Soumyadeep Sinha
CS6240

Assignment 3

Combining in Spark (20 points total)

Pseudo Code:

RDD-G:

```
val count = textFile.flatMap(line => line.split(" ")).map(word => {
 val usr = word.split(",")
 (usr(1), 1)
}).groupByKey().map((x) => (x._1, x._2.sum))
count.saveAsTextFile(args(1))
```

RDD-R:

```
val count = textFile.flatMap(line => line.split(" ")).map(word => {
 val usr = word.split(",")
 (usr(1), 1)
}).reduceByKey(_ + _)
count.saveAsTextFile(args(1))
```

RDD-F:

```
val count = textFile.flatMap(line => line.split(" ")).map(word => {
 val usr = word.split(",")
 (usr(1), 1)
}).foldByKey(0)(_ + _)
count.saveAsTextFile(args(1))
```

RDD-A:

```
val count = textFile.flatMap(line => line.split(" ")).map(word => {
 val usr = word.split(",")
 (usr(1), 1)
}).aggregateByKey(0)(_ + _, _ + _)
count.saveAsTextFile(args(1))
```

DSET:

```
val sparkSession = SparkSession.builder.master("local")
.appName("TwitterCount").getOrCreate()
import sparkSession.implicits._
val data = sparkSession.read.text(args(0)).as[String]
val count = data.flatMap(line => line.split(" ")).map(word => {
 val usr = word.split(",")
 (usr(1), 1)
}).groupByKey(_._1).count()
count.coalesce(1).write.csv(args(1))
```

For each of the four RDD-based programs, report the information returned by toDebugString(). (4 points)

RDD-R:

```
(2) ShuffledRDD[4] at reduceByKey at TwitterCount.scala:51 []
 +-(2) MapPartitionsRDD[3] at map at TwitterCount.scala:48 []
    |  MapPartitionsRDD[2] at flatMap at TwitterCount.scala:48 []
    |  input MapPartitionsRDD[1] at textFile at TwitterCount.scala:32 []
    |  input HadoopRDD[0] at textFile at TwitterCount.scala:32 []
```

RDD-G:

```
(2) MapPartitionsRDD[5] at map at TwitterCount.scala:63 []
 |  ShuffledRDD[4] at groupByKey at TwitterCount.scala:63 []
 +-(2) MapPartitionsRDD[3] at map at TwitterCount.scala:60 []
    |  MapPartitionsRDD[2] at flatMap at TwitterCount.scala:60 []
    |  input MapPartitionsRDD[1] at textFile at TwitterCount.scala:32 []
```

```
  |  input HadoopRDD[0] at textFile at TwitterCount.scala:32 []
```

RDD-F:

```
(2) ShuffledRDD[4] at foldByKey at TwitterCount.scala:75 []
 +-(2) MapPartitionsRDD[3] at map at TwitterCount.scala:72 []
    |  MapPartitionsRDD[2] at flatMap at TwitterCount.scala:72 []
    |  input MapPartitionsRDD[1] at textFile at TwitterCount.scala:32 []
    |  input HadoopRDD[0] at textFile at TwitterCount.scala:32 []
```

RDD-A:

```
(2) ShuffledRDD[4] at aggregateByKey at TwitterCount.scala:86 []
 +-(2) MapPartitionsRDD[3] at map at TwitterCount.scala:83 []
    |  MapPartitionsRDD[2] at flatMap at TwitterCount.scala:83 []
    |  input MapPartitionsRDD[1] at textFile at TwitterCount.scala:32 []
    |  input HadoopRDD[0] at textFile at TwitterCount.scala:32 []
```

For the DataSet-based program, <u>report</u> the logical and physical plans returned by explain(). (2 points)

