CS 6240 : LScale Parallel Processing Homework 3

Dharmish Shah

COMBINING IN SPARK

- **RDD-G**
  - This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using groupByKey, followed by the corresponding aggregate function.
  - Pseudo Code -

```
val textFile = sc.textFile(input)val counts = textFile.flatMap(line => line.split(" "))
.map(word => {
        val users = word.split(",")
        (users(1), 1)
})
.groupByKey().map(word => (word._1, word._2.sum))
counts.saveAsTextFile(output)
```

- **RDD-R**
  - This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using reduceByKey.
  - Pseudo-Code -

```
val textFile = sc.textFile(input)
val counts = textFile.flatMap(line => line.split(" "))
 .map(word => {
        val users = word.split(",")
        (users(1), 1)
 })
 .reduceByKey(_ + _)
counts.saveAsTextFile(output)
```

- **RDD-F**
  - This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using foldByKey.
  - Pseudo-Code -

```
val textFile = sc.textFile(input)
```

```
val counts = textFile.flatMap(line => line.split(" "))
 .map(word => {
            val users = word.split(",")
            (users(1), 1)
 })
 .foldByKey(10)(_ + _)
counts.saveAsTextFile(output)
```

- **RDD-A**
    - This program only uses RDD and pair RDD, but not DataSet or DataFrame. The grouping and aggregation step must be implemented using aggregateByKey.
    - Pseudo-code -

```
val textFile = sc.textFile(input)
def seqOp = (accumulator: Int, element: (Int)) =>
 accumulator + element
//Combiner Operation : Finding Maximum Marks out Partition-Wise Accumulators
def combOp = (accumulator1: Int, accumulator2: Int) =>
                accumulator1 + accumulator2
val counts = textFile.flatMap(line => line.split(" "))
 .map(word => {
            val users = word.split(",")
            (users(1), 1)
 }).aggregateByKey(0)(seqOp, combOp)
counts.saveAsTextFile(output)
```

- **DSET**
    - The grouping and aggregation step must be implemented using DataSet, with groupBy on the appropriate column, followed by the corresponding aggregate function.
    - Pseudo-Code -

```
val data = sparkSession.read.text(input).as[String]
  val words = data.flatMap(value => value.split(" "))
  val mapWords = words.map(word => {
            val users = word.split(",")
            (users(1), 1)
  })
  val groupedWords = mapWords.groupBy($"_1").sum()
  groupedWords.coalesce(1).write.csv(output)
```

Using log files from successful runs and Scala functions such as toDebugString() and explain(), find out which of the different programs performs aggregation before data is shuffled, i.e., the equivalent of MapReduce's in-Mapper combining.

Output of toDebugString for 4 RDDs

In Spark, all dependencies between RDDs are logged in a graph, giving us an execution plan of RDDs. toDebugString method is used to get these details.

- **RDDG**
  - **Output -**
    2020-02-28 20:11:51 INFO FileInputFormat:256 - Total input files to process : 1
    (40) MapPartitionsRDD[5] at map at TwitterCountMain.scala:48 []
     | **ShuffledRDD[4] at groupByKey** at TwitterCountMain.scala:48 []
     +-(40) MapPartitionsRDD[3] at map at TwitterCountMain.scala:44 []
       | MapPartitionsRDD[2] at flatMap at TwitterCountMain.scala:43 []
       | input MapPartitionsRDD[1] at textFile at TwitterCountMain.scala:42 []
       | input HadoopRDD[0] at textFile at TwitterCountMain.scala:42 []

- **RDDR**
  - **Output -**
    2020-02-28 21:50:47 INFO FileInputFormat:256 - Total input files to process : 1
    (40) **ShuffledRDD[4] at reduceByKey** at TwitterCountMain.scala:60 []
     +-(40) MapPartitionsRDD[3] at map at TwitterCountMain.scala:56 []
       | MapPartitionsRDD[2] at flatMap at TwitterCountMain.scala:55 []
       | input MapPartitionsRDD[1] at textFile at TwitterCountMain.scala:54 []
       | input HadoopRDD[0] at textFile at TwitterCountMain.scala:54 []

- **RDDF**
  - **Output -**
    2020-02-28 21:53:50 INFO FileInputFormat:256 - Total input files to process : 1
    (40) **ShuffledRDD[4] at foldByKey** at TwitterCountMain.scala:72 []
     +-(40) MapPartitionsRDD[3] at map at TwitterCountMain.scala:68 []
       | MapPartitionsRDD[2] at flatMap at TwitterCountMain.scala:67 []
       | input MapPartitionsRDD[1] at textFile at TwitterCountMain.scala:66 []
       | input HadoopRDD[0] at textFile at TwitterCountMain.scala:66 []

- **RDDA**
  - **Output -**
    2020-02-28 22:00:34 INFO FileInputFormat:256 - Total input files to process : 1

(40) **ShuffledRDD[4] at aggregateByKey** at TwitterCountMain.scala:93 [
+-(40) MapPartitionsRDD[3] at map at TwitterCountMain.scala:89 []
| MapPartitionsRDD[2] at flatMap at TwitterCountMain.scala:88 []
| input MapPartitionsRDD[1] at textFile at TwitterCountMain.scala:78 []
| input HadoopRDD[0] at textFile at TwitterCountMain.scala:78 []

Plans returned by explain() for DSET

Explain() method prints the logical and physical execution plan of dataset on console. It ran on a local machine with MAX = 1000.

