Geospatial data analysis with Apache SparkSQL

Takahiro Morita
Arizona State University
tmorita2@asu.edu

Haemee Park
Arizona State University
haemee.park@asu.edu

Pankaj Taneja
Arizona State University
ptaneja6@asu.edu

1. Introduction

Analysis using geospatial data is used to find the complex relationships between people and location data. In today's daily life, there are a lot of business cases that utilize geospatial data. For example, weather forecast modelling, promotion development for sales, national defence strategies, and the optimum allocation of standby taxi vehicles and so on. Geospatial data is essential for these businesses to increase operational efficiency.

To process massive volume data processing for the purpose, Hadoop and Hadoop MapReduce are introduced. Although these frameworks achieve the purpose, there is still weakness such as slow run time. Apache Spark is the solution to alleviate the weak point of Hadoop, which process data with in-memory data called RDD(Resilient Distributed Datasets).

This report describes two projects which use Apache Spark framework. The first project is to find out the whole of the New York City taxi locations which is within the targeted area. The second project is to find out hot zones using the collected New York City taxi data from January 2009 to June 2015.

ACM SIGSPATIAL Cup 2016 details the second project problem definition, which is to identify a list of the fifty most significant hot spot cells in time and space with the Getix-Ord Gi* statistic. given taxi trip data set for six years with longitude and latitude divided by month of pickup locations, determine the clusters by weighting the values from other cell values.

2. Backend implementation

2.1. Milestone 4

This project includes two scala code files SparkSQLExample.scala and SpatialQuery.scala. The main function in SparkExample.scala file is the start point. The main function calls the paramParser function. Here inside, depending on the number of arguments, queryLoader function runs. Inside queryLoader function runs all of the functions which are defined in the SpatialQuery.scala file. In the SpatialQuery.scala file, there are four functions and each executes range query, range join query, distance query and distance join query.

2.2. Milestone 5

This project includes five scala files. Entrance.scala file is the start point of this application which contains main, paramsParser and queryLoader functions. These three functions run appropriate functions for hot zone analysis or hot cell analysis after parsing input parameters. For hot zone analysis, runHotZoneAnalysis function in the HotzoneAnlaysis.scala file is called, and this function uses functions which are defined in the HotzoneUtils.scala file. Similarly, for hot cell analysis, runHotCellAnalysis function is called which uses functions defined inside the HotcellUtils.scala function.

3. System Requirements

The following OS and software are required to compile and run this program.

- Operating system requirements
  This program should run on both Windows and UNIX-like systems (e.g. Linux, Mac OS)

- Software requirements
  - Java 8:
    Java should be installed on your system PATH, or the JAVA_HOME environment variable pointing to a Java installation.
  - Scala:
    Scala(www.scala-lang.org) whose version is compatible with your Spark should be installed on your system.
  - sbt or any other Scala IDE:
    To compile this source code, SBT(www.scala-sbt.org) or any other Scala IDE is needed.
  - Spark:
    Spark (spark.apache.org) should be installed on your system. Generally latest version is recommended and this code was compiled and tested on the Spark 2.4.5

4. Architecture

Both projects of Milestone 4 and Milestone 5 are Spark applications especially using SparkSQL. Inside both projects, data processing such as select, transformation and merge is implemented with sparkSQL. Apache Spark is an analytics engine for large-scale data processing. Spark provides high performance using a DAG scheduler and in-memory data processing. It also provides usability by allowing users to write their applications in SQL. In both of the projects, we can take these advantages by implementing data processing operations with sparkSQL.

5. Structure of problem statement

To be a better city taxi service provider, we want to find out the hotspots of the city where there are maximum trips taken on an axis of time. To serve the city transportation needs better we will be doing two types of analysis on the past trip data provided by Taxi providers for taxi trips: Hot Zone analysis and Hot Cell analysis.

6. Data analysis

6.1. Milestone 4 user-defined functions

6.1.1. ST_contains

ST_Contains are mainly used for range and range join queries.
The purpose of “range query” is to find all the points within rectangle R with the given query R and a set of points P. Also, the purpose of “range join” is to find all (pint, rectangle) pairs such that the point is within the rectangle.

