CSE 511: Data Processing at Scale – Fall 2023

A Technical Report on

Project 2: Individual result - Hot Spot Analysis

Sai Likhith Yadav Seera
Computer Science
School of Computing and Augmented Intelligence
Arizona State University
Tempe, United States
sseera@asu.edu

I. Reflection

The main agenda of this project is to implement two different spatial hot spot analysis tasks in Hadoop and Apache Spark.

Hot zone analysis: This task involves conducting a range join operation between a dataset containing rectangles and another containing points. For each rectangle, the objective is to determine the count of points residing within it. A 'hotter' rectangle indicates a higher density of points within it. Therefore, the goal is to compute the relative density, or 'hotness,' of all the rectangles based on the number of points they encompass.

Hot cell analysis: This task involves conducting a range join operation between a dataset containing rectangles and another containing points. The objective is to determine, for each rectangle, the count of points it encompasses. A 'hotter' rectangle signifies a higher count of enclosed points. Thus, the task focuses on computing the hotness metric for all rectangles based on the number of points each contains.

The project aims to analyze NYC Taxi trip data to extract valuable insights, such as the number of pickups in specific zones and statistical assessments using the Getis-Ord function to calculate G-Scores for particular cells. These analyses play a pivotal role in enhancing location-based applications. For instance, ride-sharing services like Uber rely on requests from various locations for taxi pickups. By identifying areas with high pickup frequencies and determining neighboring zones with increased pickups, service efficiency can be enhanced, leading to reduced taxi pickup times. Our project goal encompasses completing two crucial functions: Hot Zone Analysis and Hot Cell Analysis. Hot Zone Analysis involves identifying the number of points within a zone and defining its 'hotness' based on point density. Hot Cell Analysis entails conducting statistical analysis by calculating Z-scores for each cell in the dataset.

II. Functionality

A. *HotzoneAnalysis.scala:*

The approach involves checking whether a point falls within a given rectangle and counting the number of points within each rectangle. To achieve this, we've established a function called ST_Contains within Hot Zone Utils, which validates if a point resides inside a rectangle. The next step is to create a view using the rectangle and point datasets and perform a join based on the output from ST_Contains. This resulting joined dataframe comprises the bounding rectangle and the associated point if ST_Contains evaluates to true. With this established relationship and the relevant columns, we proceed to write Spark SQL queries on this data to determine the count of points within each rectangle. The resultant count information is then saved into an output file for further analysis or use.

Implementation: Figure 1 indicates the implementation of **runHotZoneAnalysis** function.

```
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")

  // YOU NEED TO CHANGE THIS PART
  val returnDf = spark.sql("select rectangle, COUNT(point) from joinResult group by rectangle order by rectangle").coalesce(1)

  // return joinDf // YOU NEED TO CHANGE THIS PART
  return returnDf
}
```

Fig. 1: Implementation of runHotZoneAnalysis function.

B. *HotzoneUtils.scala:*

The functionality of ST_contains revolves around determining whether a given point lies within the provided Minimum Bounding Rectangle (MBR). This MBR can be defined either by four points as four vertices or by two points that represent the rectangle's opposing corners. The aim of the function is to return 'true' if the point is within the rectangle. However, it's crucial to note that if the point aligns precisely with the border of the rectangle, it is considered 'false' according to this logic. The method involves a simple logic check: it verifies whether the x-coordinate of the point falls within the range of x-coordinates defining the rectangle and similarly examines the y-coordinate of the point within the y-coordinate range of the rectangle. If both conditions are met, indicating that the point satisfies these coordinate boundaries, the function confirms that the point is inside the rectangle and returns 'true' for use in Hot Zone Analysis.

