Name: Sidharth Dinesh
ASU ID: 1225238352
Course: CSE 511 Data Processing at Scale

Individual Report for Project - 2

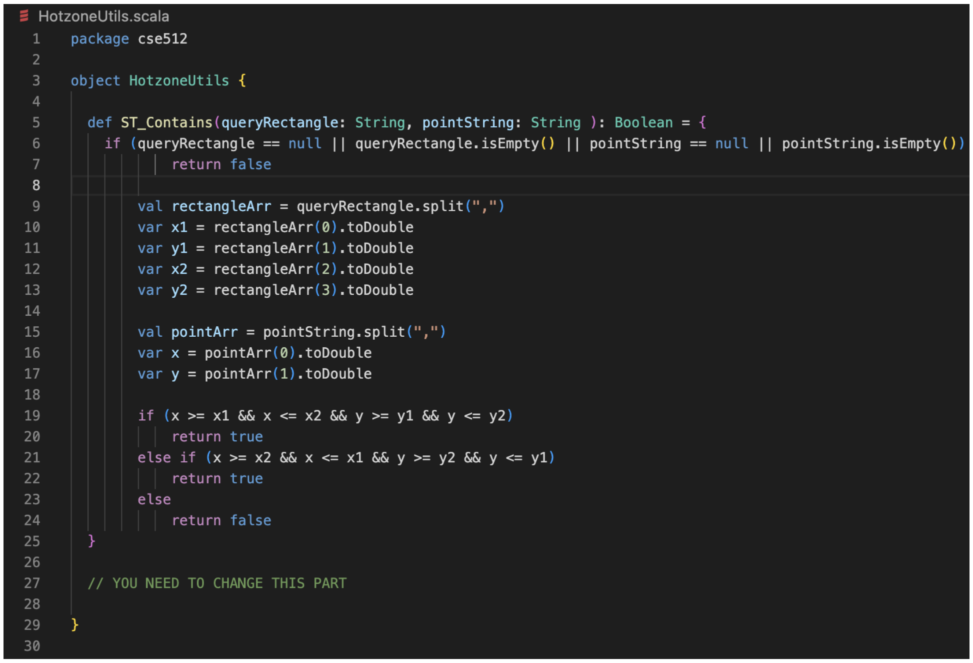
Purpose of Project:

Purpose of Project-2 is to understand and perform Spatial Data Analysis (Hot Spot). We accomplish this by completing two different hot spot analysis tasks:

1. Hot Zone Analysis: Using Scala and Spark, this analysis will count the points falling within the area of rectangular datasets. Each rectangle will have a corresponding count associated, and the "hotter" it is, the more dense it is, i.e. more points. The objective of the task is to find how "hot" each rectangle is.
2. Hot Cell Analysis: This analysis will use certain spatial statistical parameters on spatio-temporal big data using Spark and Scala to locate statistically significant geographical hot spots.

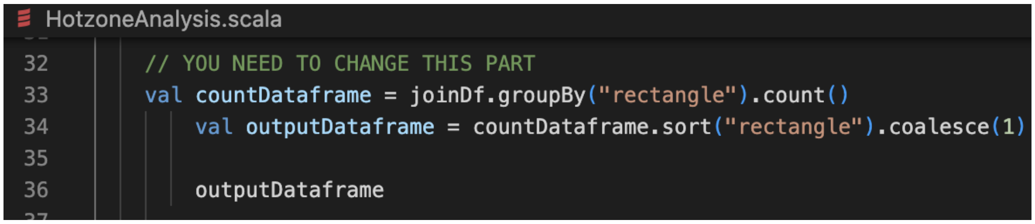
Hot Zone Analysis:

We modify the "ST_Contains" function, which returns "true" value in case the points are in the areal bounds of the rectangles, otherwise returns "false". By performing a JOIN operation on the rectangles dataset and points dataset and using WHERE condition giving us boolean values, we filter the results. GROUP BY and ORDER BY helps us to count and sort the Scala Dataframe.



