CSE 511: DATA PROCESSING AT SCALE

PROJECT 2: INDIVIDUAL REPORT

I. REFLECTION:

My project setup was done in Windows as well as Ubuntu 20.04 LTS. For testing I used Ubuntu due to some technical issues in my windows environment. I followed the windows and ubuntu guidelines given in canvas for this project. To run Scala and Apache Spark, Java Development Kit (JDK 8) was needed. JDK is suitable due to its seamless compatibility with Scala and Spark. I downloaded the JDK package and did the configurations such as setting up environment variables like JAVA_HOME.

Next step was downloading the Scala Simple Build Tool (SBT) for windows from the official Scala-sbt website. I downloaded and setup the sbt-1.5.5.msi. Scala-sbt is an open-source build tool designed for Scala and Java projects, known for its powerful project configuration and dependency management capabilities.

After these setups, I downloaded and set-up Spark 2.4.7 and Apache Hadoop 2.7.1. Hadoop was employed for its robust distributed storage and processing features. Its Hadoop Distributed File System (HDFS) provided a dependable storage solution for large data sets, crucial for this project. Configuring Hadoop's key components to work smoothly with Spark was a critical step. The project's primary tool, Apache Spark, excelled in processing vast datasets, making it ideal for Hot Zone and Hot Cell analysis. The integration of Spark with Hadoop's HDFS and fine-tuning its settings were key aspects of setting up Spark on Ubuntu 20.04 for optimal performance.

Hot Zone Analysis

Hot zone analysis is a spatial data analysis technique used to identify areas of high activity or concentration within a given dataset. In the context of projects like traffic or urban planning, this analysis involves mapping and examining regions with a high density of certain elements, such as traffic accidents or population clusters. The method typically involves overlaying different data sets, like points of interest and geographical zones, and calculating the density or frequency of these points within specific zones. This analysis is valuable for understanding spatial patterns and making data-driven decisions in urban development, public safety, and resource allocation.

In the Hot Zone Analysis, the focus was on examining spatial data to pinpoint areas with heightened point concentrations, known as hot zones. The Scala-based process encompassed a range join between datasets of rectangles and points. The intensity of each rectangle, or its "hotness," was assessed based on the point count it has.

A crucial Scala function, ***ST_Contains***, within the **HotzoneUtils.scala** file, played a central role. This Scala function was designed to ascertain whether a point resided within a specified rectangle, using mathematical evaluations to verify if the point's coordinates lay within the rectangle's boundaries. The outcomes of this function contributed to a group-by operation, tallying the points in each rectangle to delineate the hot zones. ***The output csv file contains every zone, along with its corresponding count, arranged in ascending order based on the "rectangle" string.***

HotzoneUtils.scala

```
package cse512

object HotzoneUtils {

  def ST_Contains(queryRectangle: String, pointString: String ): Boolean = {
    // check for validity of input data, i.e whether input is null or empty
    if(queryRectangle == null || queryRectangle.isEmpty || pointString == null || pointString.isEmpty) {
      return false
    }

    val rectangleCoordinates = queryRectangle.split(",")
    val pointCoordinates = pointString.split(",")

    // check whether the points have correct number of coordinates
    if(rectangleCoordinates.length < 4 || pointCoordinates.length < 2) {
      return false
    }

    val xOfCorner1 = rectangleCoordinates(0).trim.toDouble
    val yOfCorner1 = rectangleCoordinates(1).trim.toDouble
    val xOfCorner2 = rectangleCoordinates(2).trim.toDouble
    val yOfCorner2 = rectangleCoordinates(3).trim.toDouble
    val pointX = pointCoordinates(0).trim.toDouble
    val pointY = pointCoordinates(1).trim.toDouble

    //check whether the rectangle contains given point
    if(pointX >=  math.min(xOfCorner1, xOfCorner2) && pointX <= math.max(xOfCorner1, xOfCorner2)
      && pointY >= math.min(yOfCorner1, yOfCorner2) && pointY <= math.max(yOfCorner1, yOfCorner2)) {
      return true
    }

    return false
  }

}
```

