DISTRIBUTED DATA MANAGEMENT

Joining two Relationships in Scala and Spark 2.2.0

The purpose of this project is to create a program in Scala and Spark 2.2.* which will compute the following relational operation in many different ways: **R(a,b)** ⋈ **S(a,c)**. The given input data is a text file and each of its lines have the following format: *<RelationSymbol>,<key>,<value>.* The RelationSymbol can be either 'R' or 'S', the key value is an Integer and the value is a String. The output of the program would be a message on the screen for each of the solutions implemented that will contain the time (in ms.) that was needed for this solution to run and the number of results. Additionally, the joined relation should be written in different output folders in the disk.

[1] THE RDD WAY

The first way to go is using the RDD structure of the Spark framework. In this way, we need to read the input file using the SparkContext into an RDD. Then, we have to split this RDD into two new RDDs that each of them will only contain records from one relation (either "R" or "S") and then we will use the built-in method join() that RDDs have. Following this logic, I reach to the following solution:

```
def joinUsingRDDs(sc: SparkContext, inputFile: String, outputDir: String): Unit = {

  // Reading the input file into an RDD
  val inputRDD = sc.textFile(inputFile, 2)

  // Splitting the first RDD into two new RDDs, one for each relation
  val s_relation = inputRDD.filter(line => line.charAt(0) == 'S')
                                            .map(line => line.split(“,"))
                                            .map(line => (line(1), line(2)))

  val r_relation =inputRDD.filter(line => line.charAt(0) == 'R')
                                            .map(line => line.split(","))
                                            .map(line => (line(1), line(2)))

  // From this point we start to count the time spent on the join
  val startTime = DateTime.now(DateTimeZone.UTC).getMillis()

  // This is where we tell Spark to join the two relations
  val joined = r_relationship.join(s_relationship)

  // Printing the following text to screen and finally saving the joined result in the output
  // directory.
  val numOfresults = joined.count()
  val totalTime = (DateTime.now(DateTimeZone.UTC).getMillis() - startTime)
  println("[RDD join] \t Time: " + totalTime + " miliseconds \t “ +
           “Number of results: " + numOfresults)

  joined.saveAsTextFile(outputDir)
```

[2] THE DataFrame WAY

The second way to go is using the DataFrame structure of the Spark framework. In this way, we need to read the input file using the SparkSession into a DataFrame. Then, we have to split this DataFrame into two new DataFrames that each of them will only contain records from one relation (either “R” or “S”) and then we will use the built-in method join() that DataFrames have. As you can understand the logic is very similar to the one of the RDD-solution. Bellow, is the solution using DataFrames:

```
def joinUsingDFs(spark: SparkSession, inputFile: String, outputDir: String): Unit = {

  // Reading the input file into a DataFrame
  val df = spark.sqlContext.read.option("header", "false").option("delimiter", ",").csv(inputFile)

  // “_c0”, “_c1” and “_c2” are the default names that Spark gives to each of the 3 columns
  val cols = Seq[String]("_c1", “_c2")

  // Splitting the first DataFrame into two new DataFrames, one for each relation. Also the
     columns “_c1” and “_c2” are renamed to “key” and “value” respectively.
  val s_df = df.filter("_c0 == 'S'").select(cols.head, cols.tail: _*).toDF(Seq("key", "valueS"): _*)
  val r_df = df.filter("_c0 == 'R'").select(cols.head, cols.tail: _*).toDF(Seq("key", "valueR"): _*)

  // From this point we start to count the time spent on the join
  val startTime = DateTime.now(DateTimeZone.UTC).getMillis()

  // This is where we tell Spark to join the two DataFrames
  val joined = r_df.join(s_df, Seq("key"))

  // Printing the following text to screen and finally saving the joined result in the output
  // directory.
  val numOfresults = joined.count()
  val totalTime = (DateTime.now(DateTimeZone.UTC).getMillis() - startTime)
  println("[RDD join] \t Time: " + totalTime + " miliseconds \t “ +
            “Number of results: " + numOfresults)
  joined.write.csv(outputDir)

}
```