HotzoneUtils.scala

```scala
package cse512

object HotzoneUtils {

  def ST_Contains(queryRectangle: String, pointString: String ): Boolean = {
    if (queryRectangle == null || queryRectangle.isEmpty() || pointString == null || pointString.isEmpty())
        return false

      val rectangleArr = queryRectangle.split(",")
      var x1 = rectangleArr(0).toDouble
      var y1 = rectangleArr(1).toDouble
      var x2 = rectangleArr(2).toDouble
      var y2 = rectangleArr(3).toDouble

      val pointArr = pointString.split(",")
      var x = pointArr(0).toDouble
      var y = pointArr(1).toDouble

      if (x >= x1 && x <= x2 && y >= y1 && y <= y2)
        return true
      else if (x >= x2 && x <= x1 && y >= y2 && y <= y1)
        return true
      else
        return false
  }

  // YOU NEED TO CHANGE THIS PART

}

```



HotzoneAnalysis.scala

```scala
// YOU NEED TO CHANGE THIS PART
val countDataframe = joinDf.groupBy("rectangle").count()
    val outputDataframe = countDataframe.sort("rectangle").coalesce(1)

    outputDataframe
```

Hot Cell Analysis:

We need to calculate the z-score as part of Hot Cell Analysis. For each cell in a space-time cube, it has 26 neighbors, i.e. total weights in a cell are usually 27. To perform transformation of datasets, we use a highly popular column-based function of Spark SQL, which is: “User-Defined Function” (UDF). To calculate weights, we have it set according to the cell boundaries in a way that weights are equal to:

a. 18 if the point lies on the x OR y OR z boundary.
b. 12 if the point lies on the (x AND y) OR (y AND z) OR (x AND z) boundary.
c. 8 if the point lies on the (x AND y AND z) boundary.

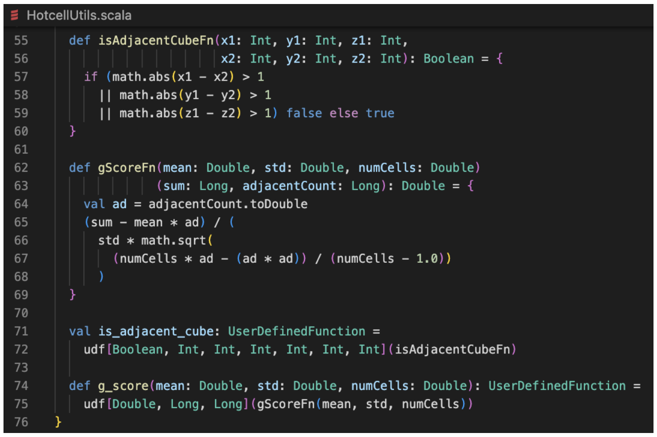
We calculate the Mean and Standard deviation using the below formulas:

$$\bar{X} = \frac{\sum_{j=1}^{n} x_j}{n} \qquad S = \sqrt{\frac{\sum_{j=1}^{n} x_j^2}{n} - (\bar{X})^2}$$

where x_j is the attribute value for cell j, $w_{i,j}$ is the spatial weight between cell i and j, n is equal to the total number of cells,

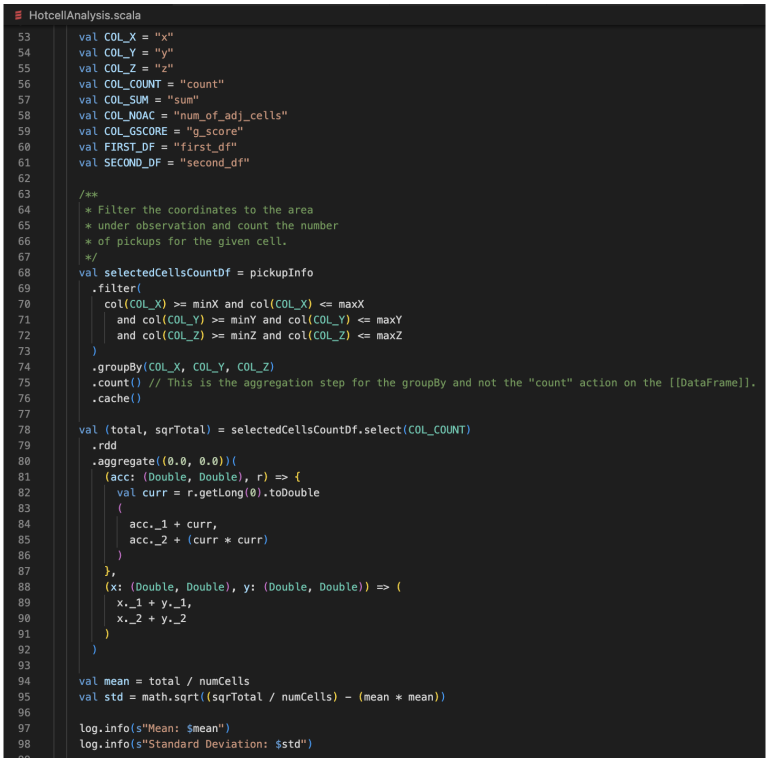
Finally using withColumn method, we calculate z-score using:

$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j} x_j - \bar{X} \sum_{j=1}^{n} w_{i,j}}{S \sqrt{\frac{\left[n \sum_{j=1}^{n} w_{i,j}^2 - \left(\sum_{j=1}^{n} w_{i,j}\right)^2\right]}{n-1}}}$$



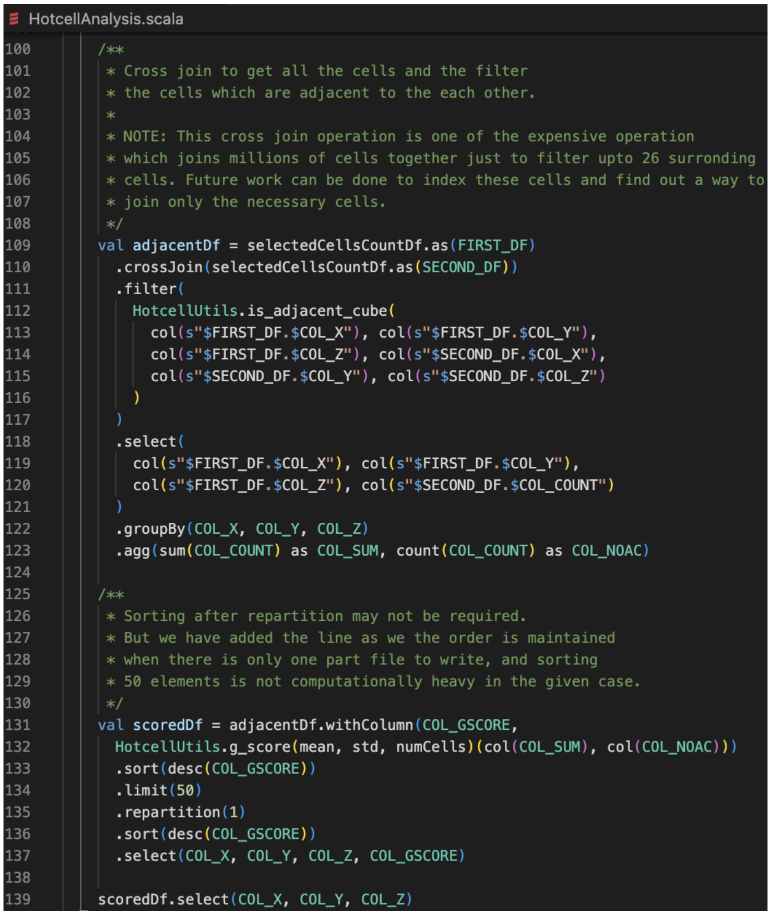
HotcellUtils.scala

```scala
  def isAdjacentCubeFn(x1: Int, y1: Int, z1: Int,
                       x2: Int, y2: Int, z2: Int): Boolean = {
    if (math.abs(x1 - x2) > 1
      || math.abs(y1 - y2) > 1
      || math.abs(z1 - z2) > 1) false else true
  }

  def gScoreFn(mean: Double, std: Double, numCells: Double)
              (sum: Long, adjacentCount: Long): Double = {
    val ad = adjacentCount.toDouble
    (sum - mean * ad) / (
      std * math.sqrt(
        (numCells * ad - (ad * ad)) / (numCells - 1.0))
      )
  }

  val is_adjacent_cube: UserDefinedFunction =
    udf[Boolean, Int, Int, Int, Int, Int, Int](isAdjacentCubeFn)

  def g_score(mean: Double, std: Double, numCells: Double): UserDefinedFunction =
    udf[Double, Long, Long](gScoreFn(mean, std, numCells))
}
```