HotzoneAnalysis.scala

```
object HotzoneAnalysis {

  Logger.getLogger("org.spark_project").setLevel(Level.WARN)
  Logger.getLogger("org.apache").setLevel(Level.WARN)
  Logger.getLogger("akka").setLevel(Level.WARN)
  Logger.getLogger("com").setLevel(Level.WARN)

  def runHotZoneAnalysis(spark: SparkSession, pointPath: String, rectanglePath: String): DataFrame = {

    var pointDf = spark.read.format("com.databricks.spark.csv").option("delimiter",";").option("header","false").load(pointPath);
    pointDf.createOrReplaceTempView("point")

    // Parse point data formats
    spark.udf.register("trim",(string : String)=>(string.replace("(", "").replace(")", "")))
    pointDf = spark.sql("select trim(_c5) as _c5 from point")
    pointDf.createOrReplaceTempView("point")

    // Load rectangle data
    val rectangleDf = spark.read.format("com.databricks.spark.csv").option("delimiter","\t").option("header","false").load(rectanglePath);
    rectangleDf.createOrReplaceTempView("rectangle")

    // Join two datasets
    spark.udf.register("ST_Contains",(queryRectangle:String, pointString:String)=>(HotzoneUtils.ST_Contains(queryRectangle, pointString)))
    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")

        // Group the result of Join operation in the previous step by rectangle and sort according to rectangle
    // Obtain the count of the number of points that lies within each rectangle
    // Return the DataFrame with rectangle and the corresponding count of number of points that lies within the rectangle
    // Use persist() to optimize query involving long chains of transformations
    // Use coalesce() to combine results from all the partitions into a single partition
    val pointsCountSortedByRectangleDf = spark.sql("select rectangle,count(point) as numberOfPoints from joinResult group by rectangle order by rectangle asc").persist().coalesce(1)
    pointsCountSortedByRectangleDf.createOrReplaceTempView("HotZoneResult")
    pointsCountSortedByRectangleDf.show()

        return pointsCountSortedByRectangleDf
  }

}
```

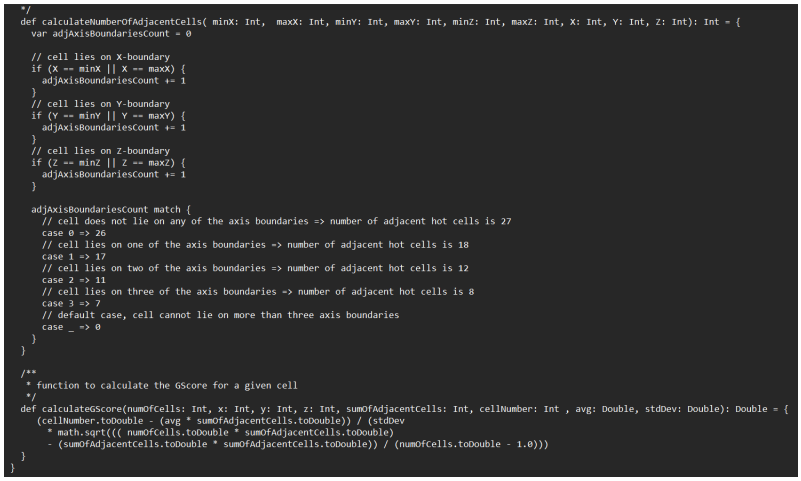
Hot Cell Analysis

Hot cell analysis is a technique used in spatial data analysis to identify statistically significant spatial clusters, or 'hot cells', within a dataset. This approach is particularly useful in fields like epidemiology, crime analysis, and environmental studies. It involves analyzing spatial and temporal data to locate areas with unusually high or intense activity. By identifying these hot cells, researchers and analysts can gain insights into patterns and trends, facilitating targeted responses or interventions.

Based on the ACM SIGSPATIAL GISCUP 2016 challenge, the Hot Cell Analysis involved applying spatial statistics to spatiotemporal large data. In order to find statistically significant hot areas in the NYC Taxi Trip dataset, the Getis-Ord statistic was the main emphasis.

