SecondJob

July 7, 2025

1 Big Data project A.Y. 2024-2025

1.1 Members

- Giovanni Antonioni
- Luca Rubboli - 0001083742

1.2 Second job

[1]:
```scala
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder
  .appName("Second job with RDDs")
  .getOrCreate()


val sc = spark.sparkContext
```

```
Intitializing Scala interpreter …
```

```
Spark Web UI available at http://localhost:4040
SparkContext available as 'sc' (version = 3.5.1, master = local[*], app id =␣
 ↪local-1751809290250)
SparkSession available as 'spark'
```

[1]:
```
import org.apache.spark.sql.SparkSession
spark: org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@7f2fa690
sc: org.apache.spark.SparkContext = org.apache.spark.SparkContext@1c395d8c
```

1.3 Definition of parameters for the job

Here are defined the variables used for the snippet.

[2]:
```scala
val datasetName = "green"
val datasetFolder = "./dataset"
val outputDir = s"/output/secondJobRDD/$datasetName"
val pathToFiles = s"$datasetFolder/$datasetName"
val weatherData = s"$datasetFolder/weather/weather_data_2017_2024.csv"
val weatherWmoLookup = s"$datasetFolder/weather/wmo_lookup_codes.csv"
```

[2]:
```
datasetName: String = green
datasetFolder: String = ./dataset
outputDir: String = /output/secondJobRDD/green
pathToFiles: String = ./dataset/green
weatherData: String = ./dataset/weather/weather_data_2017_2024.csv
weatherWmoLookup: String = ./dataset/weather/wmo_lookup_codes.csv
```

1.4 Columns for the analysis

[3]:
```
import org.apache.spark.sql.types._

val commonFields = List(
  StructField("VendorID", IntegerType),
  StructField("fare_amount", DoubleType),
  StructField("tip_amount", DoubleType),
  StructField("payment_type", LongType),
  StructField("trip_distance", DoubleType),
  StructField("total_amount", DoubleType)
)

val schemaYellow = StructType(
  StructField("tpep_pickup_datetime", TimestampType) ::
  StructField("tpep_dropoff_datetime", TimestampType) ::
  commonFields
)

val schemaGreen = StructType(
  StructField("lpep_pickup_datetime", TimestampType) ::
  StructField("lpep_dropoff_datetime", TimestampType) ::
  commonFields
)
```

[3]:
```
import org.apache.spark.sql.types._
commonFields: List[org.apache.spark.sql.types.StructField] =
List(StructField(VendorID,IntegerType,true),
StructField(fare_amount,DoubleType,true),
StructField(tip_amount,DoubleType,true),
StructField(payment_type,LongType,true),
StructField(trip_distance,DoubleType,true),
StructField(total_amount,DoubleType,true))
schemaYellow: org.apache.spark.sql.types.StructType = StructType(StructField(tpe
p_pickup_datetime,TimestampType,true),StructField(tpep_dropoff_datetime,Timestam
pType,true),StructField(VendorID,IntegerType,true),StructField(fare_amount,Doubl
eType,true),StructField(tip_amount,DoubleType,true),StructField(payment_type,Lon
gType,true),StructField(trip_distance,DoubleType,true),StructField(total_amount,
DoubleType,true))
schemaGreen: org.apache.sp…
```

2 Load Datasets

First we want to load the dataset relative to the taxi data.

[4]:
```
val projectDir: String = "/Users/giovanniantonioni/IdeaProjects/Drivers"
def getDatasetPath(localPath: String): String = {
  "file://" + projectDir + "/" + localPath
}
```

[4]:
```
projectDir: String = /Users/giovanniantonioni/IdeaProjects/Drivers
getDatasetPath: (localPath: String)String
```

