year and p.county = 'Wolverhampton' ").show(20, False)

Statistics can be carried out using SQL. For example, count how many rows there are for each county and sum the populations:

sqlDF = spark.sql("SELECT county, count(\*) as pop\_count, SUM(population) as pop\_sum FROM pop GROUP BY county ORDER BY county").show(20, False)

### Exercises to Do

* List the year, population and annul pay for Walsall.
* Sum the populations by year
* List each county, with a sum of the annual pay for all years.
* List each county, with a sum of the annual pay and population for all years.

## HDFS and Apache Spark

Spark can also read files from the HDFS file system, which is available via port 9000 on localhost. Assuming that the pay.csv and pop.csv files are still stored in your HDFS input\_csv directory:

*Male2020 = spark.read.format("csv").load("hdfs://localhost:9000/user/****1828421****/SR\_DP\_input/male\_2020.csv")*

When using files from HDFS, you need to give the full pathname to your file. Replace **<yourStudentNumber>** with your student number, for example:

dfPay2 = spark.read.format("csv").load("hdfs://localhost:9000/user/**0123456**/input\_csv/pay.csv")

This time the system has not used a header to define the column names:

dfPay2.show()

So to select one column, use the system generated names, such as \_c0:

dfPay2.select("\_c0").show()

Show the data for Wolverhampton:

dfPay2.filter(dfPay2['\_c0'] == "Wolverhampton").show()

## Final Word Count

Apache Spark also supports Map Reduce, so one last Word Count program (replace **0123456** with your own student number):

text\_file = sc.textFile("hdfs://localhost:9000/user/**0123456**/input\_word")

counts = text\_file.flatMap(lambda line: line.split(" ")) \

.map(lambda word: (word, 1)) \

.reduceByKey(lambda a, b: a + b)

counts.saveAsTextFile("hdfs://localhost:9000/user/**0123456**/spark\_output\_word")

This assumes that there is a text file in the input\_word hdfs directory, such as the testfiles or shakespeare.txt used earlier.

counts is a PythonRDD. A loop is needed to list the results:

for x in counts.collect():

print x

Or it can be converted to a DataFrame, then show() can be used to view it:

counts.toDF().show()

counts.saveAsTextFile will save the results a hdfs directory called spark\_output\_word. To view this exit pyspark by pressing Ctrl+D

Then list the contents of the spark\_output\_word directory:

hdfs dfs -ls spark\_output\_word

Assuming the program ran correctly, the results will be similar to:

Found 3 items

-rw-r--r-- 3 0123456 hadoop 0 2019-03-15 13:24 spark\_output\_word/\_SUCCESS

-rw-r--r-- 3 0123456 hadoop 559584 2019-03-15 13:24 spark\_output\_word/part-00000

-rw-r--r-- 3 0123456 hadoop 562396 2019-03-15 13:24 spark\_output\_word/part-00001

The intermediate file (part-0000) still exists, but the final results can be seen by typing:

hdfs dfs -cat spark\_output\_word/part-00001

The output can be retrieved using the –get option as seen before:

hdfs dfs -get spark\_output\_word/part-00001 spark-results.txt

# Streaming

Spark can be used to manipulate streaming data, such as live tweets or satellite data. Analysing live tweets can incur a cost, so the following will use streaming data generated by a Weather Station, based at the High Performance Wireless Research and Education Network (HPWREN), University of California, San Diego (<http://hpwren.ucsd.edu/>).

Start pyspark if not already using it.

First import and create a streaming context:

from pyspark.streaming import StreamingContext

ssc = StreamingContext(sc,1)

A *Streaming Context* provides an interface to Spark’s streaming capability, 1 specifies a batch interval of one second.

Next create a DStream to hold the weather data:

lines = ssc.socketTextStream("rtd.hpwren.ucsd.edu", 12028)

This create a new variable called lines to be a Spark DStream that streams the lines of output from the weather station.

Next create a function to parse each line and return the average wind direction (Dm):

import re

def parse(line):

match = re.search("Dm=(\d+)", line)

if match:

val = match.group(1)

return [int(val)]

return []

The following reads the measurements and calls *parse* above to get the average wind speed:

vals = lines.flatMap(parse)

Spark window functions are functions that perform a calculation over a group of records, which are called a window.

Create a sliding window of data:

window = vals.window(10,5)

This creates a DStream called window that combines ten seconds of data and moves by five seconds.