For every space-time cell, the Getis-Ord statistic had to be calculated as part of the implementation. SQL queries were utilized to ascertain the count of every cell, and the sum and square of these counts were then computed. This aided in the computation of the standard deviation (S value) and mean. **HotcellUtils.scala**'s ***calculateNumberOfAdjacentCells*** and **calculateGScore** functions were essential. They computed the Getis-Ord statistic for every cell and counted the number of neighbouring cells. The most important hotspot was then determined by sorting the findings in descending order. ***The output csv file contains the coordinates of top 50 cells with the highest heat intensity arranged in descending order based on their G score, without actually displaying the G score values.***

HotcellUtils.scala



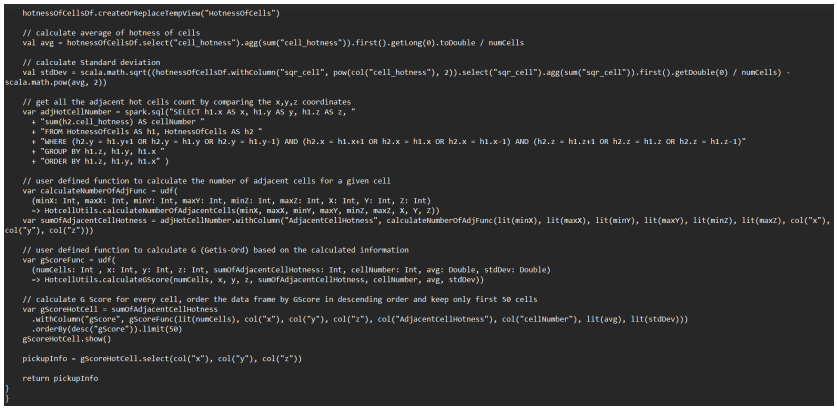
```
  */
  def calculateNumberOfAdjacentCells( minX: Int,  maxX: Int, minY: Int, maxY: Int, minZ: Int, maxZ: Int, X: Int, Y: Int, Z: Int): Int = {
    var adjAxisBoundariesCount = 0

    // cell lies on X-boundary
    if (X == minX || X == maxX) {
      adjAxisBoundariesCount += 1
    }
    // cell lies on Y-boundary
    if (Y == minY || Y == maxY) {
      adjAxisBoundariesCount += 1
    }
    // cell lies on Z-boundary
    if (Z == minZ || Z == maxZ) {
      adjAxisBoundariesCount += 1
    }

    adjAxisBoundariesCount match {
      // cell does not lie on any of the axis boundaries => number of adjacent hot cells is 27
      case 0 => 26
      // cell lies on one of the axis boundaries => number of adjacent hot cells is 18
      case 1 => 17
      // cell lies on two of the axis boundaries => number of adjacent hot cells is 12
      case 2 => 11
      // cell lies on three of the axis boundaries => number of adjacent hot cells is 8
      case 3 => 7
      // default case, cell cannot lie on more than three axis boundaries
      case _ => 0
    }
  }

  /**
   * function to calculate the GScore for a given cell
   */
  def calculateGScore(numOfCells: Int, x: Int, y: Int, z: Int, sumOfAdjacentCells: Int, cellNumber: Int , avg: Double, stdDev: Double): Double = {
      (cellNumber.toDouble - (avg * sumOfAdjacentCells.toDouble)) / (stdDev
        * math.sqrt((( numOfCells.toDouble * sumOfAdjacentCells.toDouble)
        - (sumOfAdjacentCells.toDouble * sumOfAdjacentCells.toDouble)) / (numOfCells.toDouble - 1.0)))
  }
}
```