[5]:
```
val (schema, pickupCol, dropoffCol) = datasetName match {
  case "yellow" => (schemaYellow, "tpep_pickup_datetime",␣
 ↪"tpep_dropoff_datetime")
  case _         => (schemaGreen, "lpep_pickup_datetime",␣
 ↪"lpep_dropoff_datetime")
}

val loadedDataset = spark.read
  .schema(schema)
  .option("recursiveFileLookup", "true")
  .parquet(getDatasetPath(pathToFiles))
  .select(
    $"VendorID",
    col(pickupCol).alias("pickup_datetime"),
    col(dropoffCol).alias("dropoff_datetime"),
    $"fare_amount",
    $"tip_amount",
    $"payment_type",
    $"trip_distance",
    $"total_amount"
  )
  .na.drop()
  .dropDuplicates()
  .rdd
```

[5]:
```
schema: org.apache.spark.sql.types.StructType = StructType(StructField(lpep_pick
up_datetime,TimestampType,true),StructField(lpep_dropoff_datetime,TimestampType,
true),StructField(VendorID,IntegerType,true),StructField(fare_amount,DoubleType,
true),StructField(tip_amount,DoubleType,true),StructField(payment_type,LongType,
true),StructField(trip_distance,DoubleType,true),StructField(total_amount,Double
Type,true))
pickupCol: String = lpep_pickup_datetime
dropoffCol: String = lpep_dropoff_datetime
loadedDataset: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[8] at rdd at <console>:50
```

3 Filtering

[6]:
```
import org.apache.spark.sql.Row
import org.apache.spark.rdd.RDD

def filterDataset(dataset: RDD[Row], name: String): RDD[Row] = {
  val allowedYellowVendorId = Set(1, 2, 6, 7)
  val allowedGreenVendorId = Set(1, 2, 6)

  dataset.filter { case row =>
        val allowedIds = if (name == "yellow") allowedYellowVendorId else␣
 ↪allowedGreenVendorId
        val vendorId = row.getInt(0)
        allowedIds.contains(vendorId)
      }
      .filter(row => row.getDouble(3) > 0)
      .filter(row => row.getDouble(4) >= 0)
      .filter(row => row.getDouble(4) <= row.getDouble(3) * 1.5)
      .filter(row => row.getDouble(6) > 0)
      .filter{ row =>
        val dropOffDateTime = row.getTimestamp(2)
        val pickupDateTime = row.getTimestamp(1)
        dropOffDateTime.after(pickupDateTime)
      }
}

val filtered = filterDataset(loadedDataset, datasetName)
```

[6]:
```
import org.apache.spark.sql.Row
import org.apache.spark.rdd.RDD
filterDataset: (dataset: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row],
name: String)org.apache.spark.rdd.RDD[org.apache.spark.sql.Row]
filtered: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[14] at filter at <console>:45
```

[7]:
```
val withTripDuration = filtered.map { row =>
  val durationMin = (row.getTimestamp(2).getTime - row.getTimestamp(1).getTime).
 ↪toDouble / (1000 * 60)
  (row, durationMin)
}
```

[7]:
```
withTripDuration: org.apache.spark.rdd.RDD[(org.apache.spark.sql.Row, Double)] =
MapPartitionsRDD[15] at map at <console>:30
```

[8]:
```
val tripDistances = withTripDuration.map { case (row, _) => row.getLong(5).
 ↪toInt }
val tripDurations = withTripDuration.map { case (_, duration) => duration }

val distanceDF = tripDistances.toDF("trip_distance")
val durationDF = tripDurations.toDF("trip_duration")

val tripDistanceOutlier = distanceDF.stat.approxQuantile("trip_distance",␣
 ↪Array(0.02, 0.98), 0.01)
val tripDurationOutlier = durationDF.stat.approxQuantile("trip_duration",␣
 ↪Array(0.02, 0.98), 0.01)
```

[8]:
```
tripDistances: org.apache.spark.rdd.RDD[Int] = MapPartitionsRDD[16] at map at
<console>:30
tripDurations: org.apache.spark.rdd.RDD[Double] = MapPartitionsRDD[17] at map at
<console>:31
distanceDF: org.apache.spark.sql.DataFrame = [trip_distance: int]
durationDF: org.apache.spark.sql.DataFrame = [trip_duration: double]
tripDistanceOutlier: Array[Double] = Array(1.0, 2.0)
tripDurationOutlier: Array[Double] = Array(2.25, 42.083333333333336)
```