Input: pointString1:String, pointString2:String, distance: Double
Output: Boolean (true or false)

Given three inputs: pointString1, pointString2 and distance, find all points which are with the specified distance.
Given two inputs called “queryRectangle” and “pointString” as both string type, check whether or not the “queryRectangle” fully contains the given points.
“queryRectangle” contains the string of four numbers separated by “,” such as ”-104.170631,36.174191,-103.846491,36.572659” which represent two points of X-coordinate and Y-coordinate.

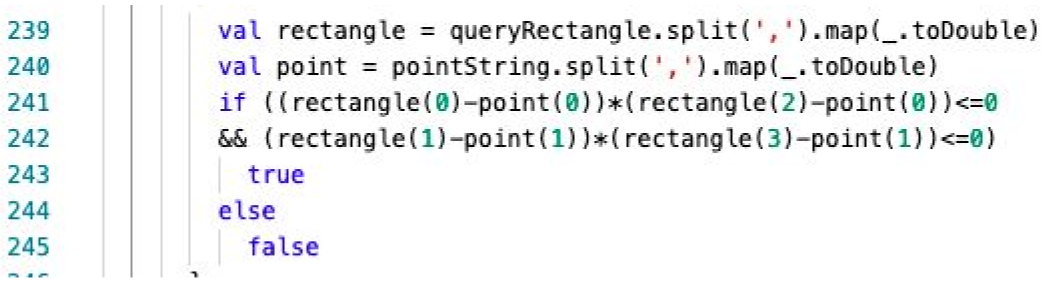
“pointString” contains the string of two numbers separated by a comma “,” such as ”-87.238875,31.41025” which represent one point of X-coordinate and Y-coordinate.

Split the “queryRectangle” string into four separated numbers and “pointString” string two separated numbers.

$$The\ distance\ = \sqrt{(xA - xB)^2 + (yA - yB)^2}$$

If the following two conditions are met, the function returns true, otherwise false.

Let’s call the first rectangle point A and the other point B. With this definition, each X-coordinate is Ax and Bx, also Y-coordinates are Ay and By.
Also, name the point as C, then Cx and Cy respectively.
The condition checks whether the computation result of (Ax-Cx) * (Bx-Cx) is less than or equal to 0 as well as (Ay-Cy) * (By-Cy) is less than or equal to 0.
The screenshot depicts the above explanation.



```
val rectangle = queryRectangle.split(',').map(_.toDouble)
val point = pointString.split(',').map(_.toDouble)
if ((rectangle(0)-point(0))*(rectangle(2)-point(0))<=0
&& (rectangle(1)-point(1))*(rectangle(3)-point(1))<=0)
  true
else
  false
```

Figure 1. ST_contains algorithm

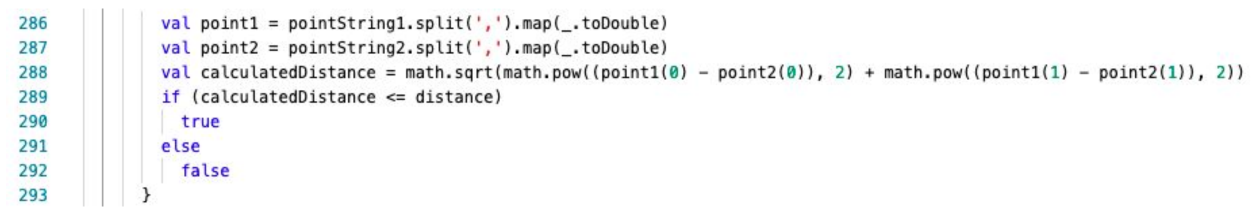
6.1.2. ST_Within

ST_Within are mainly used for distance query and range distance join query.

Input: pointString1:String, pointString2:String, distance:Double
Output: Boolean (true or false)
Convert the given two points as arrays with delimiter “,”. Using the Pythagorean theorem, calculate the distance between two points.
For example, suppose point A (xA, yA) and point B (xB, yB) are given. To find the distance between them, computer the square root of the squared differences between x and y coordinates as follows.

$$The\ distance\ = \sqrt{(xA - xB)^2 + (yA - yB)^2}$$

The screenshot depicts the above explanation.



