

Darshil Ashishbhai Vora

## Project 2- Individual Result-Hot Spot Analysis

### Reflection:-

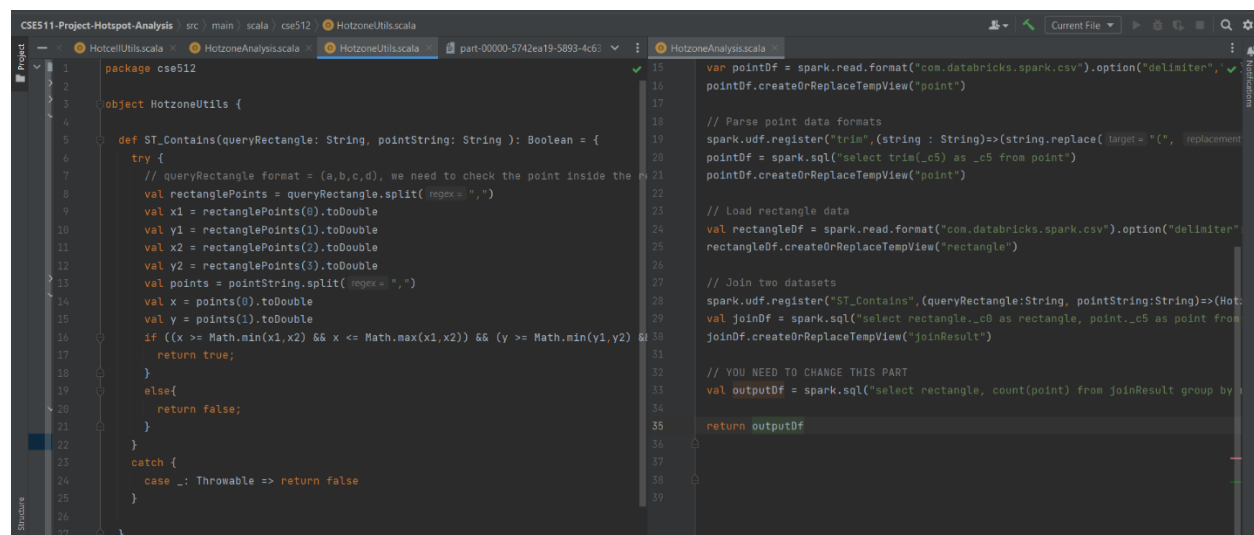
### Environment Setup :-

I have set up the project on a Windows 10 Machine with 16GB RAM. I have added Scala SDK 2.12.11, Apache Spark 3.2.1 and used Oracle OpenJDK version 1.8.0\_341 which was already installed as per the instructions provided. I have tested The Hadoop and Spark setup as suggested in the assignment by running sample Spark programs.

The coding was done in Scala using IntelliJ IDEA with the Scala plugin. I have set up the workspace by importing the provided template code files into the IDE. Necessary SBT and Assembly plugins were configured to compile and package the code.