[9]:
```
def filterOutOutlier(
  dataset: RDD[(Row, Double)],
  tripDistanceOutlier: Array[Double],
  tripDurationOutlier: Array[Double]
): RDD[(Row, Double)] = {

  val distanceLower = tripDistanceOutlier(0)
  val distanceUpper = tripDistanceOutlier(1)
  val durationLower = tripDurationOutlier(0)
  val durationUpper = tripDurationOutlier(1)

  dataset.filter { case (row, duration) =>
    val tripDistance = row.getLong(5).toInt
    tripDistance >= distanceLower && tripDistance <= distanceUpper &&
    duration >= durationLower && duration <= durationUpper
  }
}


val filteredOut = filterOutOutlier(withTripDuration, tripDistanceOutlier,␣
 ↪tripDurationOutlier)
val enriched = filteredOut.map { case (row, duration) =>
  val pickupCalendar = java.util.Calendar.getInstance()
  pickupCalendar.setTime(row.getTimestamp(1))

  val hourOfDay = pickupCalendar.get(java.util.Calendar.HOUR_OF_DAY)

  val tipAmount = row.getDouble(4)
  val totalAmount = row.getDouble(7)

  val tipPercentage = if (totalAmount != 0) (tipAmount / totalAmount) * 100␣
 ↪else 0.0

  val tripDistance = row.getDouble(6)
  val speedMph = if (duration > 0) tripDistance / (duration / 60.0) else 0.0

  (row, duration, hourOfDay, tipPercentage, speedMph)
}
```

[9]:
```
filterOutOutlier: (dataset: org.apache.spark.rdd.RDD[(org.apache.spark.sql.Row,
Double)], tripDistanceOutlier: Array[Double], tripDurationOutlier:
Array[Double])org.apache.spark.rdd.RDD[(org.apache.spark.sql.Row, Double)]
filteredOut: org.apache.spark.rdd.RDD[(org.apache.spark.sql.Row, Double)] =
MapPartitionsRDD[36] at filter at <console>:43
enriched: org.apache.spark.rdd.RDD[(org.apache.spark.sql.Row, Double, Int,
Double, Double)] = MapPartitionsRDD[37] at map at <console>:52
```

[ ]:

[11]:
```
val binned =  enriched.map { case (row, duration, hourOfDay, tipPercentage,␣
 ↪speedMph) =>
  val binConfigs = Map(
    "trip_distance" -> (Seq(1.0, 3.0, 6.0), Seq("0-1", "1-3", "3-6", "6+")),
    "trip_duration_min" -> (Seq(5.0, 15.0, 30.0), Seq("0-5", "5-15", "15-30",␣
 ↪"30+")),
    "fare_amount" -> (Seq(5.0, 10.0, 20.0, 40.0), Seq("0-5", "5-10", "10-20",␣
 ↪"20-40", "40+")),
    "tip_percentage" -> (Seq(5.0, 10.0, 20.0, 30.0), Seq("0-5%", "5-10%",␣
 ↪"10-20%", "20-30%", "30%+")),
    "speed_mph" -> (Seq(5.0, 15.0, 30.0), Seq("0-5mph", "5-15mph", "15-30mph",␣
 ↪"30mph+"))
```

```
)

def assignBin(value: Double, bins: Seq[Double], labels: Seq[String]): String␣
↪= {
  require(labels.length == bins.length + 1, "You need one more label than bin␣
↪thresholds.")

  if (value < bins.head) labels.head
  else {
    val idx = bins.indexWhere(b => value < b)
    if (idx == -1) labels.last
    else labels(idx)
  }
}

val tripDistance = row.getDouble(6)
val tripDistanceBin = assignBin(
  tripDistance,
  binConfigs("trip_distance")._1,
  binConfigs("trip_distance")._2
)

val tripDurationBin = assignBin(
  duration,
  binConfigs("trip_duration_min")._1,
  binConfigs("trip_duration_min")._2
)

val fareAmount = row.getDouble(3)
val fareAmountBin = assignBin(
  fareAmount,
  binConfigs("fare_amount")._1,
  binConfigs("fare_amount")._2
)

val speedBin = assignBin(
  speedMph,
  binConfigs("speed_mph")._1,
  binConfigs("speed_mph")._2
)

def tripHourBucket(hour: Int): String = hour match {
  case h if h >= 0 && h <= 5  => "late_night"
  case h if h >= 6 && h <= 9  => "morning"
  case h if h >= 10 && h <= 15 => "midday"
  case h if h >= 16 && h <= 19 => "evening"
  case _ => "night"
  }

  val hourBin = tripHourBucket(hourOfDay)
  (row, tripDistanceBin, tripDurationBin, fareAmountBin, speedBin, hourBin,␣
↪tipPercentage)
}
```

