Project 2-Individual Report

BY: Shreyas Tirukoilur Parvatharajan
ASU ID:1231832006

REFLECTION

CODE:

1)HotCellAnalysis

```
// Calculate standard deviation for the points in the given area
val SD = math.sqrt((squaredsumcells / numCells) - (Xbar * Xbar))

// Registering the GetNeighbourCount function which calculates the number of
neighbors for a cell
spark.udf.register("GetNeighbourCount", (minX: Int, minY: Int, minZ: Int,
maxX: Int, maxY: Int, maxZ: Int, Xin: Int, Yin: Int, Zin: Int)
  => ((HotcellUtils.GetNeighbourCount(minX, minY, minZ, maxX, maxY, maxZ, Xin,
Yin, Zin))))

// Executing a query to select neighbors of each cell, calculate the count of
valid points and sum of points for each cell's neighbors
val Neighbours = spark.sql("select " +
  "view1.x as x, " +
  "view1.y as y, " +
  "view1.z as z, " +
  "GetNeighbourCount("+minX + "," + minY + "," + minZ + "," + maxX + "," +
maxY + "," + maxZ + "," + "view1.x,view1.y,view1.z) as totalNeighbours, " +
  "count(*) as neighboursWithValidPoints, " +
  "sum(view2.numPoints) as sumAllNeighboursPoints " +
  "from filteredPointCountDfView as view1, filteredPointCountDfView as view2 "
+
  "where (view2.x = view1.x+1 or view2.x = view1.x or view2.x = view1.x-1) and
(view2.y = view1.y+1 or view2.y = view1.y or view2.y = view1.y-1) and (view2.z
= view1.z+1 or view2.z = view1.z or view2.z = view1.z-1) " +
  "group by view1.z, view1.y, view1.x order by view1.z, view1.y,
view1.x").persist()

// Registering the GetGScore UDF which computes the Getis-Ord statistic for a
cell
spark.udf.register("GetGScore", (x: Int, y: Int, z: Int, numcells: Int,
mean:Double, sd: Double, totalNeighbours: Int, sumAllNeighboursPoints: Int) =>
((
  HotcellUtils.GetGScore(x, y, z, numcells, mean, sd, totalNeighbours,
sumAllNeighboursPoints))))
```

```
// Executing a query to select cells and their G-scores, ordering the results
by the G-score in descending order
val NeighboursDesc = spark.sql("select x, y, z, " +
  "GetGScore(x, y, z," +numCells+ ", " + Xbar + ", " + SD + ",
totalNeighbours, sumAllNeighboursPoints) as gi_statistic " +
  "from NeighboursView " +
  "order by gi_statistic desc")

// Showcasing the top 50 cells ordered by their G-score
NeighboursDesc.createOrReplaceTempView("NeighboursDescView")
NeighboursDesc.show()

// Preparing the final DataFrame of the top 50 hottest cells without the G-
scores
val hotcells_dec_order = spark.sql("select x,y,z from NeighboursDescView")
hotcells_dec_order
```

2)HotCellUtils

```
// Checks if a given point is within the boundaries defined by minimum and
maximum values.
def IsCellInBounds(x:Double, y:Double, z:Int, minX:Double, maxX:Double,
minY:Double, maxY:Double, minZ:Int, maxZ:Int): Boolean = {
  (x >= minX) && (x <= maxX) && (y >= minY) && (y <= maxY) && (z >= minZ) &&
(z <= maxZ)
}

// Determines if a point is at the boundary of the given range by comparing it
with the minimum and maximum values.
def CheckBoundary(point: Int, minVal: Int, maxVal: Int) : Int = {
  if (point == minVal || point == maxVal){
    1 // Returns 1 if the point is at the boundary
  } else {
    0 // Returns 0 otherwise
  }
}

// Counts the number of neighbors for a cell in a 3D space given the minimum
and maximum ranges.
def GetNeighbourCount(minX:Int, minY:Int, minZ:Int, maxX:Int, maxY:Int,
maxZ:Int, Xin:Int, Yin:Int, Zin:Int): Int ={
  // Mapping the location types to the number of neighbors
```

```
  val pointLocationInCube: Map[Int, String] = Map(0->"inside", 1 -> "face", 2-
> "edge", 3-> "corner")
  val mapping: Map[String, Int] = Map("inside" -> 26, "face" -> 17, "edge" ->
11, "corner" -> 7)

  // Initial state of the cell location type
  var intialState = 0

  // Update the state based on the cell's position in relation to the boundary
  intialState += CheckBoundary(Xin, minX, maxX)
  intialState += CheckBoundary(Yin, minY, maxY)
  intialState += CheckBoundary(Zin, minZ, maxZ)

  // Get the cell's location type based on its state
  val location = pointLocationInCube(intialState)

  // Return the number of neighbors based on the cell's location type
  mapping(location)
}

// Calculates the G-Score statistic for hot spot analysis.
def GetGScore(x: Int, y: Int, z: Int, numcells: Int, mean:Double, sd: Double,
totalNeighbours: Int, sumAllNeighboursPoints: Int): Double ={
  // Calculate the numerator of the G-Score formula
  val numerator = (sumAllNeighboursPoints.toDouble -
(mean*totalNeighbours.toDouble))

  // Calculate the denominator of the G-Score formula
  val denominator = sd * math.sqrt((((numcells.toDouble *
totalNeighbours.toDouble) - (totalNeighbours.toDouble *
totalNeighbours.toDouble)) / (numcells.toDouble-
1.0).toDouble).toDouble).toDouble

  // Return the G-Score
  numerator/denominator
}
```

