Project 2: Hot Spot Analysis

Vighnesh Sridhar

Reflection:

In this project, I performed hot spot analysis using spatial queries on geographical locations and a taxi company. The project is divided into two portions. The first part observed the hotness or how many points were in rectangles. The basic approach is to make comparisons between the point values to see whether a rectangle contains a point. I implemented the function ST_contains, which checks whether a point is inside of a rectangle. I used the split function to parse the input string. Then, I made comparisons to check whether the condition is satisfied. Finally, I used data frames to count the number of points and performed sorting based on the rectangle string.

The second part uses data from a taxi company where I performed spatial queries on three dimensions (x, y, and z) to compute the G score. I used spark SQL queries to find all the parameters for the G score equation. This included the mean, the standard deviation, the sum of the weights of adjacent cells, and the sum of the pickups of adjacent cells. The weight is defined as the number of adjacent cells to cell i, while the attribute value x_j is the number of pickups in the j^{th} cell. I included formulas in HotcellUtils.scala which computed the number of adjacent cells to a cell and the G score of a cell. To compute the number of adjacent cells, the boundaries of the space must be defined. Then, for the x, y, and z coordinates of a given cell, I check how many of these are equal to the boundary. If none of them are equal to a boundary point, it's an inside point, in which case the cell would have 26 adjacent cells. If one of them is equal to a boundary point, it would be a face point and would have 17 adjacent cells. Similarly, an edge point would have 11 adjacent cells and a corner point would have 7 adjacent cells.

The final output for the first part included a list of rectangles sorted by their values along with the point count. In the second part, the results included the x, y, and z coordinates sorted in descending order by G scores.

Lessons Learned:

The main lesson that I learned is that I should first think about how to approach the project before I start writing code. For example, for the HotcellAnalysis.scala file, I tried to solve the problems by using data frames rather than spark SQL queries, which made it difficult to do. I ended up wasting a lot of time because of this. Another thing I learned is how to debug effectively. I used print statements, and I tried different things to solve syntax errors in SQL queries or otherwise. Also, sometimes the issue in the code can be something very minor or small. I looked online to find solutions to syntax errors that occurred in my code, which was very helpful.

One issue I faced was in the HotzoneAnalysis.scala file. I was performing the sorting on the rectangles data frame, but for some reason, it was not sorted in the output. Then, I realized that performing a join or deduplicating the data frame could mess up the sorting. Hence, the solution I found was to sort in the end, after the join and deduplication.

Implementation:

The image below shows my implementation for ST_contains.

```
object HotzoneUtils {

  def ST_Contains(queryRectangle: String, pointString: String ): Boolean = {
    // println ("test    test");
    // println ("\n");
    val rectangle_points = queryRectangle.split(",");
    // println ("test 1   test 1");
    val p1_x = rectangle_points(0).toDouble;
    val p1_y = rectangle_points(1).toDouble;
    val p2_x = rectangle_points(2).toDouble;
    val p2_y = rectangle_points(3).toDouble;

    // println ("test 2   test 2");
    val point = pointString.split(",");
    val p_x = point(0).toDouble;
    val p_y = point(1).toDouble;

    // println ("test 3   test 3");
    val left_x = p1_x.min(p2_x);
    val right_x = p1_x.max(p2_x);
    val bottom_y = p1_y.min(p2_y);
    val top_y = p1_y.max(p2_y);

    // println ("test 4   test 4");
    if (p_x >= left_x && p_x <= right_x && p_y >= bottom_y && p_y <= top_y){
        return true;
    }

    // println ("test 5   test 5");

    return false; // YOU NEED TO CHANGE THIS PART
  }

  // YOU NEED TO CHANGE THIS PART

}
```