[3] THE [DataFrame AND SQL] WAY

The third way is using the DataFrame structure of the Spark framework, but this time along with the SQL method of SparkSession. Again, at first we need to read the input file using the SparkSession into a DataFrame. Then, we have to split this DataFrame into two new DataFrames that each of them will only contain records from one relation (either “R” or “S”) and then we will use the SQL method provided by Spark to enter our SQL query which will join the two relations.

```
def joinUsingSQL(spark: SparkSession, inputFile: String, outputDir: String): Unit = {

  // Reading the input file into a DataFrame
  val df = spark.sqlContext.read.option("header", "false").option("delimiter", ",").csv(inputFile)

  // "_c0", "_c1" and "_c2" are the default names that Spark gives to each of the 3 columns
  val cols = Seq[String]("_c1", "_c2")

  // Creating two TempViews, one for each relation. Also the  columns "_c1" and "_c2" are
     renamed to "key" and "value" respectively.
  df.filter("_c0 == 'S'").select(cols.head, cols.tail: _*).toDF(Seq("key", "valueS"): _*)
          .createOrReplaceTempView("s_table")
  df.filter("_c0 == 'R'").select(cols.head, cols.tail: _*).toDF(Seq("key", "valueR"): _*)
          .createOrReplaceTempView("r_table")

  // From this point we start to count the time spent on the join
  val startTime = DateTime.now(DateTimeZone.UTC).getMillis()

  // We join the two tables using an SQL query and store the result in the joined DataFrame
  val joined = spark.sql("SELECT s_table.key, s_table.valueS, r_table.valueR
                                FROM s_table
                                INNER JOIN r_table ON s_table.key = r_table.key")

  // Printing the following text to screen and finally saving the joined result in the output
  // directory.
  val numOfresults = joined.count()
  val totalTime = (DateTime.now(DateTimeZone.UTC).getMillis() - startTime)
  println("[RDD join] \t Time: " + totalTime + " miliseconds \t " +
           "Number of results: " + numOfresults)
  joined.write.csv(outputDir)

}
```

Note: We could have used SQL to also split the DataFrame, but since we start to count time just before we join the two relations, it would have made no difference

[4] THE DataSet WAY

The fourth way and the last built-in way we are going to explore is using the DataSet structure of the Spark framework. At first we need to read the input file using the SparkSession into a DataSet. Then, we have to split this DataSet into two new DataSets that each of them will only contain records from one relation (either "R" or "S") and then we will use the joinWith() method of DataSet to join the two sets.

The logic is again similar to what we have already seen. The only difference is the data structures that are used in each case.

```
def joinUsingDS(spark: SparkSession, inputFile: String, outputDir: String): Unit = {

  // We need to import the Spark’s implicits for this solution to work
  import spark.implicits._

  // Reading the input file into a DataSet
  val inputData = spark.read.option("header", "false").option("delimiter", ",").textFile(inputFile)

  // Splitting the first DataSet into two new DataSets, one for each relation.
  val s_relation = inputData.filter(line => line.charAt(0) == 'S')
                                            .map(line => line.split(","))
                                            .map(line => (line(1), line(2)))

  val r_relation = inputData.filter(line => line.charAt(0) == 'R')
                                            .map(line => line.split(","))
                                            .map(line => (line(1), line(2)))

  // From this point we start to count the time spent on the join
  val startTime = DateTime.now(DateTimeZone.UTC).getMillis()

  // We join the two relations using the joinWith method on the equality of their two columns
     “_1”.
  val joined = s_relation
                      .joinWith(r_relation, s_relation("_1") === r_relation("_1"))


  // Printing the following text to screen and finally saving the joined result in the output
  // directory.
  val numOfresults = joined.count()
  val totalTime = (DateTime.now(DateTimeZone.UTC).getMillis() - startTime)
  println("[RDD join] \t Time: " + totalTime + " miliseconds \t “ +
          “Number of results: " + numOfresults)
  joined.write.json(outputDir)
```

[5] THE MANUAL WAY

The this way we will read the data into a RDD structure and then without using any built-in method we will group the records by their keys, then for each key we will take all the values, filter them by their relation symbol column and combine each record of the one relation with all the records of the other.

This is the simplest manual way to perform a join and it would be interesting to see how efficient this way is compared to the other built-in ways.

```
def joinManually(sc: SparkContext, inputFile: String, outputDir: String): Unit = {

  // Reading the input file into a DataSet
  val myData = sc.textFile(inputFile,2)

  // From this point we start to count the time spent on the join
  val startTime = DateTime.now(DateTimeZone.UTC).getMillis()

  // In the following lines of code, we take the input and transform it to have as key the join key
  // and as value the whole tuple. Then we group all records by their key. Then by filtering two
  // times we first keep all the records of the relation 'R' and then cross them with all the
  // records of the relation 'S'
  val joined = myData.map(line => line.split(","))
          .map(line => (line(1), (line(0), line(1), line(2))))
          .groupByKey()
          .flatMapValues(tuples =>
                  tuples.filter(tuple => tuple._1 == "R")
                                .flatMap(tupleR =>
                                        tuples.filter(tuple => tuple._1 == "S")
                                                .map(tupleS => (tuple._3,tuple3._3))))


  // Printing the following text to screen and finally saving the joined result in the output
  // directory.
  val numOfresults = joined.count()
  val totalTime = (DateTime.now(DateTimeZone.UTC).getMillis() - startTime)
  println("[RDD join] \t Time: " + totalTime + " miliseconds \t " +
          "Number of results: " + numOfresults)
  joined.saveAsTextFile(outputDir)
```

[6] THE MANUAL WAY #2

There is another way to handle the join task using a different optimised technique. We are going to use the value-to-key conversion design pattern. That means that in the mapper phase, instead of simply emitting the join key as the intermediate key, we instead create a composite key consisting of the join key and the tuple id (from either R or S). At this point, we must define the sort order of the keys to first sort by the join key, and then sort all tuple ids from R before all tuple ids from S - and we are lucky because this is what Spark does by default. The tricky part is that we must define a custom partitioner to pay attention to only the join key, so that all composite keys with the same join key arrive at the same reducer.

