**Clairvoyant**

**Intro to Apache Spark Workshop:**

**Advanced Exercise Answers**

**Exercise 1 – Running Spark Jobs**

**Advanced Question:**

N/A

**Exercise 2 – Access Logs**

**Advanced Question:**

In the advanced portion of the exercise you will work with the same access.log file that you used in the previous section. Do the following:

* Map each line into an AccessLogEntry object which contains the fields: ipAddress, date, method, resource, protocol, status, url, userAgent

For Example, the line:

10.236.133.247 - - [Mon, 19 May 2014 16:29:29 GMT] "GET /admin HTTP/1.1" 304 - "https://my.analytics.app/admin" "Mozilla/5.0"

Will translate to:

* ipAddress: 10.236.133.247
* date: Mon, 19 May 2014 16:29:29 GMT
* method: GET
* resource: /admin
* protocol: HTTP/1.1
* status: 304
* url: https://my.analytics.app/admin
* userAgent: Mozilla/5.0

**Java Answer:**

public static String *accessLogRegex* = "^(\\S+) (\\S+) (\\S+) \\[(.\*?)\\] \"(.\*?) (.\*?) (.\*?)\" (\\S+) (\\S+)( \"(.\*?)\" \"(.\*?)\")?";  
public static String *dateFormat* = "E, d MMM y H:m:s z";  
static class AccessLogEntry implements Serializable {  
 public String ipAddress;  
 public Date date;  
 public String method;  
 public String resource;  
 public String protocol;  
 public String status;  
 public String url;  
 public String userAgent;  
 public AccessLogEntry(String ipAddress, Date date, String method, String resource, String protocol, String status, String url, String userAgent) {  
 this.ipAddress = ipAddress;  
 this.date = date;  
 this.method = method;  
 this.resource = resource;  
 this.protocol = protocol;  
 this.status = status;  
 this.url = url;  
 this.userAgent = userAgent;  
 }  
 @Override  
 public String toString() {  
 return "AccessLogEntry{" +  
 "ipAddress='" + ipAddress + '\'' +  
 ", date=" + date +  
 ", method='" + method + '\'' +  
 ", resource='" + resource + '\'' +  
 ", protocol='" + protocol + '\'' +  
 ", status='" + status + '\'' +  
 ", url='" + url + '\'' +  
 ", userAgent='" + userAgent + '\'' +  
 '}';  
 }  
}

public static AccessLogEntry stringToAccessLogEntry(String accessLogLine) throws Exception {  
 Pattern p = Pattern.*compile*(*accessLogRegex*);  
 Matcher m = p.matcher(accessLogLine);  
 if(m.matches()) {  
 if(m.groupCount() == 12) {  
 return new AccessLogEntry(  
 m.group(1),  
 new SimpleDateFormat(*dateFormat*).parse(m.group(4)),  
 m.group(5),  
 m.group(6),  
 m.group(7),  
 m.group(8),  
 m.group(11),  
 m.group(12)  
 );  
 } else {  
 System.*out*.println("ERROR: Group Count of '" + m.groupCount() + "' isn't 12. There isn't enough information in the line '" + accessLogLine + "' to create the AccessLogEntry object.");  
 }  
 } else {  
 System.*out*.println("ERROR: Regular Expression didn't match line '" + accessLogLine + "'");  
 }  
 return null;  
}

JavaRDD<AccessLogEntry> advancedMappedAccessLogs = accessLogs.map(new Function<String, AccessLogEntry>() {  
 public AccessLogEntry call(String line) throws Exception {  
 return *stringToAccessLogEntry*(line);  
 }  
});  
advancedMappedAccessLogs.saveAsTextFile("/user/cloudera/spark-workshop-output-data/logs/access-logs-advanced-mapped");

