Reflection:

In this project we were tasked with analyzing a New York Taxi dataset to identify hot spots. There were two types of hot spot queries we were tasked with fulfilling; HotZoneAnalysis and HotCellAnalysis. HotZoneAnalysis was a query in which our function was passed a list of rectangles and a list of points and the result was meant to be those rectangles which contained the most points. The Hot Cell analysis is based on the ACM sigspatial competition problem in which the goal was to identify the top 50 ‘hottest’ locations. For each point we calculate their Getis-Ord statistic in order to rank them on the basis of ‘hotness’.

This project was a difficult one, which required setting up an Apache Spark Environment and learning to write scala code and translating abstract equations into functional queries on our database. I learned how to write UDFs in scala, commit tables to a virtual db, and combine and coordinate different queries in order to retrieve and compute the desired result. What follows is an overview of the functions we made as well as the lessons learned along the way.

HotZoneAnalysis:

HotZoneAnalysis involves calculating the hotness of rectangles, where each rectangle represents a zone and a point represents the pick-up location of a New York taxi trip. To perform this task, I needed geospatial data related to the rectangles and points. For each rectangle, I obtained the longitude and latitude of the two opposite corner points, as well as the longitude and latitude of the point to be tested. To determine if the point is inside the rectangle, I utilized the ST_Contains(rec_string, point_string) function, which returns a boolean flag. The input for this function includes a query rectangle and a point string in the format of ("2.3, 5.1, 6.8, 8.9", "5.5, 5.5"). To implement this function correctly, I first determined the minimum and maximum corners of the rectangle by comparing the longitudes and latitudes using the "math.min" and "math.max" functions.

```
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
  }
```

Using ST_Contains I queried the rectangle and point tables and combined their result in a dataframe called joinDF and submitted it to the DB in order to query it for the rectangles containing the most points. Finally, I returned each rectangle with its coordinates and the count of points located inside it by executing the SQL query: "select rectangle, count(point) as numberOfPoints from joinResult group by rectangle order by rectangle asc".

HotCellAnalysis:

HotCellAnalysis, on the other hand, calculates the hotness of a given cell, which is defined by its longitude(x), latitude(y), and dateTime(z). In this analysis, the goal is to calculate the Getis-Ord statistic, which measures the number of pickups for a specific location on a particular day. To perform this calculation, I created two UDFs to assist in the process: The calculateNumberOfAdjacentCells function, and the calculateGScore function.

CalculateNumberOfAdjacentCells calculates the number of adjacent cells for a given cell based on its position within the specified boundaries. The function determines if the cell lies on the X, Y, or Z boundaries and increments a count accordingly. The function then uses the adjAxisBoundariesCount value to determine the number of adjacent hot cells. If the cell does not lie on any of the axis boundaries, there are 26 adjacent hot cells. If it lies on one, two, or three axis boundaries, there are 17, 11, or 7 adjacent hot cells, respectively. In case the adjAxisBoundariesCount value does not fall into any of these categories (which is unlikely), the function returns 0.

The calculateGScore function also resides in the utils file. It takes in parameters such as numOfCells, x, y, z, sumOfAdjacentCells, cellNumber, avg, and stdDev. This function calculates the GScore for a given cell based on the provided information. The formula for the GScore involves subtracting the product of the average and the sum of adjacent cells from the cell number. This result is then divided by the product of the standard deviation and the square root of a complex expression involving the number of cells, the sum of adjacent cells, and the number of cells minus one.

I performed the Hot Cell Analysis by first querying for all adjacent hot cells:

```
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" )
```

After which I called the CalculateNumberOfAdjacentCells() function on the above object to calculate the number of adjacent cells and used the avg and stdDev to calculate the gScore for each 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()
```

Since the table was already in descending order and only included 50 locations, I was able to select for the x, y, and z values of these and save the values back into pickupInfo before being returned to the user.

Lessons Learned:
Throughout this project, I gained valuable insights and skills that have contributed to my learning and future development. Here are the key lessons I learned:

- Setting up Apache Spark: I learned how to set up and configure Apache Spark for data processing tasks. This involved understanding the necessary configurations and dependencies to work with Spark efficiently.
- Running SQL queries on Spark: I learned how to leverage Spark's SQL capabilities to run SQL queries on large datasets. This enabled me to extract the required information and perform aggregations and transformations efficiently.
- Geospatial data processing: Working with geospatial data exposed me to various challenges and concepts, such as determining whether a point is located inside a zone, extracting zone boundaries, and interacting with longitude and latitude values. This experience enhanced my understanding of geospatial data processing techniques.
- Scala project structure and development: This project provided me with hands-on experience in structuring and writing a Scala project. It was my first time working with Scala code, and I gained valuable insights into building a simple project, using SBT commands, compiling, cleaning, and packaging a Scala project.
- User-defined Functions (UDFs) and DataFrames: I acquired knowledge on creating and using UDFs in Spark, allowing me to extend its functionality and perform custom operations on data. Additionally, I worked extensively with DataFrames, which provided a structured and efficient way to manipulate and analyze data.
- Local testing and defining test output: I learned how to test my code locally by providing input files and defining the test output directory. This allowed me to validate the correctness of my implementations and identify any issues or bugs.

Future Work:
Future challenges may involve parsing other larger datasets to assess the existence and location of other kinds of hotspots. This could be to identify where in the world the most people turned into the US Open, Olympics, or other sports events, or to study the migration patterns and groupings of animals. Larger datasets may prove harder to query and might challenge the developer to find better ways to store/group the data either geographically, temporally, or with some combination of the two. Other interesting challenges may be to combine the output values with a visualization library like matplotlib to graph the data or find new and meaningful trends from it.