### Hot Zone Analysis

The hot zone analysis task involved performing a range join on rectangle and point datasets to obtain the count of points within each rectangle. I have loaded the rectangle dataset containing coordinates of 30 zones from the provided file. And as provided in the assignment, the point dataset corresponded to January 2009 NYC yellow taxi trip data containing pickup location coordinates. I have added my code as indicated in a spatial range join using the ST\_Contains and ST\_Within functions from the GeoSpark library to check if each point lies within a rectangle. This returned a DataFrame with the rectangle ID and enclosed point count for each rectangle. Then, I sorted the output DataFrame by the 'rectangle' column and written to file. I have also added the coalesce(1) after my query to generate 1 csv file because otherwise it was generating more than 1 csv file each containing only 1 value. This completed the task of calculating hotness values for different zones based on enclosed point counts.



```
1 package cse512
2
3 object HotzoneUtils {
4
5   def ST_Contains(queryRectangle: String, pointString: String): Boolean = {
6     try {
7       // queryRectangle format = (a,b,c,d), we need to check the point inside the rectangle
8       val rectanglePoints = queryRectangle.split(",")
9       val x1 = rectanglePoints(0).toDouble
10      val y1 = rectanglePoints(1).toDouble
11      val x2 = rectanglePoints(2).toDouble
12      val y2 = rectanglePoints(3).toDouble
13      val points = pointString.split(",")
14      val x = points(0).toDouble
15      val y = points(1).toDouble
16      if ((x >= Math.min(x1,x2) && x <= Math.max(x1,x2)) && (y >= Math.min(y1,y2) && y <= Math.max(y1,y2))) {
17        return true;
18      }
19      else{
20        return false;
21      }
22    }
23    catch {
24      case _: Throwable => return false
25    }
26  }
27 }
28
29 // HotzoneAnalysis.scala
30
31 var pointDf = spark.read.format("com.databricks.spark.csv").option("delimiter", ",")
32   .load("point.csv")
33   .createOrReplaceTempView("point")
34
35 // Parse point data formats
36 spark.udf.register("trim", (string: String) => string.replace(" ", ""))
37 pointDf = spark.sql("select trim(_c5) as _c5 from point")
38 pointDf.createOrReplaceTempView("point")
39
40 // Load rectangle data
41 val rectangleDf = spark.read.format("com.databricks.spark.csv").option("delimiter", ",")
42   .load("rectangle.csv")
43   .createOrReplaceTempView("rectangle")
44
45 // Join two datasets
46 spark.udf.register("ST_Contains", (queryRectangle: String, pointString: String) => HotzoneUtils.ST_Contains(queryRectangle, pointString))
47 val joinDf = spark.sql("select rectangle._c0 as rectangle, point._c5 as point from rectangle join point on ST_Contains(rectangle._c0, point._c5)")
48 joinDf.createOrReplaceTempView("joinResult")
49
50 // YOU NEED TO CHANGE THIS PART
51 val outputDf = spark.sql("select rectangle, count(point) from joinResult group by rectangle")
52 outputDf.coalesce(1).write.csv("output.csv", true)
```

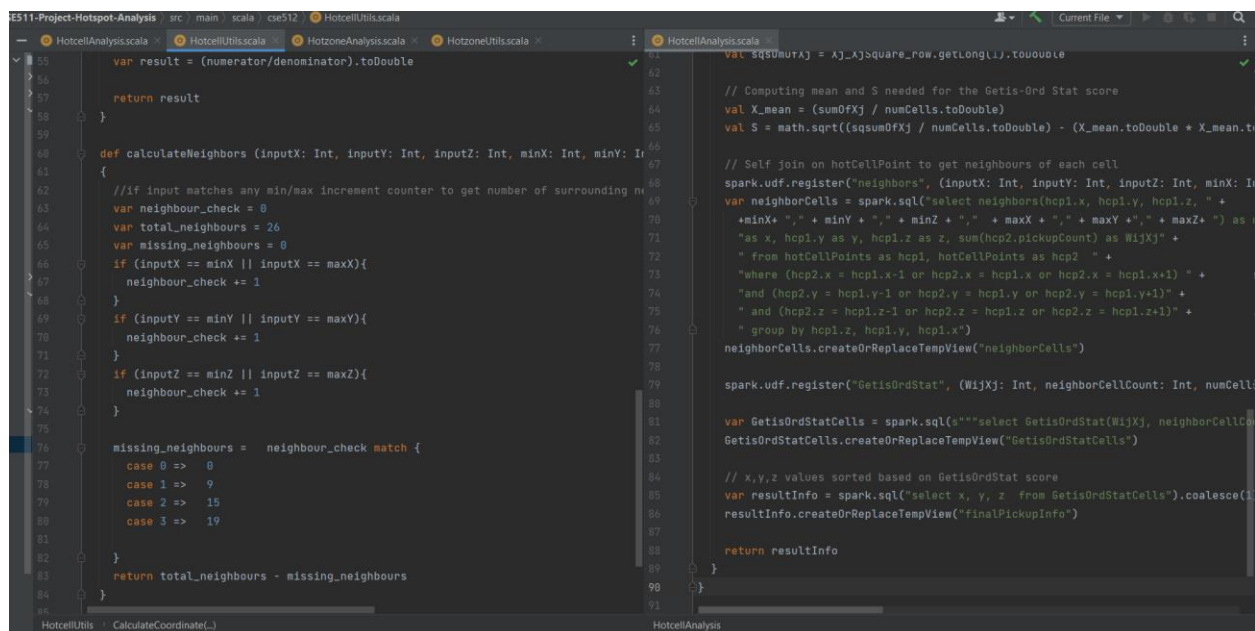
## Hot Cell Analysis

For the hot cell analysis task, January 2009 yellow taxi trip data was used as input. The DataFrame is partitioned into cells of size 0.01 latitude x 0.01 longitude degrees.