The ST_contains function checks whether a point is inside of a rectangle. The ST_contains function takes in two strings: a rectangle string and a point string. The rectangle string contains four values representing two corner points of the rectangle. The point string contains two values that represent the coordinates of the point. I used a split with the comma delimiter to obtain the coordinates for all three points. Then, I took the min and max x, and the min and max y of the coordinates of the rectangle corner points. I checked if the coordinates of the point fall within the boundaries of the rectangle by making the necessary comparisons. For example, I checked if the x coordinate of point is greater than min x but less than max x. If the point falls within the rectangle, I returned true; otherwise, I returned false.

```
    // YOU NEED TO CHANGE THIS PART
    val joinDfCoord = joinDf.withColumn("x1", split(col("rectangle"), ",")(0).cast("double")).withColumn("y1", split(col("rectangle"), ",")(1).
    val countDf = joinDfCoord.groupBy("rectangle").agg(count("point").as("point_count"))
    val countDf_drop = countDf.drop(col("point"))
    val deduplicatedDf = joinDfCoord.dropDuplicates("rectangle")
    val resultDf = deduplicatedDf.join(countDf, Seq("rectangle"))
    val resultSortedDf = resultDf.sort(desc("x1"), desc("y1"), desc("x2"), desc("y2"))
    val resultFilteredDf = resultSortedDf.select("rectangle", "point_count").coalesce(1)

    return resultFilteredDf // YOU NEED TO CHANGE THIS PART
  }

}
```

In the HotzoneAnalysis.scala file, I count the total number of points in each rectangle and sort it based on the rectangle string. First, I split the rectangle strings in the column into four separate columns: x1, y1, x2, and y2. To do this I used the split function and the with column function. Then, I grouped the data frame by rectangles and used the aggregate function to count the number of points in each rectangle as point count. I dropped the duplicate rectangles from the data frame, and I performed a join between the count data frame and the coordinate data frame. I sorted the resulting data frame in descending order by x1, y1, x2, and y2 and then selected the rectangle and count column. I added .coalesce(1) to the last query the resulting data frame contained each rectangle and its hotness, sorted in descending order.

In the HotcellAnalysis.scala file, I start by selecting x, y, z, and count(*) as x_j. When I group by x, y, and z, the count(*) will give the attribute value. I employed the create or replace temp view function for this and the other tables in this file. I sum x_j and x_j * x_j to obtain the summation of the attribute value and summation of the attribute value squared, respectively. I used these values to compute the mean and standard deviation. Next, I computed the parameters for weight and weight times attribute value. To do this, I performed a join between cells c1 and cells c2 with the conditions that x, y, and z of c1 must be within 1 unit of x, y, and z of c2. After selecting c1.x, c1.y, and c1.z, I used the function from utils that computes the number of adjacent cells to find the weight of the cell c1 and I summed the x_j to find the summation of the weight of cell i multiplied by the attribute value of adjacent cells.

Finally, I computed the G score by passing the five parameters calculated: the sum of the attribute value of adjacent cells, the mean of the data, the weight of a cell, the standard deviation of the data, and the number of cells. The formula for the G score is shown below as part of the HotcellUtils.scala file. The final data frame contains the x, y, and z coordinates sorted in descending order by G score.

```
  def computeGScore(sum_w_ij_x_j: Double, mean: Double, weight: Double, S: Double, numCells: Int): Double =
  {

    val G_score = ((sum_w_ij_x_j.toDouble - mean * weight.toDouble).toDouble / (S * Math.sqrt((numCells.toDouble *
    weight.toDouble - weight.toDouble * weight.toDouble) / (numCells.toDouble - 1)).toDouble).toDouble).toDouble
    return G_score
  }

}
```

```
  spark.udf.register("computeGScore", (sum_w_ij_x_j: Double, mean: Double, weight: Double, S: Double, numCells: Int) => (HotcellUtils.computeGScore(sum_w_ij_x_j, mean, weight,
  S, numCells)))

  val Gscores = spark.sql(s"""
  SELECT cellX, cellY, cellZ, computeGScore(${adjacentCells("x_j_times_w_ij")}, $X, ${adjacentCells("adjacentCellCount")}, $S, $numCells) AS Gscore
  FROM adjacentCells
  ORDER BY Gscore desc
""")



  Gscores.createOrReplaceTempView("Gscores")
  val resultDf = spark.sql("select cellX, cellY, cellZ from Gscores")
  resultDf.createOrReplaceTempView("resultDf")

  return resultDf // YOU NEED TO CHANGE THIS PART
}
}
```