Implementation: Figure 2 indicates the implementation of **ST_Contains** function.

```
def ST_Contains(queryRectangle: String, pointString: String ): Boolean = {
  // YOU NEED TO CHANGE THIS PART
  val rectangle_coordinates = queryRectangle.split(",")
  val target_point_coordinates = pointString.split(",")

  val point_lat: Double = target_point_coordinates(0).trim.toDouble
  val point_lon: Double = target_point_coordinates(1).trim.toDouble

  val rect_lat1:Double = math.min(rectangle_coordinates(0).trim.toDouble, rectangle_coordinates(2).trim.toDouble)
  val rect_lat2:Double = math.max(rectangle_coordinates(0).trim.toDouble, rectangle_coordinates(2).trim.toDouble)

  val rect_lon1:Double = math.min(rectangle_coordinates(1).trim.toDouble, rectangle_coordinates(3).trim.toDouble)
  val rect_lon2:Double = math.max(rectangle_coordinates(1).trim.toDouble, rectangle_coordinates(3).trim.toDouble)

  if ((point_lat >= rect_lat1) && (point_lon >= rect_lon1) && (point_lat <= rect_lat2) && (point_lon <= rect_lon2))
  {
    return true
  }
  else
  {
    return false
  }
  // return true // YOU NEED TO CHANGE THIS PART
}
```

Fig. 2: Implementation of ST_Contains function.

C. *HotcellAnalysis.scala:*

The primary function of Hot Cell Analysis is to gauge the relative 'hotness' of each cell within a given area. This analysis relies on the Getis-Ord G* score, which extends beyond solely assessing an individual cell's density by considering its neighboring cells as well. For instance, suppose a particular cell exhibits a high density of taxi pickups. However, if the neighboring cells lack similar density, merely increasing the number of taxis in that specific region might not significantly improve overall cab utilization for a service like Uber. By taking into account the characteristics and features of all adjacent cells, we can discern whether a particular cell qualifies as 'hot'. This information aids in identifying areas that could benefit from specific interventions or improvements to enhance their overall performance.