Now define a function called stats to produce some statistics, in this case the minimum and maximum value of our window:

def stats(rdd):

print(rdd.collect())

if rdd.count() > 0:

print("max = {}, min = {}".format(rdd.max(), rdd.min()))

Link the stats() function with each RDD in the DStream:

window.foreachRDD(lambda rdd: stats(rdd))

Finally start the stream process:

ssc.start()

This will produce a set of streaming data, such as:

[76, 79, 71, 72]

max = 79, min = 71

[76, 79, 71, 72, 66, 58, 65, 48, 44]

max = 79, min = 44

[66, 58, 65, 48, 44, 36, 33, 28, 34, 30]

max = 66, min = 28

[36, 33, 28, 34, 30, 20, 18, 18, 22, 13]

max = 36, min = 13

[20, 18, 18, 22, 13, 28, 27, 36, 30, 34]

max = 36, min = 13

[28, 27, 36, 30, 34, 43, 44, 35, 33, 39]

max = 44, min = 27

43, 44, 35, 33, 39, 33,

Let it run for a minute, when finished, press Ctrl-d to stop the program.

# Summary

Workbooks 8 and 9 give you an introduction to using the Hadoop Distributed File System (HDFS) either by accessing it via Java or via Spark.

There are plenty of further examples available online. If using Spark, note there are differences between Versions 1 and 2. For example, a Spark Session is automatically created for you called spark and a Spark Context is available as sc in Version 2.

.

# Appendices

## Appendix One: Population.java

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class Population {

public static class PopMapper extends Mapper <Object, Text, Text, Text> {

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

String record = value.toString();

String[] parts = record.split(",");

// 0: Key (county) 1: Year 2: figure

// need to deal with null values - defaults to 0

if (parts.length == 3 )

context.write(new Text(parts[0]), new Text(parts[2]));

else

context.write(new Text(parts[0]), new Text("0"));

} // map

} // PopMapper

public static class csvReducer extends Reducer <Text, Text, Text, Text> {

public void reduce(Text key, Iterable<Text> values, Context context)

throws IOException, InterruptedException {

String popName = "";

int popTotal = 0;

int popCount = 0;

for (Text t : values) {

String parts[] = t.toString().split("\t");

popCount++;

popTotal += Integer.parseInt(parts[0]);

} // for loop

String str = String.format("%d,%d", popCount, popTotal);

context.write(new Text(key), new Text(str));

} //reduce

} // csvReducer

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

//set output delimiter to comma

conf.set("mapreduce.output.textoutputformat.separator", ",");

Job job = Job.getInstance(conf, "Population Count");

job.setJarByClass(Population.class);

job.setMapperClass(PopMapper.class);

job.setReducerClass(csvReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

FileInputFormat.addInputPath(job, new Path(args[0]));

Path outputPath = new Path(args[1]);

FileOutputFormat.setOutputPath(job, outputPath);

// Delete the output directory - true means if path is a directory it does recursive delete

outputPath.getFileSystem(conf).delete(outputPath, true);

System.exit(job.waitForCompletion(true) ? 0 : 1);

} // main

} // Population class

## Appendix Two: CountyJoin.java

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;

import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class CountyJoin {

public static class PayMapper extends Mapper <Object, Text, Text, Text> {

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

String record = value.toString();

String[] parts = record.split(",");

// 0: Key (county) 1: Year 2: figure

// default to 0 if null value

if (parts.length == 3 )

context.write(new Text(parts[0]), new Text("pay\t" + parts[2]));

else

context.write(new Text(parts[0]), new Text("pay\t" + 0));

} // map method

} // PayMapper

public static class PopMapper extends Mapper <Object, Text, Text, Text> {

public void map(Object key, Text value, Context context)

throws IOException, InterruptedException {

String record = value.toString();

String[] parts = record.split(",");

// 0: Key (county) 1: Year 2: figure

// default to 0 if null value

if (parts.length == 3 )

context.write(new Text(parts[0]), new Text("pop\t" + parts[2]));

else

context.write(new Text(parts[0]), new Text("pop\t" + 0));

} // map method

} // PopMapper

public static class JoinReducer extends Reducer <Text, Text, Text, Text> {

public void reduce(Text key, Iterable<Text> values, Context context)

throws IOException, InterruptedException {

String payName = "";

String popName = "";

double payTotal = 0.0;

int popTotal = 0;

int popCount = 0;

int payCount = 0;

for (Text t : values) {

String parts[] = t.toString().split("\t");

if (parts[0].equals("pop")) {

popCount++;

popTotal += Integer.parseInt(parts[1]);

} // if

else if (parts[0].equals("pay")) {

payCount++;

payTotal += Float.parseFloat(parts[1]);

} // else

} // for loop

String str = String.format("%d\t%f\t%d\t%d", payCount, payTotal, popCount, popTotal);

context.write(new Text(key), new Text(str));

} // reduce method

} // JoinReducer

public static void main(String[] args) throws Exception {

Configuration conf = new Configuration();

Job job = new Job(conf, "Reduce-side join");

job.setJarByClass(CountyJoin.class);

job.setReducerClass(JoinReducer.class);

job.setOutputKeyClass(Text.class);

job.setOutputValueClass(Text.class);

MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class, PayMapper.class);

MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class, PopMapper.class);

Path outputPath = new Path(args[2]);

FileOutputFormat.setOutputPath(job, outputPath);

outputPath.getFileSystem(conf).delete(outputPath);

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}