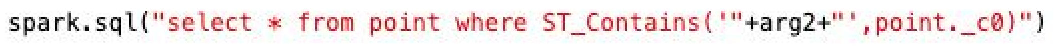
```scala
      val point1 = pointString1.split(',').map(_.toDouble)
      val point2 = pointString2.split(',').map(_.toDouble)
      val calculatedDistance = math.sqrt(math.pow((point1(0) - point2(0)), 2) + math.pow((point1(1) - point2(1)), 2))
      if (calculatedDistance <= distance)
        true
      else
        false
    }
```

Figure 2. ST_Within algorithm

6.1.3. Spatial queries

Those two functions are run with four spatial queries.

1. Range query

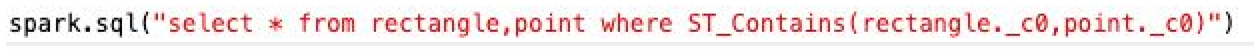


```scala
spark.sql("select * from point where ST_Contains('"+arg2+"',point._c0)")
```

Figure 3. Range Query

This query returns all of the columns from point dataset where the dataset meets the ST_Contains function logic. In the above screenshot, the arg2 indicates the location of the point dataset file.

2. Range join query

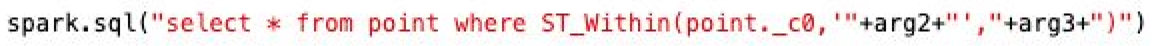


```scala
spark.sql("select * from rectangle,point where ST_Contains(rectangle._c0,point._c0)")
```

Figure 4. Range join query

This query returns all of the columns from both rectangle and point dataset where the dataset meets the ST_Contains function logic.

3. Distance query

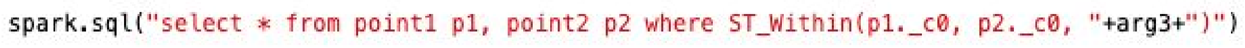


```scala
spark.sql("select * from point where ST_Within(point._c0,'"+arg2+"',"+arg3+")")
```

Figure 5. Distance query

This query returns all of the columns from point dataset which satisfies the ST_Within logic. In the above screenshot, the arg2 and arg3 indicate the location of both point and rectangle dataset files.

4. Distance join query



```scala
spark.sql("select * from point1 p1, point2 p2 where ST_Within(p1._c0, p2._c0, "+arg3+")")
```

Figure 6. Distance join query

This query returns all of the columns return all of the columns from two point-datasets. In the above screenshot, arg3 indicates the distance as the reference. And p1._c0 and p2._c0 are to calculate the distance between two points.

6.2. Milestone 5 Geospatial analysis

6.2.1. Hot Zone analysis:

In hot zone analysis, we are expected to calculate the hotness of a cell. The hotness of a cell is equal to the total number of points located within the rectangle. In the hot zone analysis, we are calculating the hotness of all the rectangles.