The Getis-Ord statistic was implemented to calculate z-scores and p-values for each cell. They indicate the significance of spatial clustering of high or low values around that cell. For this part, I have made other calculations to find the sum and sum of square values. So, this also helped with counting average and standard deviation. Then I have also implemented the functions like calculateNumberOfNeighbors() and calculateGScore() to find out the number of neighbour cells Getis-Ord statistics for that cell.

I have extracted only the coordinates of top 50 hottest cells from the output DataFrame after sorting in descending order of z-scores. These coordinates are written to output file without the actual z-score values. As in the previous analysis, I had to add ,coalesce(1) to generate 1 csv file.

This concludes the hot spot analysis tasks on the given datasets using Apache Spark. The coded solutions were packaged into a JAR file and executed on a Spark cluster to perform distributed computation and obtain results.



```
var result = (numerator/denominator).toDouble
return result

def calculateNeighbors (inputX: Int, inputY: Int, inputZ: Int, minX: Int, minY: Int, maxZ: Int) {
  //if input matches any min/max increment counter to get number of surrounding neighbors
  var neighbour_check = 0
  var total_neighbours = 26
  var missing_neighbours = 0
  if (inputX == minX || inputX == maxZ){
    neighbour_check += 1
  }
  if (inputY == minY || inputY == maxZ){
    neighbour_check += 1
  }
  if (inputZ == minZ || inputZ == maxZ){
    neighbour_check += 1
  }
  missing_neighbours = neighbour_check match {
    case 0 => 0
    case 1 => 9
    case 2 => 15
    case 3 => 19
  }
  return total_neighbours - missing_neighbours
}

val sqSumX = xj.xsquare_row.getLong(1).toDouble
// Computing mean and S needed for the Getis-Ord Stat score
val X_mean = (sumOfXj / numCells.toDouble)
val S = math.sqrt((sqsumOfXj / numCells.toDouble) - (X_mean.toDouble * X_mean.toDouble))

// Self join on hotCellPoint to get neighbours of each cell
spark.udf.register("neighbors", (inputX: Int, inputY: Int, inputZ: Int, minX: Int, minY: Int, maxZ: Int) => {
  var neighborCells = spark.sql("select neighbors(hcp1.x, hcp1.y, hcp1.z, " +
    +minX+ ", " + minY+ ", " + minZ+ ", " + maxX+ ", " + maxY+ ", " + maxZ+ ") as " +
    "as x, hcp1.y as y, hcp1.z as z, sum(hcp2.pickupCount) as WjXj" +
    " from hotCellPoints as hcp1, hotCellPoints as hcp2 " +
    "where (hcp2.x = hcp1.x-1 or hcp2.x = hcp1.x or hcp2.x = hcp1.x+1) " +
    "and (hcp2.y = hcp1.y-1 or hcp2.y = hcp1.y or hcp2.y = hcp1.y+1) " +
    "and (hcp2.z = hcp1.z-1 or hcp2.z = hcp1.z or hcp2.z = hcp1.z+1)" +
    " group by hcp1.z, hcp1.y, hcp1.x")
  neighborCells.createOrReplaceTempView("neighborCells")

  spark.udf.register("GetisOrdStat", (WjXj: Int, neighborCellCount: Int, numCells: Int) => {
    var GetisOrdStatCells = spark.sql(s"select GetisOrdStat($WjXj, $neighborCellCount, $numCells) as GetisOrdStatCells")
    GetisOrdStatCells.createOrReplaceTempView("GetisOrdStatCells")

    // x,y,z values sorted based on GetisOrdStat score
    var resultInfo = spark.sql("select x, y, z from GetisOrdStatCells").coalesce(1)
    resultInfo.createOrReplaceTempView("finalPickupInfo")

    return resultInfo
  })
}
```

## Lessons Learned :-

This project provided valuable hands-on learning experience in applying spatial analysis techniques on large datasets using Apache Spark. I have learned spatial relationships and operations which is critical for problems involving location data like range joins. The GeoSpark library simplifies these types of geospatial tasks.