DSET:

```
== Physical Plan ==
*(3) HashAggregate(keys=[value#16], functions=[count(1)])
+- Exchange hashpartitioning(value#16, 200)
   +- *(2) HashAggregate(keys=[value#16], functions=[partial_count(1)])
      +- *(2) Project [value#16]
                              +- AppendColumnsWithObject <function1>, [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, assertnotnull(input[0, scala.Tuple2, true])._1, true, false) AS _1#12, assertnotnull(input[0, scala.Tuple2, true])._2 AS _2#13], [staticinvoke(class org.apache.spark.unsafe.types.UTF8String, StringType, fromString, input[0, java.lang.String, true], true, false) AS value#16]
            +- MapElements <function1>, obj#11: scala.Tuple2
              +- MapPartitions <function1>, obj#6: java.lang.String
                +- DeserializeToObject value#0.toString, obj#5: java.lang.String
                          +- *(1) FileScan text [value#0] Batched: false, Format: Text, Location: InMemoryFileIndex[file:/Users/soumyadeepsinha/Desktop/Spark-Demo/input],  PartitionFilters: [], PushedFilters: [], ReadSchema: struct<value:string>
```

()2

Based on this information, state clearly which of the programs performs aggregation before shuffling, and which does not. (4 points)

Basing my observation on the above debugstring and explain string, I can conclude that for each RDD operation, shuffle is performed before aggregation. However, for the Dataframe We cannot see any shuffling taking place.

Join Implementation (48 points total)

Show the pseudo-code for all four Spark Scala programs (RS-R, RS-D, Rep-R, Rep-D), including Max-filter functionality if you used it. Since many Scala functions are similar to pseudo-code, you may copy-and-paste them here whenever appropriate. (20 points)

RS-R:

Example:

Lets take a simple graph:

1,2

2,3

3,1

According to the code we will get From table as :

(1,2)

(2,3)

(3,1)

And the to table as:

(2,1)

(3,2)

(1,3)

Now lets join the from and the to table to get the two path.

(1,(2,3))

(2,(3,1))

(3,(1,2))

Once we get the two path we can create a reverse two path as

((2,3),1)

((3,1),2)

((1,2),3)

And the reverse path as:

((1,2),null)

((2,3),null)

((3,1),null)

We can now join the reversetwopath and the reversepath to get the triangles:

((2,3),(1,null))

((3,1),(2,null))

((1,2),(3,null))

This gives us the three paths for the same triangle so we can determine the number of triangle by simple dividing the count by 3.

```
val textFile = textFile1.flatMap(line => line.split(" ")).filter(word => {
 val usr = word.split(",")
 usr(0).toInt < args(2).toInt && usr(1).toInt < args(2).toInt
})
val from = textFile.flatMap(line => line.split(" ")).map(
 word => {
   val usr = word.split(",")
   (usr(0), usr(1))
 }
)
val to = textFile.flatMap(line => line.split(" ")).map(
 word => {
   val usr = word.split(",")
   (usr(1), usr(0))
 }
)
val twoPath = from.join(to)
val twoPathReverse = twoPath.map(x => {
 ((x._2._1, x._2._2), x._1)
})
```

```
val reverseFollower = textFile.map(word => {
 val usr = word.split(",")
 ((usr(0), usr(1)), null)
})

val triangle = twoPathReverse.join(reverseFollower)
```

RS-D:

Lets take a simple graph:

1,2

2,3

3,1

So the table would look like:

follower	user
1	2
2	3
3	1

Now lets join the table with itself where the user is the follower which would give us:

follower	user	follower1	user1
2	3	3	1
3	1	1	2
1	2	2	3

This gives us the two path. Now we need to join it again with the first table such that the user1 in two path is the follower in the first table and the follower in the two path is the user in the first table.

This will give us the three path which we will divide by 3 to get the user count.

```
val sparkSession = SparkSession.builder.
 master("local")
 .appName("TwitterCount")
 .getOrCreate()
import sparkSession.implicits._
val data1 = sparkSession.read.text(args(0)).as[String]
val data = data1.flatMap(line => line.split(" ")).filter(word => {
 val usr = word.split(",")
 usr(0).toInt < args(2).toInt && usr(1).toInt < args(2).toInt
})
val user = data.map(word => {
 val usr = word.split(",")
 (usr(0), usr(1))
})
val table = user.toDF("follower", "user")
val duplicate = table.toDF("follower1", "user1")
val twoPath = table.join(duplicate, table("user") <=> duplicate("follower1"))
twoPath.show()
val newTable = table.toDF("follower2", "user2")
val threePath = twoPath.join(newTable, twoPath("user1") === newTable("follower2") &&
twoPath("follower") === newTable("user2") )
```

Rep-R:

Lets take a simple graph:

1,2

2,3

3,1

First what we need to broadcast is the a map that contains the list of neighbours.

