Spark Fundamentals - II

Optimizing Transformations and Actions

Contents

Optimizing Transformations and Actions

You will be using the same bike dataset from the previous lab. There are some helper Scala classes, which will be used to parse each lines of the CSV file instead of working with raw arrays.

After completing this hands-on lab, you should be able to:

- Use advanced Pair RDD operations such as groupByKey and aggregateByKey
- Understand why you should use aggregateByKey and avoid groupByKey
- Familiarize yourself with different operations and their impact to performance and job scheduling

Allow 30 minutes to complete this section of lab.

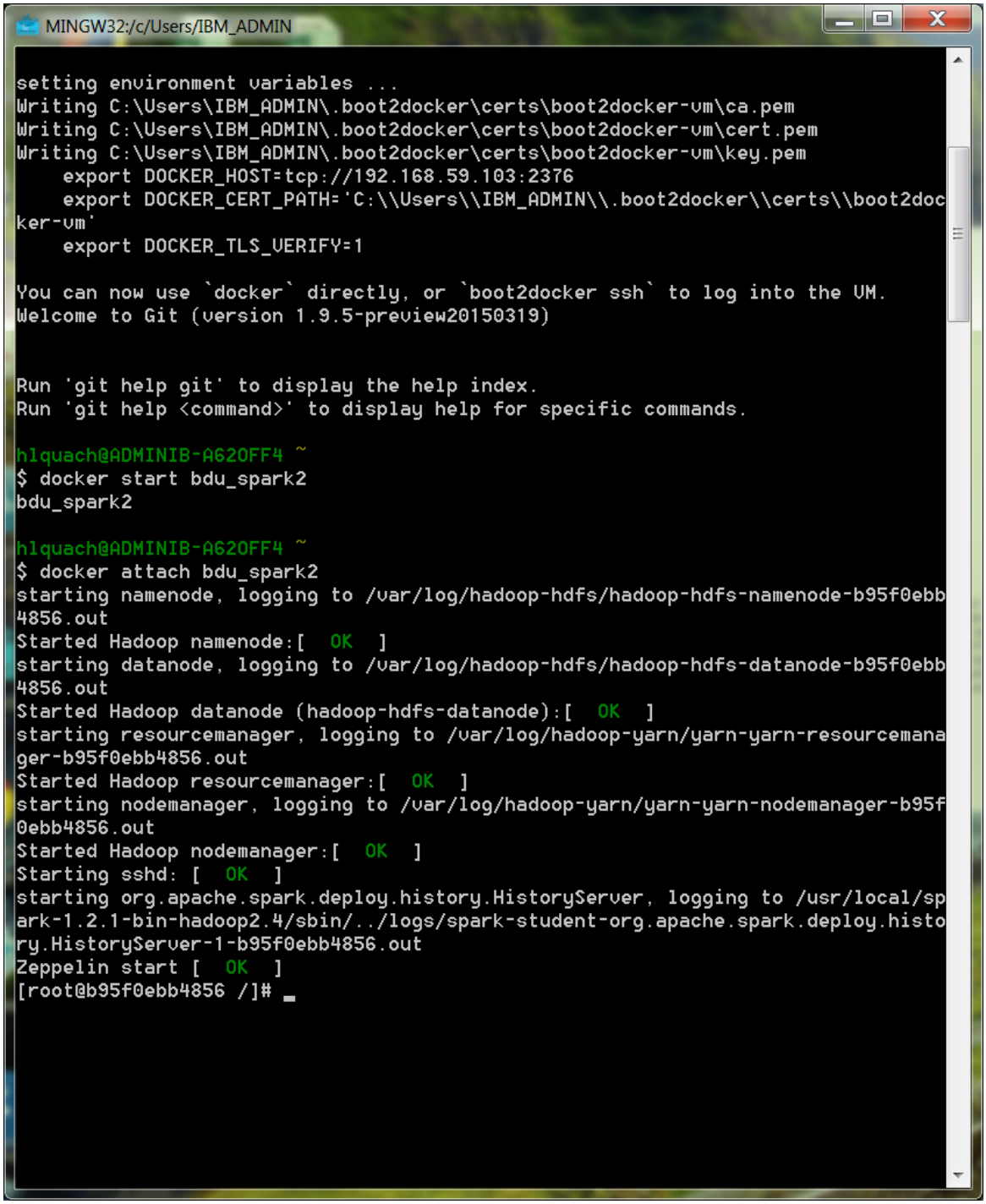
1.1 Optimizing Transformations and Actions

Make sure your docker and the Zeppelin is running. If not, get it started before continuing with this lab. Use the instructions here:

https://registry.hub.docker.com/u/bigdatauniversity/spark2/

We are using the same two datasets from the last lab.