Output -

```
== Physical Plan ==
*(3) HashAggregate(keys=[_1#12], functions=[sum(cast(_2#13 as bigint))])
+- Exchange hashpartitioning(_1#12, 200)
  +- *(2) HashAggregate(keys=[_1#12], functions=[partial_sum(cast(_2#13 as bigint))])
    +- *(2) SerializeFromObject [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]
      +- *(2) 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:/home/dharmish/work/large-scale-parallel-processing/homework/
hwk3/Spark-Tw..., PartitionFilters: [], PushedFilters: [], ReadSchema:
struct<value:string>
```

Based on the above highlighted information, we can conclude that shuffling is always performed before aggregation. Shuffled RDDs are used to perform aggregate function on given input files. Whereas, no shuffling operation was performed using datasets and hence there is no precedence of shuffling over aggregation. We can explicitly create a shuffled dataframe using orderBy method on a dataframe.

JOIN IMPLEMENTATION

- **RS-R**

```
  val textFile = sc.textFile(input)
// getting users only upto MAX_VALUE
  val maxfilter = textFile.filter(edges =>{
    val users = edges.split(",");
    val fromUser = users(0).toInt
    val toUser = users(1).toInt
    (fromUser < maximum && toUser < maximum)
  })
// making a map of followers and following where fromUser is the key
val fromUsers = maxfilter.map(edges =>{
    val users = edges.split(",");
    (users(0),users(1))
  })

// making a map of followers and following where toUser is the key
  val toUsers = maxfilter.map(edges =>{
    val users = edges.split(",");
    (users(1),users(0))
  })
// joining to find 2-Paths
  val twoPath = fromUsers.join(toUsers)
// processing the 2-Paths to match with last edge to form a triangle
  val processedtwoPath = twoPath.map(twoPath =>{
     val secondPath = twoPath._2.toString().replace("(","").replace(")","")
       .split(",")
    ((secondPath(0),secondPath(1)),1)
  })
//  list of edges which has last missing edge of triangle
  val fromUsersAsKey = maxfilter.map(edges =>{
    val users = edges.split(",");
    ((users(0),users(1)),1)
  })
// finding all edges which completes the triangle
  val threePath = processedtwoPath.join(fromUsersAsKey)
// giving final count of triangles
  var finalOutput = threePath.map(count =>{
    ("RDD Reduce Join",1)
```

```
  }).reduceByKey(_ + _).map(count => { (count._1,count._2/3) })
  finalOutput.coalesce(1).saveAsTextFile(output)
 }
```

- **RS-D**

// reading from input file

```
val data = sparkSession.read.text(input).as[String]
```

// getting users only upto MAX_VALUE

```
  val maxfilter = data.filter(edges =>{
          val users = edges.split(",");
          val fromUser = users(0).toInt
          val toUser = users(1).toInt
          (fromUser < maximum && toUser < maximum)
  })
```

// making a map of followers and following where fromUser is the key

```
  val fromUsers = maxfilter.map(edges =>{
          val users = edges.split(",");
          (users(0),users(1))
  })
```

// making a map of followers and following where toUser is the key

```
  val toUsers = maxfilter.map(edges =>{
          val users = edges.split(",");
          (users(1),users(0))
  })
  // finding two paths using dataframes join
  var fromDF = fromUsers.toDF("twoPathNode","source")
  var toDF = toUsers.toDF("twoPathNode","destination")
  var twoPath = fromDF.join(toDF,"twoPathNode")
  // finding a triangle which completes from two paths found using dataframes join
  var fromLastPath = fromUsers.toDF("source","destination")
  var threePath = twoPath.toDF().join(fromLastPath,Seq("source","destination"))
  //threePath.coalesce(1).write.csv(output)
```

// writing final count of triangles

```
  var finalOutput = threePath.rdd.map(count =>{
       ("DF Reduce Join",1)
  }).reduceByKey(_ + _).map(count => { (count._1,count._2/3) })

  finalOutput.coalesce(1).saveAsTextFile(output)
```