Implementation: Figure 3 indicates the implementation of **runHotcellAnalysis** function.

```
def runHotcellAnalysis(spark: SparkSession, pointPath: String): DataFrame =
{
  // Load the original data from a data source
  var pickupInfo = spark.read.format("com.databricks.spark.csv").option("delimiter",";").option("header","false").load(pointPath);
  pickupInfo.createOrReplaceTempView("nyctaxitrips")
  pickupInfo.show()

  // Assign cell coordinates based on pickup points
  spark.udf.register("CalculateX",(pickupPoint: String)=>((
    HotcellUtils.CalculateCoordinate(pickupPoint, 0)
    )))
  spark.udf.register("CalculateY",(pickupPoint: String)=>((
    HotcellUtils.CalculateCoordinate(pickupPoint, 1)
    )))
  spark.udf.register("CalculateZ",(pickupTime: String)=>((
    HotcellUtils.CalculateCoordinate(pickupTime, 2)
    )))
  pickupInfo = spark.sql("select CalculateX(nyctaxitrips._c5),CalculateY(nyctaxitrips._c5), " +
                         "CalculateZ(nyctaxitrips._c1) from nyctaxitrips")
  var newCoordinateName = Seq("x", "y", "z")
  pickupInfo = pickupInfo.toDF(newCoordinateName:_*)
  pickupInfo.show()

  // Define the min and max of x, y, z
  val minX = -74.50/HotcellUtils.coordinateStep
  val maxX = -73.70/HotcellUtils.coordinateStep
  val minY = 40.50/HotcellUtils.coordinateStep
  val maxY = 40.90/HotcellUtils.coordinateStep
  val minZ = 1
  val maxZ = 31
  val numCells = (maxX - minX + 1)*(maxY - minY + 1)*(maxZ - minZ + 1)

// YOU NEED TO CHANGE THIS PART
pickupInfo.createOrReplaceTempView("pickupInfo")

val aggregatedData_df = spark.sql(
  s"""
  SELECT x, y, z, COUNT(*) AS total_count
  FROM pickupInfo
  WHERE x BETWEEN $minX AND $maxX
    AND y BETWEEN $minY AND $maxY
    AND z BETWEEN $minZ AND $maxZ
  GROUP BY x, y, z
  """
).persist()

aggregatedData_df.createOrReplaceTempView("sample_data")

val summaryStats_df = spark.sql(
  """
  SELECT SUM(total_count) AS totalPickupCount,
         SUM(total_count * total_count) AS totalPickupCountSquared
  FROM sample_data
  """
).persist()

val totalPickupCount = summaryStats_df.first().getLong(0).toDouble
val totalPickupCountSquared = summaryStats_df.first().getLong(1).toDouble

val averagePickupCount = totalPickupCount / numCells
val pickupCountStdDev = Math.sqrt((totalPickupCountSquared / numCells) - (averagePickupCount * averagePickupCount))

val cellNeighborsStats_df = spark.sql(
  """
  SELECT S1.x AS x, S1.y AS y, S1.z AS z,
         COUNT(*) AS numNeighbors,
         SUM(S2.total_count) AS sigma
  FROM sample_data AS S1
  INNER JOIN sample_data AS S2
  ON (ABS(S1.x - S2.x) <= 1 AND ABS(S1.y - S2.y) <= 1 AND ABS(S1.z - S2.z) <= 1)
  GROUP BY S1.x, S1.y, S1.z
  """).persist()
  cellNeighborsStats_df.createOrReplaceTempView("cellNeighborsStats_df")

  spark.udf.register("calculateSpatialZScore", HotcellUtils.calculateSpatialZScore(_:Double, _:Double, _:Int, _:Int, _:Int))

  val cellHotnessScores_df = spark.sql(
    s"""
    SELECT x, y, z,
           calculateSpatialZScore($averagePickupCount, $pickupCountStdDev, numNeighbors, sigma, $numCells) AS ZScore
    FROM cellNeighborsStats_df
    """
  )

  cellHotnessScores_df.createOrReplaceTempView("cellHotnessScores_df")

  val sortedHotCells_df = spark.sql("SELECT x, y, z FROM cellHotnessScores_df ORDER BY ZScore DESC")

  // return pickupInfo // YOU NEED TO CHANGE THIS PART
  return sortedHotCells_df
}
```

Fig. 3: Implementation of runHotcellAnalysis function.

D. *HotcellUtils.scala:*

This function primarily computes the Z score for each cell by utilizing the mean, standard deviation of the cell's points, and the cell's weight with respect to the neighboring cells. To understand the Z score calculation, how the entire area is divided into multiple cells and how the value for each cell is determined is being discussed.

1) Cell Value Computation: Every cell in this context is characterized by three attributes:

1. X: Denoting latitudes
2. Y: Representing the longitudinal range
3. Z: Signifying the day of the pickup, within a range of 0 to 31, considering a month.

The cell size is set at 0.01*0.01 concerning latitude and longitude, resulting in the calculation of the total number of cells. This calculation is performed based on a sample from the extensive dataset, defining the range of X, Y, and Z. Then, we calculate the total count of points within each cell and compute the cumulative sum of all these points via an SQL query.

2) Mean & Deviation Calculation: With the provided values, sums, and sums of squares, we derive the mean and standard deviation necessary for computing the Z-score using the following formulas.

Mean (μ): $\mu = \frac{\sum_{i=1}^{n} X_i}{n}$

Standard Deviation (σ): $\sigma = \sqrt{\frac{\sum_{i=1}^{n} (X_i - \mu)^2}{n}}$

These formulas represent the calculation of the mean and standard deviation, respectively, where X_i denotes the individual data points and n is the total number of data points.

3) Z Score Calculation: Z score is calculated using the below formula

Z score (Z) = (X_i - μ) / σ

In this formula, 'X_i' represents the individual data point, 'μ' is the mean, 'σ' is the standard deviation, and 'Z' is the Z-score.