3)HotZoneAnalysis

```
package cse512


import org.apache.spark.sql.{DataFrame, SaveMode, SparkSession}

object HotzoneAnalysis {

  // Define the method to run hot zone analysis, which takes a Spark session
and paths to point and rectangle datasets
  def runHotZoneAnalysis(spark: SparkSession, pointPath: String,
rectanglePath: String): DataFrame = {
    // Read the point dataset using the CSV format, specifying the delimiter
and lack of header
    var pointDf =
spark.read.format("com.databricks.spark.csv").option("delimiter",
",").option("header", "false").load(pointPath);
    pointDf.createOrReplaceTempView("point") // Create a temporary view of the
points

    // Register a UDF to trim brackets from the string representations of
points
    spark.udf.register("trim", (string: String) => (string.replace("(",
"").replace(")", "")))

    // Apply the 'trim' UDF to all records in pointDf and create a temporary
view with the results
    pointDf = spark.sql("select trim(_c5) as _c5 from point")
    pointDf.createOrReplaceTempView("point")

    // Read the rectangle dataset using the CSV format, specifying the
delimiter as tab and lack of header
    val rectangleDf =
spark.read.format("com.databricks.spark.csv").option("delimiter",
"\t").option("header", "false").load(rectanglePath);
    rectangleDf.createOrReplaceTempView("rectangle") // Create a temporary
view of the rectangles

    // Register a UDF that checks if a point is within a rectangle using a
method from the HotzoneUtils object
    spark.udf.register("ST_Contains", (queryRectangle: String, pointString:
String) => (HotzoneUtils.ST_Contains(queryRectangle, pointString)))

    // Perform a join between the rectangles and points where the points are
contained within the rectangles
    val joinDf = spark.sql("select rectangle._c0 as rectangle, point._c5 as
point from rectangle,point where ST_Contains(rectangle._c0, point._c5)")
```

```
    joinDf.createOrReplaceTempView("joinResult") // Create a temporary view of
the join result

    // Group the joined data by rectangle and count the number of points in
each, then order by the rectangle
    val orderedJoinDf = spark.sql("select rectangle, count(point) from
joinResult group by rectangle order by rectangle").persist()
    orderedJoinDf.createOrReplaceTempView("orderedJoin") // Create a temporary
view of the ordered results

    // Coalesce the results into a single partition (useful for small datasets
or when writing out to a single file)
    orderedJoinDf.coalesce(1)
  }
}
```

4)HotZoneUtils

```
// Define a package named cse512
package cse512

// Declare the HotzoneUtils object. This object contains utility methods for
spatial analysis.
object HotzoneUtils {

  // Define a method called ST_Contains that checks if a point is within a
rectangle.
  // It takes two strings as inputs: one representing a rectangle and the
other a point.
  def ST_Contains(queryRectangle: String, pointString: String): Boolean = {

    // Split the string representing the rectangle into an array of strings,
    // then convert each string to a Double to get the rectangle's
coordinates.
    val Array(x1, y1, x2, y2) = queryRectangle.split(",").map(x => x.toDouble)

    // Calculate the minimum and maximum x (longitude) bounds of the
rectangle.
    val xmin = math.min(x1, x2)
    val xmax = math.max(x1, x2)
```