I -> [2]

2 -> [3]

3 -> [1]

Now we will create a map and loop over each user/ the second part of the value for example if the edge is (1,2) we will check 2’s neighbour’s which is 3 and see if its neighbour is 1.

If it matches with the source then we got a triangle. Then we will simply divide it by 3 to get the number of triangles.

```
val sparkSession = SparkSession.builder.
 master("local")
 .appName("TwitterCount")
 .getOrCreate()

val textFile = textFile1.flatMap(line => line.split(" ")).filter(word => {
 val usr = word.split(",")
 usr(0).toInt < args(2).toInt && usr(1).toInt < args(2).toInt
})

val table = textFile.map(edges => {
 val users = edges.split(",");
 (users(0), users(1))
}).groupBy(_._1).mapValues(_.map(_._2).toList)

val dc = table.collectAsMap()
val distributedCache = sparkSession.sparkContext.broadcast(dc)

val countT = textFile.flatMap(line => line.split(" ")).map(word => {
 val usr = word.split(",")

 val follower = usr(0)
 val user = usr(1)

 var c = 0
 if (distributedCache.value.get(user) != None) {
   val neigh = distributedCache.value(user)

   neigh foreach (n => {
          if (distributedCache.value.get(n) != None) {
            val nn = distributedCache.value(n)
            if (nn.contains(follower)) {
              c = c + 1
            }
          }
        })


      }
```

Rep-D:

Lets take a simple graph:

1,2

2,3

3,1

So the table would look like:

The only thing we are doing extra here is we are broadcasting the edges and the converting it to DF so that we can perform join operation.

follower	user
1	2
2	3
3	1

Now lets join the table with itself where the user is the follower which would give us:

follower	user	follower1	user1
2	3	3	1
3	1	1	2
1	2	2	3

This gives us the two path. Now we need to join it again with the first table such that the user1 in two path is the follower in the first table and the follower in the two path is the user in the first table.

This will give us the three path which we will divide by 3 to get the user count.

```
val sparkSession = SparkSession.builder.
 master("local")
 .appName("TwitterCount")
 .getOrCreate()

import sparkSession.implicits._
val data1 = sparkSession.read.text(args(0)).as[String]
val data = data1.flatMap(line => line.split(" ")).filter(word => {
 val usr = word.split(",")
 usr(0).toInt < args(2).toInt && usr(1).toInt < args(2).toInt
})
val users = data.map(word => {
 val usr = word.split(",")
 (usr(0), usr(1))
})
val broadcast = sparkSession.sparkContext.broadcast(users)
val table = broadcast.value.toDF("follower", "user")
val duplicate = broadcast.value.toDF("follower1", "user1")
val twoPath = table.join(duplicate, table("user") <=> duplicate("follower1"))
//twoPath.show()
val newTable = table.toDF("follower2", "user2")
val threePath = twoPath.join(newTable, twoPath("user1") === newTable("follower2") &&
twoPath("follower") === newTable("user2") )
```

Results:

I was not able to run the program in 8 and above number of machines as AWS was not able to allocate machines that were required. Hence, I have gone with 6 and 7 machines to perform my analysis.

6 cheap machines = (1 master 5 workers)

7 cheap machines = (1 master 6 workers)

Configuration	**Small Cluster Result**	**Large Cluster Result**
RS-R, MAX = 10000	Running time: 92 sec, Triangle count: 520296	Running time: 80 sec, Triangle count: 520296
RS-D, MAX = 10000	Running time: 132 sec, Triangle count: 520296	Running time: 130 sec, Triangle count: 520296
Rep-R, MAX = 10000	Running time: 82 sec, Triangle count: 520296	Running time: 78 sec, Triangle count: 520296
Rep-D, MAX = 10000	Running time: 120 sec, Triangle count: 520296	Running time: 118 sec, Triangle count: 520296

Reference:

http://www.ccs.neu.edu/home/mirek/classes/CS6240-stable/index.htm
https://community.cloudera.com/t5/Support-Questions/Can-I-join-2-dataframe-with-condition-in-column-value/td-p/168264
https://sparkbyexamples.com/spark/spark-sql-dataframe-inner-join/