I have also learned about Spark SQL which makes code more concise for complex multi-step data transformations compared to procedural approaches. I have also learned about the spark's in-memory

processing which makes it very efficient to work with big data and especially spatial data. Performance tuning becomes important at scale. So, optimization techniques can greatly boost job performance. I have also learned about integrating different technologies such as Spark, Scala, Hadoop and how to work with different systems like windows. Handling diverse data formats and merging these technologies really helped me to execute this project.

This assignment used real world data and prepared us for the real-world questions, so I also learned about applying these techniques for real-world applications. So, Overall, this project provided a wealth of experience in designing, implementing, and optimizing distributed parallel programs on Spark for geospatial analytics.

## Output and Results:-

```
*.csv files are supported in other JetBrains IDEs
1 -7399, 4075, 15
2 -7399, 4075, 29
3 -7399, 4075, 22
4 -7399, 4075, 28
5 -7399, 4075, 14
6 -7399, 4075, 30
7 -7398, 4075, 15
8 -7399, 4075, 23
9 -7399, 4075, 16
10 -7398, 4075, 29
11 -7399, 4075, 21
12 -7398, 4075, 28
13 -7399, 4075, 27
14 -7398, 4075, 22
15 -7399, 4074, 30
16 -7399, 4074, 15
17 -7398, 4075, 14
18 -7399, 4074, 23
19 -7399, 4074, 29
20 -7399, 4075, 13
21 -7399, 4074, 22
22 -7399, 4074, 16
23 -7398, 4075, 30
24 -7398, 4075, 23
25 -7398, 4076, 15
26 -7399, 4075, 9
27 -7398, 4075, 16
28 -7398, 4075, 21
29 -7398, 4075, 27
30 -7399, 4074, 28

*csv files are supported in other JetBrains IDEs
1 "-73.789411, 40.666459, -73.756364, 40.680494", 1
2 "-73.793638, 40.710719, -73.752336, 40.730202", 1
3 "-73.795658, 40.743334, -73.753772, 40.779114", 1
4 "-73.796512, 40.722355, -73.756699, 40.745784", 1
5 "-73.797297, 40.738291, -73.775740, 40.770411", 1
6 "-73.802033, 40.652546, -73.738566, 40.668036", 8
7 "-73.805770, 40.666526, -73.772204, 40.690003", 3
8 "-73.815233, 40.715862, -73.790295, 40.738951", 2
9 "-73.816380, 40.690802, -73.768447, 40.715693", 1
10 "-73.819131, 40.582543, -73.761289, 40.609861", 1
11 "-73.825921, 40.702281, -73.790734, 40.719217", 2
12 "-73.826577, 40.757744, -73.790317, 40.779587", 1
13 "-73.832707, 40.620010, -73.746541, 40.665414", 200
14 "-73.839460, 40.746988, -73.815375, 40.781725", 3
15 "-73.840130, 40.662481, -73.801134, 40.691956", 4
16 "-73.840817, 40.775618, -73.790584, 40.801020", 1
17 "-73.842332, 40.804005, -73.790864, 40.845347", 2
18 "-73.843148, 40.701398, -73.816380, 40.715998", 2
19 "-73.849479, 40.681155, -73.810634, 40.704221", 2
20 "-73.861099, 40.714345, -73.817260, 40.764083", 21
21 "-73.862040, 40.706406, -73.829798, 40.739016", 16
22 "-73.864482, 40.833000, -73.851686, 40.842478", 1
23 "-73.867911, 40.734291, -73.847510, 40.748860", 1
24 "-73.869236, 40.765468, -73.825139, 40.798517", 136
25 "-73.873659, 40.744942, -73.853128, 40.758349", 1
26 "-73.875093, 40.711476, -73.856258, 40.735947", 3
27 "-73.876389, 40.756642, -73.854497, 40.771776", 67
28 "-73.878249, 40.581702, -73.766710, 40.639962", 10
29 "-73.884652, 40.822974, -73.856153, 40.833622", 1
30 "-73.891752, 40.727801, -73.864472, 40.749165", 8
```