HotcellAnalysis.scala



```
    hotnessOfCellsDf.createOrReplaceTempView("HotnessOfCells")

    // calculate average of hotness of cells
    val avg = hotnessOfCellsDf.select("cell_hotness").agg(sum("cell_hotness")).first().getLong(0).toDouble / numCells

    // calculate standard deviation
    val stdDev = scala.math.sqrt((hotnessOfCellsDf.withColumn("sqr_cell", pow(col("cell_hotness"), 2)).select("sqr_cell").agg(sum("sqr_cell")).first().getDouble(0) / numCells) -
scala.math.pow(avg, 2))

    // get all the adjacent hot cells count by comparing the x,y,z coordinates
    var adjHotCellNumber = spark.sql("SELECT h1.x AS x, h1.y AS y, h1.z AS z, "
      + "sum(h2.cell_hotness) AS cellNumber "
      + "FROM HotnessOfCells AS h1, HotnessOfCells AS h2 "
      + "WHERE (h2.y = h1.y+1 OR h2.y = h1.y OR h2.y = h1.y-1) AND (h2.x = h1.x+1 OR h2.x = h1.x OR h2.x = h1.x-1) AND (h2.z = h1.z+1 OR h2.z = h1.z OR h2.z = h1.z-1)"
      + "GROUP BY h1.z, h1.y, h1.x "
      + "ORDER BY h1.z, h1.y, h1.x" )

    // user defined function to calculate the number of adjacent cells for a given cell
    var calculateNumberOfAdjFunc = udf(
      (minX: Int, maxX: Int, minY: Int, maxY: Int, minZ: Int, maxZ: Int, X: Int, Y: Int, Z: Int)
      => HotcellUtils.calculateNumberOfAdjacentCells(minX, maxX, minY, maxY, minZ, maxZ, X, Y, Z))
    var sumOfAdjacentCellHotness = adjHotCellNumber.withColumn("AdjacentCellHotness", calculateNumberOfAdjFunc(lit(minX), lit(maxX), lit(minY), lit(maxY), lit(minZ), lit(maxZ), col("x"),
col("y"), col("z")))

    // user defined function to calculate G (Getis-Ord) based on the calculated information
    var gScoreFunc = udf(
      (numCells: Int , x: Int, y: Int, z: Int, sumOfAdjacentCellHotness: Int, cellNumber: Int, avg: Double, stdDev: Double)
      => HotcellUtils.calculateGScore(numCells, x, y, z, sumOfAdjacentCellHotness, cellNumber, avg, stdDev))

    // calculate G Score for every cell, order the data frame by GScore in descending order and keep only first 50 cells
    var gScoreHotCell = sumOfAdjacentCellHotness
      .withColumn("gScore", gScoreFunc(lit(numCells), col("x"), col("y"), col("z"), col("AdjacentCellHotness"), col("cellNumber"), lit(avg), lit(stdDev)))
      .orderBy(desc("gScore")).limit(50)
    gScoreHotCell.show()

    pickupInfo = gScoreHotCell.select(col("x"), col("y"), col("z"))

    return pickupInfo
}
}
```

II. LESSONS LEARNED:

I learned the below lessons:

1. **Efficiency in Big Data Processing:** The project underscored the effectiveness of Apache Spark in handling big data. Its in-memory processing capabilities were key in efficiently performing complex spatial computations.
2. **Navigating Spatial Data Complexity:** It highlighted the intricacies in processing spatial data, stressing the need for accurate computations and a deep understanding of spatial relationships.
3. **Synergy of Technologies:** The project underscored the importance of integrating various technologies like Hadoop, Spark, Scala, and Windows/Ubuntu, each contributing uniquely to the project's success.
4. **Practical Application of Spatial Statistics**: It offered hands-on experience in applying spatial statistics to real-world data, showing how statistical analysis can uncover significant patterns and hotspots in voluminous datasets.
5. **Integrating Complex Systems:** The integration of Spark with Hadoop's HDFS taught me the significance of seamless system integration for handling large datasets.
6. **Spatial Data Analysis Skills:** Conducting both Hot Zone and Hot Cell analyses enhanced my understanding of spatial data processing and the application of statistical methods in identifying patterns.
7. **Practical Application of Theoretical Knowledge:** Applying concepts like the Getis-Ord statistic in a real-world context like the NYC Taxi Trip dataset was a practical demonstration of theoretical knowledge.
8. **Problem-Solving and Debugging:** Throughout the project, tackling various challenges honed my problem-solving and debugging skills, especially in a complex, multi-component setup.

III. OUTPUTS:

These are the outputs from csv files which I got after building the jar and testing the code.

Hot Zone Analysis

	A	B	C
1	-73.78941	1	
2	-73.79363	1	
3	-73.79565	1	
4	-73.79651	1	
5	-73.79729	1	
6	-73.80203	8	
7	-73.80577	3	
8	-73.81523	2	
9	-73.81638	1	
10	-73.81913	1	
11	-73.82592	2	
12	-73.82657	1	
13	-73.83270	200	
14	-73.83946	3	
15	-73.84013	4	
16	-73.84081	1	
17	-73.84233	2	
18	-73.84314	2	
19	-73.84947	2	
20	-73.86109	21	
21	-73.86204	16	
22	-73.86448	1	
23	-73.86791	1	
24	-73.86923	136	
25	-73.87365	1	
26	-73.87509	3	
27	-73.87638	67	
28	-73.87824	10	

Hot Cell Analysis

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

IV. RESULTS:

Please find the below screenshots which shows I tested my jar and code in ubuntu.