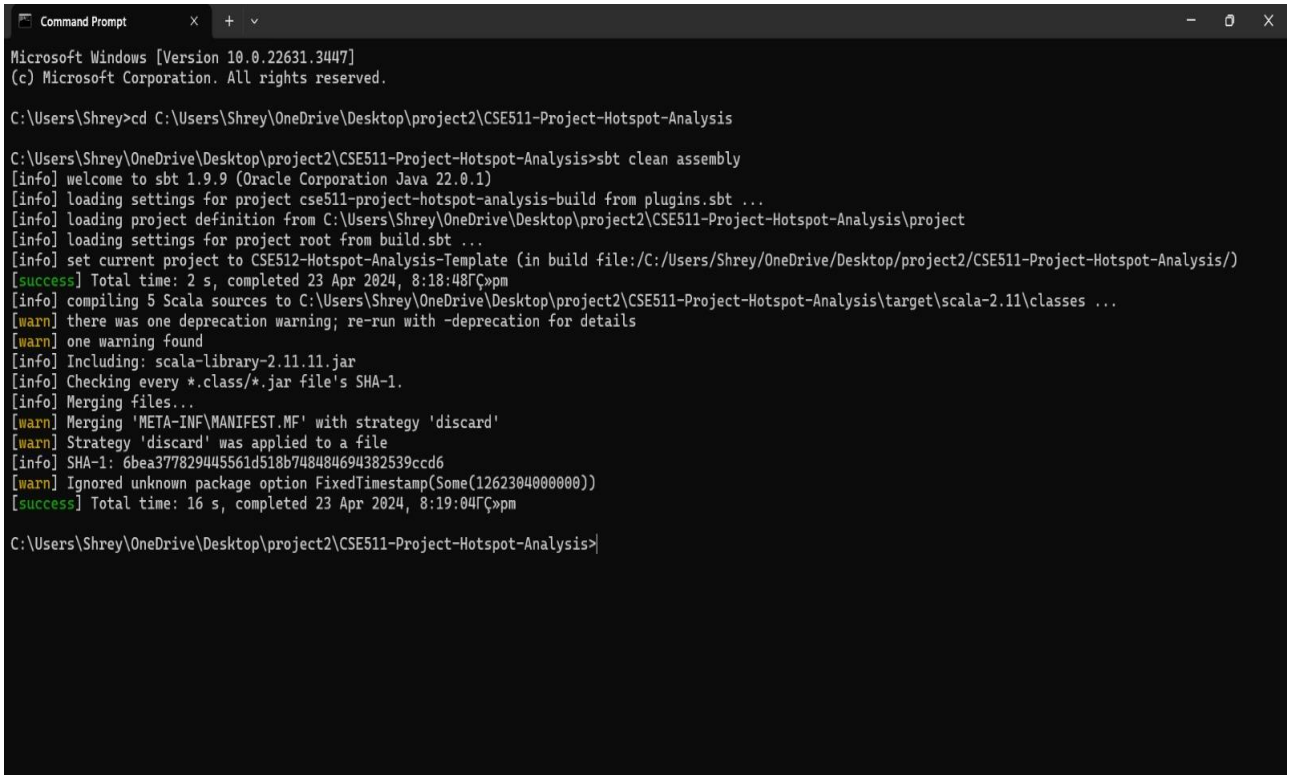
```
    // Calculate the minimum and maximum y (latitude) bounds of the rectangle.
    val ymin = math.min(y1, y2)
    val ymax = math.max(y1, y2)

    // Split the string representing the point into an array of strings,
    // then convert each string to a Double to get the point's coordinates.
    val Array(xp, yp) = pointString.split(',').map(x => x.toDouble)

    // Return true if the point's x coordinate is within the rectangle's x
bounds
    // and the point's y coordinate is within the rectangle's y bounds.
    // Otherwise, return false.
    (xp >= xmin) && (xp <= xmax) && (yp >= ymin) && (yp <= ymax)
  }
}
```

OUTPUT:

ASSEMBLY:



LEASSONS LEARNED

1)Apache Spark Mastery: Advanced understanding of Spark's distributed computing for large-scale spatial data processing.

2) Big Data Handling: Improved strategies for managing and efficiently processing extensive datasets

3) Hot Zone Analysis:a) Function Development: Creation of a range join operation to correlate rectangle datasets with point datasets and assess "hotness".
b) UDF Implementation: Utilization of a User Defined Function in Spark to validate point containment within rectangles.
c) Output Precision: Ensuring the orderly arrangement of the DataFrame in accordance with "rectangle" strings.

4)Hot Cell Analysis:a) Strategic Planning: Employment of spatial statistics to pinpoint spatial hot spots.
b) Data Loading and Coordination: Extraction and calculation of cell coordinates from NYC Taxi Trip datasets.
c) Statistical Analysis: Computation of the Getis-Ord G* statistic for cell evaluation and identification of the top 50 hot spots."