__1. Here I show how you would restart the container if you need to. Otherwise, continue from the previous instance of the boot2docker terminal from lab 2.



__2. Open up **Lab 3** in the Zeppelin notebook. You will start by reading the input files and parsing each line into a trip or station record. This will allow us to work with a Scala object, rather than directly from the raw array.

__3. You can take a few minutes to review this helper class.

__4. Essentially, it creates Serializable objects Trip and Station with all the columns from the two CSV files. In the *case class Trip*, there are the columns: id, duration, startDate, startStation, startTerminal, endDate, endStation, endTerminal, bike, subscriberType, and zipCode.

In the case class Station, there are the columns: id, name, lat, lon, docks, landmark, and installDate

__5. Now create the trips and stations RDD by utilizing the helper class. To create the *trips* RDD, you are going to use the exact same transformations that you had done before, to load in the text file, filter out the header column and map each individual value. The difference here, then, would be to perform another map operation to parse each line into a *Trip* record.

```
val trips = input1.filter(_ !=
header1).map(_.split(",")).map(utils.Trip.parse(_))
```

__6. The same operations will be done for the stations RDD as well:

```
val stations = input2.filter(_ !=
header2).map(_.split(",")).map(utils.Station.parse(_))
```

__7. Run the panel by filling in the codes to create the trips and stations RDD.

__8. Calculate the average duration of each start terminal using *groupByKey*. First you need to convert the trips RDD into a Pair RDD. Use the *keyBy* method.

```
val byStartTerminal = trips.keyBy(_.startStation)
```

__9. Next get the duration for each of the start terminal by mapping the values (duration) to the key.

```
val durationsByStart = byStartTerminal.mapValues(_.duration)
```

__10. Group the stations together to get their average duration.

```
val grouped = durationsByStart.groupByKey().mapValues(list =>
list.sum/list.size)
```

__11. Get the first 10 values:

```
grouped.take(10).foreach(println)
```

```
byStartTerminal: org.apache.spark.rdd.RDD[(String, utils.Trip)] = MappedRDD[63] at keyBy at <console>:41
durationsByStart: org.apache.spark.rdd.RDD[(String, Int)] = MappedValuesRDD[64] at mapValues at <console>:44
grouped: org.apache.spark.rdd.RDD[(String, Int)] = MappedValuesRDD[66] at mapValues at <console>:46
(Mountain View City Hall,1663)
(California Ave Caltrain Station,5023)
(San Jose Civic Center,3028)
(Yerba Buena Center of the Arts (3rd @ Howard),970)
(Commercial at Montgomery,761)
(SJSU 4th at San Carlos,1966)
(2nd at South Park,683)
(University and Emerson,6545)
(Embarcadero at Sansome,1549)
(Townsend at 7th,746)
Took 4 seconds
```

__12. As you recall from the lesson, *groupByKey* should be avoided when possible, because all the key-value pairs are shuffled around. Instead, use *aggregateByKey*, so that the keys are combined first on the same key before it is shuffled. Let's see how this is done:

```
val results = durationsByStart.aggregateByKey((0, 0))((acc, value) => (acc._1 +
value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
```

__13. Let's break this down:

```
val results = durationsByStart.aggregateByKey((0, 0))((acc, value) => (acc._1 +
value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
```

That's just the zeroValue, the initial amount you will start the aggregation.

```
val results = durationsByStart.aggregateByKey((0, 0))((acc, value) => (acc._1 +
value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
```

This is the function to aggregate the values of each key. It keeps the running tally of sum + count so that we could calculate the averages later. It performs the reduce function in the current partition before the data is shuffled out. This reduces the unnecessary data from being transferred out.

```
val results = durationsByStart.aggregateByKey((0, 0))((acc, value) => (acc._1 +
value, acc._2 + 1), (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
```

This is the function to aggregate the output of the first function -- happens after the shuffle to do the final aggregation. This adds up all the sums and counts together. A lot more efficient because everything comes in as compact as possible.