**Scala Answer:**

accessLogs.map(accessLogLine => {  
 **val** p: Pattern = Pattern.*compile*(*accessLogRegex*)  
 **val** m: Matcher = p.matcher(accessLogLine)  
 **if** (m.matches) {  
 **if** (m.groupCount == 12) {  
 **return new** AccessLogEntry(m.group(1), **new** SimpleDateFormat(*dateFormat*).parse(m.group(4)), m.group(5), m.group(6), m.group(7), m.group(8), m.group(11), m.group(12))  
 }  
 **else** {  
 System.*out*.println("ERROR: Group Count of '" + m.groupCount + "' isn't 12. There isn't enough information in the line '" + accessLogLine + "' to create the AccessLogEntry object.")  
 }  
 }  
 **else** {  
 System.*out*.println("ERROR: Regular Expression didn't match line '" + accessLogLine + "'")  
 }  
 **null** }).saveAsTextFile("/user/cloudera/spark-workshop-output-data/logs/access-logs-advanced-mapped")  
 }  
  
}

**Python Answer:**

accessLogRegex = "^(\\S+) (\\S+) (\\S+) \\[(.\*?)\\] \"(.\*?) (.\*?) (.\*?)\" (\\S+) (\\S+)( \"(.\*?)\" \"(.\*?)\")?"  
dateFormat = "E, d MMM y H:m:s z"  
**class AccessLogEntry**(object):  
 **def** \_\_init\_\_(self, ipAddress, date, method, resource, protocol, status, url, userAgent):  
 self.ipAddress = ipAddress  
 self.date = date  
 self.method = method  
 self.resource = resource  
 self.protocol = protocol  
 self.status = status  
 self.url = url  
 self.userAgent = userAgent  
 **def** \_\_str\_\_(self):  
 **return** "AccessLogEntry{" + \  
 "ipAddress='" + self.ipAddress + '\'' + \  
 ", date=" + self.date + \  
 ", method='" + self.method + '\'' + \  
 ", resource='" + self.resource + '\'' + \  
 ", protocol='" + self.protocol + '\'' + \  
 ", status='" + self.status + '\'' + \  
 ", url='" + self.url + '\'' + \  
 ", userAgent='" + self.userAgent + '\'' + \  
 '}'  
  
**def stringToAccessLogEntry**(accessLogLine):  
 m = re.search(accessLogRegex, accessLogLine)  
 **return** AccessLogEntry(m.group(1), m.group(4), m.group(5), m.group(6), m.group(7), m.group(8), m.group(11), m.group(12))  
  
accessLogs.map(stringToAccessLogEntry).saveAsTextFile("/user/cloudera/spark-workshop-output-data/logs/access-logs-advanced-mapped")

**Exercise 3 – Joining Datasets**

**Advanced Question:**

Using the join data that was used in the standard question, get the addresses that don’t have a name associated with them.

Hint: there’s a type of join operation you can use for this.

**Java Answer:**

final int companyAddressIDColumnIndex = 2;  
final int addressAddressIDColumnIndex = 0;  
  
JavaRDD<String> company = sc.textFile("/user/cloudera/spark-workshop-data/join/company.tsv");  
JavaRDD<String> address = sc.textFile("/user/cloudera/spark-workshop-data/join/address.tsv");  
  
JavaPairRDD<String, String> companyParsed = company.mapToPair(new PairFunction<String, String, String>() {  
 public Tuple2<String, String> call(String line) throws Exception {  
 String[] lineSplit = line.split("\t");  
 return new Tuple2<String, String>(lineSplit[companyAddressIDColumnIndex], line);  
 }  
});  
JavaPairRDD<String, String> addressParsed = address.mapToPair(new PairFunction<String, String, String>() {  
 public Tuple2<String, String> call(String line) throws Exception {  
 String[] lineSplit = line.split("\t");  
 return new Tuple2<String, String>(lineSplit[addressAddressIDColumnIndex], line);  
 }  
});  
  
JavaRDD<String> addressesWithoutCompany = addressParsed.leftOuterJoin(companyParsed).filter(new Function<Tuple2<String, Tuple2<String, Optional<String>>>, Boolean>() {  
 public Boolean call(Tuple2<String, Tuple2<String, Optional<String>>> stringTuple2Tuple2) throws Exception {  
 return !stringTuple2Tuple2.\_2().\_2().isPresent();  
 }  
}).map(new Function<Tuple2<String, Tuple2<String, Optional<String>>>, String>() {  
 public String call(Tuple2<String, Tuple2<String, Optional<String>>> stringTuple2Tuple2) throws Exception {  
 return stringTuple2Tuple2.\_2().\_1();  
 }  
});  
  