- **Rep-R**

// reading from input file

```
  val textFile = sc.textFile(input)
// getting users only upto MAX_VALUE
  val maxfilter = textFile.filter(edges =>{
          val users = edges.split(",");
          val fromUser = users(0).toInt
          val toUser = users(1).toInt
          (fromUser < maximum && toUser < maximum)
  })
// making a map of followers and following where fromUser is the key
  val fromUsers = maxfilter.map(edges =>{
   val users = edges.split(",");
   (users(0),users(1))
  }).groupBy(_._1).mapValues(_.map(_._2).toList)
// broadcasting the map on spark context
  val smallRDDLocal = fromUsers.collectAsMap()
  val broadcastedMap = sc.broadcast(smallRDDLocal)

  val toUsers = maxfilter.map(edges =>{
   val users = edges.split(",");
   (users(1),users(0))
  })
  val addToCounts = (n: Int, v: Int) => n + v
  val sumPartitionCounts = (p1: Int, p2: Int) => p1 + p2
  val twoPath = toUsers.map(word =>{
   var count = 0
//using broadcasted map to find all triangles
    if(broadcastedMap.value.get(word._1) != None){
// comparing the values for find existing two Paths
     val path2 = broadcastedMap.value.get(word._1).get
     for(a <- path2){
       if(broadcastedMap.value.get(a) != None){
//  checking for final edge which completes the triangle using broadcasted map
       val path3 = broadcastedMap.value.get(a).get
       for(b <- path3) {
        if(b == word._2) {
         println("triangle123")
         count += 1}}}}}
   ("Triangle",count)
```

```
  }).filter(a => {
    (a._2 > 0)
  }).aggregateByKey(0)(addToCounts, sumPartitionCounts)
```
// **writing total number of triangles in output file**
```
  var totalTriangles = twoPath.map(count => {
    ("RDD Replicated Join",count._2.toInt/3)
  })
  totalTriangles.coalesce(1).saveAsTextFile(output)
 }
```

- **Rep-D**

```
val data = sparkSession.read.text(input).as[String]
```
// getting users only upto MAX_VALUE
```
  val maxfilter = data.filter(edges =>{
    val users = edges.split(",");
    val fromUser = users(0).toInt
    val toUser = users(1).toInt
    (fromUser < maximum && toUser < maximum)
  })
```
// making a map of followers and following where fromUser is the key
```
  val fromUsers = maxfilter.map(edges =>{
    val users = edges.split(",");
    (users(0),users(1))
  })

  val toUsers = maxfilter.map(edges =>{
    val users = edges.split(",");
    (users(1),users(0))
  })
```
// **broadcast the maps**
```
  val broadcastedFromUser = sc.broadcast(fromUsers)
  val broadcastedToUser = sc.broadcast(toUsers)
```

// finding two paths using dataframes join and broadcasted map
```
  var fromDF = broadcastedFromUser.value.toDF("twoPathNode","source")
  var toDF = broadcastedToUser.value.toDF("twoPathNode","destination")
  var twoPath = fromDF.join(toDF,"twoPathNode")
```

// finding a triangle which completes from two paths found using dataframes join and broadcasted maps
```
  var fromLastPath = broadcastedFromUser.value.toDF("source","destination")
```

```
var threePath = twoPath.toDF().join(fromLastPath,Seq("source","destination"))
// writing total number of triangles in output file
var finalOutput = threePath.rdd.map(count =>{
  ("DF Replicated Join",1)
}).reduceByKey(_ + _).map(count => { (count._1,count._2/3) })
finalOutput.coalesce(1).saveAsTextFile(output)
```

PERFORMANCE ANALYSIS

The Reduce and replicated join was tried to run on AWS 9 machines and 8 machines for both m4.xlarge and m5.xlarge. But, unfortunately, AWS never started the program in the cluster and was left in infinite state. The experiment for each run was tried for an average 18-20 minutes and each time the cluster was stuck in “Starting” state.
So, after multiple trials, I decided to run on the same configuration as it was in Homework 2.

So, it ran the program on AWS with following configuration :
1) Small Cluster - 6 cheap machine instance 1 master and 5 workers on m5.xlarge instance
2) Large Cluster - 7 cheap machine instance 1 master and 6 workers on m5.xlarge instance

Configuration	**Small Cluster Result**	**Large Cluster Result**
RS-R, MAX = 10000	Running time: 112 secs Triangle count: 520296	Running time: 114 secs Triangle count: 520296
RS-D, MAX = 10000	Running time: 80 secs Triangle count: 520296	Running time: 84 secs Triangle count: 520296
Rep-R, MAX = 10000	Running time: 86 secs Triangle count: 520296	Running time: 86 secs Triangle count: 520296
Rep-D, MAX = 10000	Running time: 88 secs Triangle count: 520296	Running time: 84 secs Triangle count: 520296

REFERENCES

- https://spark.apache.org/docs/0.9.1/scala-programming-guide.html
- https://spark.apache.org/docs/2.2.0/rdd-programming-guide.html
- https://docs.scala-lang.org/tour/tour-of-scala.html
- https://docs.scala-lang.org/tutorials/scala-for-java-programmers.html
- http://spark.apache.org/examples.html
- https://spark.apache.org/docs/0.9.1/scala-programming-guide.html#resilient-distributed-datasets-rdds