Following this procedure results in having all the tuples from S with the same join key to be encountered first, which the reducer can keep in memory. Then, as the reducer processes each tuple from S, it is crossed with all the tuples from R, which will finally give us the desired result.

Of course, we are assuming that the tuples from R (with the same join key) will fit into memory, which is a limitation of this algorithm.

```
def joinManually(sc: SparkContext, inputFile: String, outputDir: String): Unit = {

  // Reading the input file into a DataSet
  val myData = sc.textFile(inputFile,2)

  // From this point we start to count the time spent on the join
  val startTime = DateTime.now(DateTimeZone.UTC).getMillis()

  // In the followin lines of code we take the input and in the map phase we emit key-value pairs
  // with a composite key consisted of the join-key and a letter for the relation and as value the
  // rest of it tuple. Then we repartition the tuples so as the records with the same joining-key to
  // be in the same reducer and then we sort these records in each partition separately. Then,
  // we only keep the joining key as the key and we group the records by it. Now, for each key
  // we will have a 'stream' of records where the first n records would be from relation 'R' and
  // the rest of the relation 'S'. So we keep the first of them in a List, and when the records from
  // 'S' appears we cross each of them with all the elements of the List. Finally, the last 3 lines
  // make the output to have the same format with all the others implementations presented.
  val joined = myData.map(line => line.split(","))
          .map(line => ((line(1), line(0)), (line(0), line(2))))
          .repartitionAndSortWithinPartitions(new CustomPartitioner(2))
          .map(line => (line._1._1, line._2))
          .groupByKey()
          .flatMap(tuple => {
                  var listOfR : List[String] = List()
                  tuple._2.map(record => {
                          if (record._1 == "R") {
                                  listOfR = listOfR :+ record._2
                          } else
                                  listOfR.map(valueR => (tuple._1, (valueR, record._2)))
                          })
                  })
          .filter(_.isInstanceOf[List[(String, (String, String))]])
          .map(_.asInstanceOf[List[(String, (String, String))]])
          .flatMap(x => x)

  // Printing the following text to screen and finally saving the joined result in the output
  // directory.
  val numOfresults = joined.count()
  val totalTime = (DateTime.now(DateTimeZone.UTC).getMillis() - startTime)
  println("[RDD join] \t Time: " + totalTime + " miliseconds \t " +
          "Number of results: " + numOfresults)
  joined.saveAsTextFile(outputDir)

}

// This is the CustomPartitioner I used to partition the data according to the desired key
class CustomPartitioner(numParts: Int) extends Partitioner {
  override def numPartitions: Int = numParts
  override def getPartition(key: Any): Int = {
    key match {
      case tuple @ (a: String, b: String) => Integer.valueOf(a) % 2
    }
  }
}
```

<————————————CONCLUSIONS————————————>

Running all the implementations described above, for the given input text file will produces this output:

```
                                   Output
-------------------------------------------------------------------------------

[RDD join]                 Time: 2294 ms          Number of results: 2499886
[DataFrame SQL join]       Time: 2038 ms          Number of results: 2499886
[DataFrame join]           Time: 1223 ms          Number of results: 2499886
[DataSets join]            Time: 1311 ms          Number of results: 2499886
[Manual join]              Time: 1479 ms          Number of results: 2499886
[Optimised manual join]    Time: 3529 ms          Number of results: 2499886
```

As we can see the slowest method was the RDD structure which the expected result. RDDs is an old structure of Spark framework, a stable one of course, but on the other hand there are newer structures like DataFrames and DataSets that use better optimisations under the hood.

The second slowest method was the usage of SQL to join the two tables. This method was not so slow and it had small differences with the best solution. Maybe this is a good indication that using SQL on DataFrames may not be the most optimal way to perform actions on data, but still Spark gives its users a decent way to use the well-known SQL queries over one of his best structures.

DataFrame and DataSet data structures are the newer structures of Spark that use a lot of optimisations to achieve the best possible performance. As we can see on our output, in our example DataFrames performed better than the DataSets but this may not be always true.

Surprisingly, our simple manual way to join the two tables was very close to the best solution. Maybe just filtering an RDD to split it into two and then directly cross each record of the first one with all the records of the second one is very close to the optimal solution. Maybe, this logic becomes more slow as the data become bigger and does not fit in the main memory. At this scenario, the optimisations of Spark built-in data structures may be more efficient.

Last but not least we have the optimised solution that works as we described above. This solution performed really bad needing twice the time the other methods needed. This slowness may have occurred mainly due to the really high cost of transferring data between the reducers and due to the necessary transformations that transformed our joined result in the desired form.