HotcellAnalysis.scala

```scala
val COL_X = "x"
val COL_Y = "y"
val COL_Z = "z"
val COL_COUNT = "count"
val COL_SUM = "sum"
val COL_NOAC = "num_of_adj_cells"
val COL_GSCORE = "g_score"
val FIRST_DF = "first_df"
val SECOND_DF = "second_df"

/**
 * Filter the coordinates to the area
 * under observation and count the number
 * of pickups for the given cell.
 */
val selectedCellsCountDf = pickupInfo
  .filter(
    col(COL_X) >= minX and col(COL_X) <= maxX
      and col(COL_Y) >= minY and col(COL_Y) <= maxY
      and col(COL_Z) >= minZ and col(COL_Z) <= maxZ
  )
  .groupBy(COL_X, COL_Y, COL_Z)
  .count() // This is the aggregation step for the groupBy and not the "count" action on the [[DataFrame]].
  .cache()

val (total, sqrTotal) = selectedCellsCountDf.select(COL_COUNT)
  .rdd
  .aggregate((0.0, 0.0))(
    (acc: (Double, Double), r) => {
      val curr = r.getLong(0).toDouble
      (
        acc._1 + curr,
        acc._2 + (curr * curr)
      )
    },
    (x: (Double, Double), y: (Double, Double)) => (
      x._1 + y._1,
      x._2 + y._2
    )
  )

val mean = total / numCells
val std = math.sqrt((sqrTotal / numCells) - (mean * mean))

log.info(s"Mean: $mean")
log.info(s"Standard Deviation: $std")
```



HotcellAnalysis.scala

```scala
/**
 * Cross join to get all the cells and the filter
 * the cells which are adjacent to the each other.
 *
 * NOTE: This cross join operation is one of the expensive operation
 * which joins millions of cells together just to filter upto 26 surrounding
 * cells. Future work can be done to index these cells and find out a way to
 * join only the necessary cells.
 */
val adjacentDf = selectedCellsCountDf.as(FIRST_DF)
  .crossJoin(selectedCellsCountDf.as(SECOND_DF))
  .filter(
    HotcellUtils.is_adjacent_cube(
      col(s"$FIRST_DF.$COL_X"), col(s"$FIRST_DF.$COL_Y"),
      col(s"$FIRST_DF.$COL_Z"), col(s"$SECOND_DF.$COL_X"),
      col(s"$SECOND_DF.$COL_Y"), col(s"$SECOND_DF.$COL_Z")
    )
  )
  .select(
    col(s"$FIRST_DF.$COL_X"), col(s"$FIRST_DF.$COL_Y"),
    col(s"$FIRST_DF.$COL_Z"), col(s"$SECOND_DF.$COL_COUNT")
  )
  .groupBy(COL_X, COL_Y, COL_Z)
  .agg(sum(COL_COUNT) as COL_SUM, count(COL_COUNT) as COL_NOAC)

/**
 * Sorting after repartition may not be required.
 * But we have added the line as we the order is maintained
 * when there is only one part file to write, and sorting
 * 50 elements is not computationally heavy in the given case.
 */
val scoredDf = adjacentDf.withColumn(COL_GSCORE,
  HotcellUtils.g_score(mean, std, numCells)(col(COL_SUM), col(COL_NOAC)))
  .sort(desc(COL_GSCORE))
  .limit(50)
  .repartition(1)
  .sort(desc(COL_GSCORE))
  .select(COL_X, COL_Y, COL_Z, COL_GSCORE)

scoredDf.select(COL_X, COL_Y, COL_Z)
```