__14. Compute the average of the results by diving the sums over the counts and print the value to the console.

```
val finalAvg = results.mapValues(i => i._1 / i._2)

finalAvg.take(10).foreach(println)
```

```
byStartTerminal: org.apache.spark.rdd.RDD[(String, utils.Trip)] = MappedRDD[81] at keyBy at <console>:41
durationsByStart: org.apache.spark.rdd.RDD[(String, Int)] = MappedValuesRDD[82] at mapValues at <console>:44
results: org.apache.spark.rdd.RDD[(String, (Int, Int))] = ShuffledRDD[83] at aggregateByKey at <console>:48
finalAvg: org.apache.spark.rdd.RDD[(String, Int)] = MappedValuesRDD[84] at mapValues at <console>:47
(Mountain View City Hall,1663)
(California Ave Caltrain Station,5023)
(San Jose Civic Center,3028)
(SJSU 4th at San Carlos,1966)
(Commercial at Montgomery,761)
(Yerba Buena Center of the Arts (3rd @ Howard),970)
(2nd at South Park,683)
(University and Emerson,6545)
(Embarcadero at Sansome,1549)
(Townsend at 7th,746)
Took 3 seconds
```

Notice the time savings between the two functions. Although it's very small, this would matter if we were dealing with actual big data sets. Our data set is just small sample for the purpose of showing the functionality of Spark.

__15. Find the first trip starting at each terminal using *groupByKey*. Remember, Spark is best when everything is done in-memory. However, when you use *groupByKey* operations on a large dataset, you will likely get an OOM error because the entire dataset is stored in memory in order to be grouped. This is very inefficient. We show that this works for our dataset, but it may not work if the dataset is very large.

```
val firstGrouped = byStartTerminal.groupByKey().mapValues(list =>
list.toList.sortBy(_.startDate.getTime).head)
println(firstGrouped.toDebugString)
firstGrouped.take(10).foreach(println)
```

```
val firstGrouped = byStartTerminal.groupByKey().mapValues(list => list.toList.sortBy(_.startDate.getTime).head)
println(firstGrouped.toDebugString)
firstGrouped.take(10).foreach(println)
```

```
firstGrouped: org.apache.spark.rdd.RDD[(String, utils.Trip)] = MappedValuesRDD[92] at mapValues at <console>:43
(2) MappedValuesRDD[92] at mapValues at <console>:43 []
 |  ShuffledRDD[91] at groupByKey at <console>:43 []
 +-(2) MappedRDD[87] at keyBy at <console>:41 []
    |  MappedRDD[57] at map at <console>:43 []
    |  MappedRDD[56] at map at <console>:42 []
    |  FilteredRDD[55] at filter at <console>:41 []
    |  data/trips/* MappedRDD[54] at textFile at <console>:34 []
    |  data/trips/* HadoopRDD[53] at textFile at <console>:34 []
(Mountain View City Hall,Trip(4081,218,2013-08-29 09:38:00.0,Mountain View City Hall,27,2013-08-29 09:41:00.0,Mountain Vi
(California Ave Caltrain Station,Trip(4375,880,2013-08-29 12:26:00.0,California Ave Caltrain Station,36,2013-08-29 12:41:
(San Jose Civic Center,Trip(4510,166,2013-08-29 13:31:00.0,San Jose Civic Center,3,2013-08-29 13:34:00.0,San Salvador at
(Yerba Buena Center of the Arts (3rd @ Howard),Trip(4355,2044,2013-08-29 12:18:00.0,Yerba Buena Center of the Arts (3rd @
(Commercial at Montgomery,Trip(4086,178,2013-08-29 09:42:00.0,Commercial at Montgomery,45,2013-08-29 09:45:00.0,Commercia
(SJSU 4th at San Carlos,Trip(4394,1309,2013-08-29 12:34:00.0,SJSU 4th at San Carlos,12,2013-08-29 12:56:00.0,San Salvador
(2nd at South Park,Trip(4069,174,2013-08-29 09:08:00.0,2nd at South Park,64,2013-08-29 09:11:00.0,2nd at South Park,64,28
(University and Emerson,Trip(4116,1213,2013-08-29 10:11:00.0,University and Emerson,35,2013-08-29 10:32:00.0,California A
(Embarcadero at Sansome,Trip(4388,3582,2013-08-29 12:33:00.0,Embarcadero at Sansome,60,2013-08-29 13:33:00.0,San Francisc
(Townsend at 7th,Trip(4315,1432,2013-08-29 12:07:00.0,Townsend at 7th,65,2013-08-29 12:31:00.0,5th at Howard,57,538,Custo
```