[11]:
```
binned: org.apache.spark.rdd.RDD[(org.apache.spark.sql.Row, String, String,
String, String, String, Double)] = MapPartitionsRDD[38] at map at <console>:30
```

[12]:
```
val weatherFileRDD = spark.read
  .format("CSV")
  .option("header", "true")
  .load(getDatasetPath(weatherData))
  .rdd
  .map { row =>
    val code = row.getString(1).trim.toInt
    val date = row.getString(0).trim
    (code, date)
  }

val wmoLookupFile = spark.read
  .format("CSV")
  .option("header", "true")
  .load(getDatasetPath(weatherWmoLookup))
  .rdd

val wmoLookupPairRDD = wmoLookupFile.map { row =>
  val data = row.getString(0).split(";")
  val code =data(0).trim.toInt
  val description = data(1).trim
  (code, description)
}

import java.time.LocalDate
import java.sql.Timestamp

val transformedWeatherClassRDD = weatherFileRDD
  .join(wmoLookupPairRDD)
  .map(row => {
    val (id, (date, description)) = row
    val formattedDate  = LocalDate.parse(date)
    val timestamp = Timestamp.valueOf(formattedDate.atStartOfDay())
    (id, timestamp, description)
})
```

[12]:
```
weatherFileRDD: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[54]
at map at <console>:38
wmoLookupFile: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[69] at rdd at <console>:47
wmoLookupPairRDD: org.apache.spark.rdd.RDD[(Int, String)] = MapPartitionsRDD[70]
at map at <console>:50
import java.time.LocalDate
import java.sql.Timestamp
transformedWeatherClassRDD: org.apache.spark.rdd.RDD[(Int, java.sql.Timestamp,
String)] = MapPartitionsRDD[74] at map at <console>:62
```

[13]:
```
val weatherByDate = transformedWeatherClassRDD.map { row =>
  (row._2.toLocalDateTime.toLocalDate, row)
}

val rideByDate = binned.map {data  =>
  val pickupDateTime = data._1.getTimestamp(1).toLocalDateTime.toLocalDate
  (pickupDateTime, data)
}
```

[13]:
```
weatherByDate: org.apache.spark.rdd.RDD[(java.time.LocalDate, (Int,
java.sql.Timestamp, String))] = MapPartitionsRDD[75] at map at <console>:33
rideByDate: org.apache.spark.rdd.RDD[(java.time.LocalDate,
(org.apache.spark.sql.Row, String, String, String, String, String, Double))] =
MapPartitionsRDD[76] at map at <console>:37
```