List<String> list = addressesWithoutCompany.collect();  
for(String ad : list) {  
 System.*out*.println(ad);  
}  
//A006 1706 E. Curry Tempe AZ 85280

**Scala Answer:**

**val** companyAddressIDColumnIndex = 2  
**val** addressAddressIDColumnIndex = 0  
  
**val** company = sc.textFile("/user/cloudera/spark-workshop-data/join/company.tsv")  
**val** address = sc.textFile("/user/cloudera/spark-workshop-data/join/address.tsv")  
  
**val** companyParsed = company.map( line => (line.split("\t")(companyAddressIDColumnIndex), line))  
**val** addressParsed = address.map( line => (line.split("\t")(addressAddressIDColumnIndex), line))  
  
**val** addressesWithoutCompany = addressParsed  
 .leftOuterJoin(companyParsed)  
 .filter(entry => entry.\_2.\_2.isEmpty)  
 .map(entry => entry.\_2.\_1)

.collect()  
  
addressesWithoutCompany.foreach(*println*)  
//A006 1706 E. Curry Tempe AZ 85280

**Python Answer:**

companyAddressIDColumnIndex = 2  
 addressAddressIDColumnIndex = 0  
  
 company = sc.textFile("/user/cloudera/spark-workshop-data/join/company.tsv")  
 address = sc.textFile("/user/cloudera/spark-workshop-data/join/address.tsv")  
  
 companyParsed = company.map(**lambda** line: (line.split("\t")[companyAddressIDColumnIndex], line))  
 addressParsed = address.map(**lambda** line: (line.split("\t")[addressAddressIDColumnIndex], line))  
  
 addressesWithoutCompany = addressParsed\  
 .leftOuterJoin(companyParsed)\  
 .filter(**lambda** (key\_1, (address\_1, company\_2)): company\_2 == None)\  
 .map(**lambda** (key\_1, (address\_1, company\_2)): address\_1)\  
 .collect()  
  
 **for** entry **in** addressesWithoutCompany:  
 **print**(entry)  
# [u'A006\t1706 E. Curry\tTempe\tAZ\t85280']

**Exercise 4 – Shared Variables**

**Advanced Question:**

Use an **Accumulator** to count the number of user transactions that have the most used TransactionCodes for that bank. Create a **Broadcast** variable first to contain all the most used TransactionCodes for each bank by processing the data and collecting it to the Driver.

**Java Answer:**

JavaRDD transactionFile = sc.textFile("/user/cloudera/spark-workshop-data/transactions/user\_financial\_transactions.tsv");  
JavaRDD transactionData = transactionFile.map(new Function<String, String[]>() {  
 public String[] call(String line) throws Exception {  
 return line.split("\t");  
 }  
});  
  
Map<String, String> broadcastData = transactionData.mapToPair(new PairFunction<String[], Tuple2<String, String>, Integer>() {  
 public Tuple2<Tuple2<String, String>, Integer> call(String[] entry) throws Exception {  
 return new Tuple2<Tuple2<String, String>, Integer>(new Tuple2<String, String>(entry[5], entry[3]), 1);  
 }  
}).reduceByKey(new Function2<Integer, Integer, Integer>() {  
 public Integer call(Integer o, Integer o2) throws Exception {  
 return o + o2;  
 }  
}).mapToPair(new PairFunction<Tuple2<Tuple2<String, String>, Integer>, String, Tuple2<String, Integer>>() {  
 public Tuple2<String, Tuple2<String, Integer>> call(Tuple2<Tuple2<String, String>, Integer> entry) throws Exception {  
 return new Tuple2<String, Tuple2<String, Integer>>(entry.\_1().\_1(), new Tuple2<String, Integer>(entry.\_1().\_2(), entry.\_2()));  
 }  
}).reduceByKey(new Function2<Tuple2<String, Integer>, Tuple2<String, Integer>, Tuple2<String, Integer>>() {  
 public Tuple2<String, Integer> call(Tuple2<String, Integer> d1, Tuple2<String, Integer> d2) throws Exception {  
 if(d1.\_2() > d2.\_2()) {  
 return d1;  
 } else {  
 return d2;  
 }  
 }  
}).mapToPair(new PairFunction<Tuple2<String, Tuple2<String, Integer>>, String, String>() {  
 public Tuple2<String, String> call(Tuple2<String, Tuple2<String, Integer>> entry) throws Exception {  
 return new Tuple2<String, String>(entry.\_1(), entry.\_2().\_1());  
 }  
}).collectAsMap();  
  