Took 5 seconds

__16. The better approach would be to use *reducebyKey*, which will look within its own worker node for the first trip and only send out at most one to be combined with the other worker nodes (instead of sending everything out to be combined):

```
val firstTrips = byStartTerminal.reduceByKey((a, b) => {
    a.startDate.before(b.startDate) match {
        case true => a
        case false => b
    }
})
println(firstTrips.toDebugString)
firstTrips.take(10)
```

```
val firstTrips = byStartTerminal.reduceByKey((a, b) => {
    a.startDate.before(b.startDate) match {
        case true => a
        case false => b
    }
})
println(firstTrips.toDebugString)
firstTrips.take(10)
```

```
firstTrips: org.apache.spark.rdd.RDD[(String, utils.Trip)] = ShuffledRDD[93] at reduceByKey at <console>:43
(2) ShuffledRDD[93] at reduceByKey at <console>:43 []
 +-(2) MappedRDD[87] at keyBy at <console>:41 []
    |  MappedRDD[57] at map at <console>:43 []
    |  MappedRDD[56] at map at <console>:42 []
    |  FilteredRDD[55] at filter at <console>:41 []
    |  data/trips/* MappedRDD[54] at textFile at <console>:34 []
    |  data/trips/* HadoopRDD[53] at textFile at <console>:34 []
res87: Array[(String, utils.Trip)] = Array((Mountain View City Hall,Trip(4081,218,2013-08-29 09:38:00.0,Mountain View City Hall,27,2013-08-29
```

Took 2 seconds

You can spot the negligible time savings here, for our relatively small dataset.

__17. Broadcast joins allow you to map the key directly to perform a join, rather than shuffling it for the join. Remember that the broadcast variables are read-only. To create the broadcast variable:

```
val bcStations = sc.broadcast(stations.keyBy(_.id).collectAsMap)
```

__18. Join the trips and stations using a broadcast join:

```
val joined = trips.map(trip =>{
    (trip, bcStations.value.getOrElse(trip.startTerminal, Nil),
bcStations.value.getOrElse(trip.endTerminal, Nil))
})
```

```
val bcStations = sc.broadcast(stations.keyBy(_.id).collectAsMap)

val joined = trips.map(trip =>{
    (trip, bcStations.value.getOrElse(trip.startTerminal, Nil), bcStations.value.getOrElse(trip.endTerminal, Nil))
})

println(joined.toDebugString)

joined.take(10)
```

```
bcStations: org.apache.spark.broadcast.Broadcast[scala.collection.Map[Int,utils.Station]] = Broadcast(50)
joined: org.apache.spark.rdd.RDD[(utils.Trip, Product with Serializable, Product with Serializable)] = MappedRDD[96]
(2) MappedRDD[96] at map at <console>:50 []
 |  MappedRDD[57] at map at <console>:43 []
 |  MappedRDD[56] at map at <console>:42 []
 |  FilteredRDD[55] at filter at <console>:41 []
 |  data/trips/* MappedRDD[54] at textFile at <console>:34 []
 |  data/trips/* HadoopRDD[53] at textFile at <console>:34 []
res96: Array[(utils.Trip, Product with Serializable, Product with Serializable)] = Array((Trip(4576,63,2013-08-29 14:
Took 1 seconds
```

It takes almost no time in this case to join on the broadcast values of the keys. How does this compare to the join in the previous lab? Both with and without the partitioners.

Summary

Having completed this exercise, you should have much better knowledge of why groupByKey should be avoided if possible and be replaced with reduceByKey or aggregateByKey. However, you should understand that not all groupByKey use cases can be replaced. It depends on the dataset and the operation you need to perform on it. When possible, avoid groupByKey. Broadcast variables allows for a much more efficient join by eliminating the need for a lot of shuffles.

NOTES

NOTES

The information contained in these materials is provided for informational purposes only, and is provided AS IS without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, these materials. Nothing contained in these materials is intended to, nor shall have the effect of, creating any warranties or representations from IBM or its suppliers or licensors, or altering the terms and conditions of the applicable license agreement governing the use of IBM software. References in these materials to IBM products, programs, or services do not imply that they will be available in all countries in which IBM operates. This information is based on current IBM product plans and strategy, which are subject to change by IBM without notice. Product release dates and/or capabilities referenced in these materials may change at any time at IBM's sole discretion based on market opportunities or other factors, and are not intended to be a commitment to future product or feature availability in any way.