4 Join weather and Ride data

[14]:
```
val joinedWeather = rideByDate.join(weatherByDate).map {
  case (_, (ride, weather)) => (ride, weather)
}

val finalRDD = joinedWeather.map { case (ride, weather) =>
  def generalWeatherLabel(wmoCode: Int): String = wmoCode match {
    case c if Seq(0, 1).contains(c)              => "clear"
    case c if Seq(2, 3, 4).contains(c)           => "cloudy"
    case c if Seq(45, 48).contains(c)            => "foggy"
    case c if (50 to 67).contains(c)       => "rainy"
    case c if (70 to 77).contains(c)       => "snowy"
    case c if (80 to 99).contains(c)       => "stormy"
    case _                                       => "unknown"
  }

  val generalWeather = generalWeatherLabel(weather._1)
  (ride, weather._1, generalWeather)
}
```

[14]:
```
joinedWeather: org.apache.spark.rdd.RDD[((org.apache.spark.sql.Row, String,
String, String, String, String, Double), (Int, java.sql.Timestamp, String))] =
MapPartitionsRDD[80] at map at <console>:33
finalRDD: org.apache.spark.rdd.RDD[((org.apache.spark.sql.Row, String, String,
String, String, String, Double), Int, String)] = MapPartitionsRDD[81] at map at
<console>:37
```

5 Export the results

[15]:
```
val binFields = Seq(
  "tripDistanceBin",
  "tripDurationBin",
  "fareAmountBin",
  "speedBin"
)

val keyedRDD = finalRDD.flatMap{ case (ride, code, generalWeather) =>
  binFields.map { field =>
    val bin = field match {
      case "fareAmountBin" => ride._4
      case "tripDistanceBin" => ride._2
      case "tripDurationBin" => ride._3
      case "speedBin" => ride._5
    }
    val key = s"${field}_$bin"
    (key, ride)
  }
}
```

[15]:
```
binFields: Seq[String] = List(tripDistanceBin, tripDurationBin, fareAmountBin,
speedBin)
keyedRDD: org.apache.spark.rdd.RDD[(String, (org.apache.spark.sql.Row, String,
String, String, String, String, Double))] = MapPartitionsRDD[82] at flatMap at
<console>:39
```

[16]:
```
val distributedKeyedRDD = keyedRDD.mapValues{ case ride =>
  val tipPercentage = ride._7
  (tipPercentage, 1L)
}

val avgTipRDD = distributedKeyedRDD
  .reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2))
  .map {
    case (id, (sum, count)) => Row(id, sum / count)
  }


val allTipByBinSchema = StructType(Seq(
  StructField("feature", StringType),
  StructField("avg_tip_pct", DoubleType)
))

spark.createDataFrame(avgTipRDD, allTipByBinSchema)
.write
.mode("overwrite")
.parquet(getDatasetPath(s"$outputDir/tip_avg_per_bin/all_features"))
```

[16]:
```
distributedKeyedRDD: org.apache.spark.rdd.RDD[(String, (Double, Long))] =
MapPartitionsRDD[83] at mapValues at <console>:35
avgTipRDD: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[85] at map at <console>:42
allTipByBinSchema: org.apache.spark.sql.types.StructType = StructType(StructFiel
d(feature,StringType,true),StructField(avg_tip_pct,DoubleType,true))
```

[17]:
```
val avgTipByWeather = finalRDD
    .map(r => (r._3, (r._1._7, 1L)))
    .reduceByKey((a, b) => (a._1 + b._1, a._2 + b._2))
    .map { case (weather, (sumTip, count)) => Row(weather, sumTip / count) }

val weatherSchema = StructType(Seq(
  StructField("weather", StringType),
  StructField("avg_tip_pct", DoubleType)
))

spark.createDataFrame(avgTipByWeather, weatherSchema)
  .write
  .mode("overwrite")
  .parquet(getDatasetPath(s"$outputDir/avg_tip_by_weather"))
```

[17]:
```
avgTipByWeather: org.apache.spark.rdd.RDD[org.apache.spark.sql.Row] =
MapPartitionsRDD[91] at map at <console>:38
weatherSchema: org.apache.spark.sql.types.StructType = StructType(StructField(we
ather,StringType,true),StructField(avg_tip_pct,DoubleType,true))
```