final Broadcast broadcasted = sc.broadcast(broadcastData);  
final Accumulator countAccum = sc.accumulator(0);  
  
transactionData.foreach(new VoidFunction<String[]>() {  
 public void call(String[] entry) throws Exception {  
 Map<String, String> broadcastedMap = (Map<String, String>) broadcasted.getValue();  
 if(broadcastedMap.get(entry[5]).equals(entry[3])) {  
 countAccum.add(1);  
 }  
 }  
});  
  
System.*out*.println(countAccum.value());  
//9

**Scala Answer:**

**def** reduceTuples(d1: Tuple2[String, Int], d2: Tuple2[String, Int]): Tuple2[String, Int] = {  
 **if**(d1.\_2 > d2.\_2) {  
 **return** d1  
 } **else** {  
 **return** d2  
 }  
}  
  
**def** transformListIntoMap(list: Array[(String, String)]): mutable.HashMap[String, String] = {  
 **var** map = **new** mutable.HashMap[String, String]()  
 list.foreach(entry => map.put(entry.\_1, entry.\_2))  
 **return** map  
}  
  
  
**case class** Transaction (userId: String, name: String, transactionId: String, transactionCode: String, reason: String, bankId: String)  
  
**val** transactionFile = sc.textFile("/user/cloudera/spark-workshop-data/transactions/")  
**val** transactionData = transactionFile.map(\_.split("\t")).map(  
 r => *Transaction*(r(0), r(1), r(2), r(3), r(4), r(5))  
)  
  
**val** broadcastData = transactionData.map(entry => {  
 ((entry.bankId, entry.transactionCode), 1)  
 }).reduceByKey(\_ + \_).map(entry => (entry.\_1.\_1, (entry.\_1.\_2, entry.\_2))).reduceByKey(reduceTuples).map(entry => (entry.\_1, entry.\_2.\_1)).collect()  
  
**val** broadcasted = sc.broadcast(transformListIntoMap(broadcastData))  
**val** accumulator = sc.accumulator(0)  
  
transactionData.foreach(entry => {  
 **if**(broadcasted.value(entry.bankId) == entry.transactionCode) {  
 accumulator += 1  
 }  
})  
  
*println*(accumulator.value)  
//9

**Python Answer:**

**def reduceTuples**(d1, d2):  
 **if** d1[1] > d2[1]:  
 **return** d1  
 **else**:  
 **return** d2  
  
**def transformListIntoMap**(list):  
 map = {}  
 **for** entry **in** list:  
 map[entry[0]] = entry[1]  
 **return** map  
  
transactionFile = sc.textFile("/user/cloudera/spark-workshop-data/transactions/user\_financial\_transactions.tsv")  
transactionData = transactionFile.map(**lambda** line: line.split("\t"))  
  
broadcastData = transactionData.map(**lambda** entry: ((entry[5], entry[3]), 1))\  
 .reduceByKey(**lambda** a, b: a + b)\  
 .map(**lambda** entry: (entry[0][0], (entry[0][1], entry[1])))\  
 .reduceByKey(reduceTuples)\  
 .map(**lambda** entry: (entry[0], entry[1][0]))\  
 .collect()  
  
  
  
broadcasted = sc.broadcast(transformListIntoMap(broadcastData))  
countAccum = sc.accumulator(0)  
  
**def accumFunction**(entry):  
 **global** countAccum  
 **if** broadcasted.value[entry[5]] == entry[3]:  
 countAccum += 1  
  
transactionData.foreach(accumFunction)  
  
**print**(countAccum.value)  
#9