The Z-score formula determines the deviation of a specific attribute value (x) of a cell from the mean (μ), divided by the standard deviation (σ). It helps in assessing whether features with high or low values are clustered together by providing a measure of how far an individual data point is from the mean in terms of standard deviations.

In spatial analysis, the spatial weight (w) between two cells (i and j) is determined based on their proximity or adjacency. Two cells are considered neighbors if the difference between their X, Y, and Z coordinates is less than or equal to 1. This spatial relationship helps in defining the interconnectedness or adjacency between cells and influences the spatial weight used in the context of Z-score calculations.

4) Neighborhood Acquisition: The neighborhood relationships are established by uniformly subdividing latitudes and longitudes concerning past, present, and subsequent time periods. This subdivision is accomplished by conducting a self-join operation on the sample results, verifying if the difference is less than or equal to one. Once the neighbor_df view is obtained, the Z-scores are computed for each cell and stored in the results. Following this, the attributes of the relation are selected based on their Z-scores in decreasing order and stored in the designated output location.

Implementation: Figure 4 indicates the implementation of **CalculateCoordinate**.

```
val coordinateStep = 0.01

def CalculateCoordinate(inputString: String, coordinateOffset: Int): Int =
{
  // Configuration variable:
  // Coordinate step is the size of each cell on x and y
  var result = 0
  coordinateOffset match
  {
    case 0 => result = Math.floor((inputString.split(",")(0).replace("(","").toDouble/coordinateStep)).toInt
    case 1 => result = Math.floor(inputString.split(",")(1).replace(")","").toDouble/coordinateStep).toInt
    // We only consider the data from 2009 to 2012 inclusively, 4 years in total. Week 0 Day 0 is 2009-01-01
    case 2 => {
      val timestamp = HotcellUtils.timestampParser(inputString)
      result = HotcellUtils.dayOfMonth(timestamp) // Assume every month has 31 days
    }
  }
  return result
}

def timestampParser (timestampString: String): Timestamp =
{
  val dateFormat = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss")
  val parsedDate = dateFormat.parse(timestampString)
  val timeStamp = new Timestamp(parsedDate.getTime)
  return timeStamp
}

def dayOfYear (timestamp: Timestamp): Int =
{
  val calendar = Calendar.getInstance
  calendar.setTimeInMillis(timestamp.getTime)
  return calendar.get(Calendar.DAY_OF_YEAR)
}

def dayOfMonth (timestamp: Timestamp): Int =
{
  val calendar = Calendar.getInstance
  calendar.setTimeInMillis(timestamp.getTime)
  return calendar.get(Calendar.DAY_OF_MONTH)
}

// YOU NEED TO CHANGE THIS PART
def calculateSpatialZScore(averageSpatialValue: Double,
                           spatialStandardDeviation: Double, adjacentCellCount: Int,
                           totalNeighborValuesSum: Int, totalGridCellCount: Int): Double =
{
  val zScoreNume = totalNeighborValuesSum - (averageSpatialValue * adjacentCellCount)
  val zScoreDeno = spatialStandardDeviation * Math.sqrt((totalGridCellCount *
                                                         adjacentCellCount - adjacentCellCount *
                                                         adjacentCellCount) / (totalGridCellCount-1))
  return zScoreNume / zScoreDeno
}
```

Fig. 4: Implementation of CalculateCoordinate function.

III. Acknowledgements

I would like to express my gratitude to Dr. Samira Ghayekhloo for her guidance in explaining spatial data, as well as to Yujian Xiong and Qixian Zhao for their support in in addressing doubts or queries related to the subject.

References

[1] ACM SIGSPATIAL Cup 2016
https://sigspatial2016.sigspatial.org/giscup2016/prob.

[2] Documentation on ST_Contains
https://postgis.net/docs/ST_Contains.html.

[3] Apache Spark Documentation
https://spark.apache.org/docs/3.5.0/.