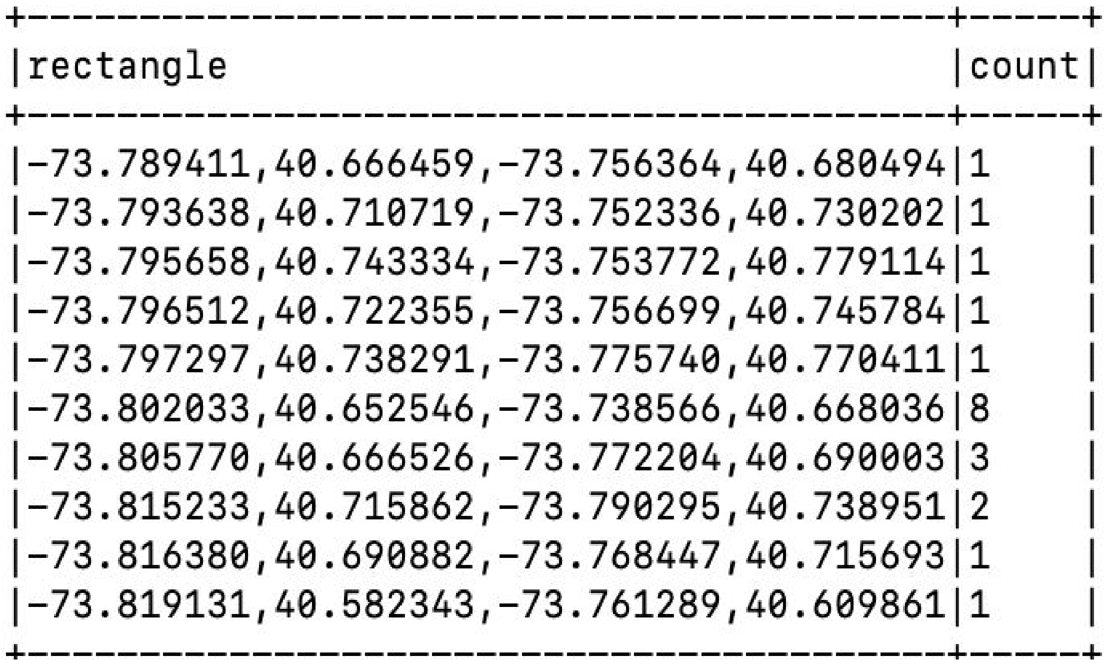
We will be given the set of points/coordinates and set of coordinates of the multiple rectangles. We first retrieve the list of points from the points input path and then retrieve all the corner points of the rectangle from the rectangle input path. Once we have datasets of all the points and rectangles we do the join operation with the help of custom function ST_Contain where we check the point location compared to rectangle corner points. If the location of the points overlaps within the rectangle area then we keep inserting the rows in the output dataset. Once we have a dataset of the rectangles, then we do the group by operation by the rectangle column and sort it by count column.

The neighbourhood for each cell in the space-time cube is established by the neighbours in a grid-based on subdividing latitude and longitude uniformly. This spatial neighbourhood is created for preceding, current, and following time periods(i.e each cell has 26 neighbours). For simplicity of computation, the weight of each neighbour cell is presumed to be equal.

Below piece of code depicts the above explanation:

```
// 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")
```



```
+-------------------------------------------+-----+
|rectangle                                  |count|
+-------------------------------------------+-----+
|-73.789411,40.666459,-73.756364,40.680494|1    |
|-73.793638,40.710719,-73.752336,40.730202|1    |
|-73.795658,40.743334,-73.753772,40.779114|1    |
|-73.796512,40.722355,-73.756699,40.745784|1    |
|-73.797297,40.738291,-73.775740,40.770411|1    |
|-73.802033,40.652546,-73.738566,40.668036|8    |
|-73.805770,40.666526,-73.772204,40.690003|3    |
|-73.815233,40.715862,-73.790295,40.738951|2    |
|-73.816380,40.690882,-73.768447,40.715693|1    |
|-73.819131,40.582343,-73.761289,40.609861|1    |
+-------------------------------------------+-----+
```

Figure 7. Hot Zone analysis

6.2.2. Hot Cell analysis:

In hot cell analysis we are expected to identify statistically significant spatial hot spots, that means particular coordinate/point at a given time. We identify coordinates/points with the following attributes x, y and z, where x and y are latitude and longitude and z is the day of the month for simplicity: time is diversified just days of the month and we assume that all months have 31 days. Time and space should be aggregated into cube cells like in the picture below.
We first parse the x, y and z from the given data and then we filter the point dataset based on the location, we define the max and min latitude and longitude of the location and min and max day of the month respectively. Below is the calculation of GScore for each and every point in the cell.

Counting neighbors:

```
val neighbour = spark.sql("select NeighbourCount("+ minX +
"," + minY + "," + minZ+
 "," + maxX + "," + maxY + "," + maxZ + "," + "a1.x, a1.y,
a1.z) as countN," +
 " count(*) as countall," +
 " a1.x as x," +
 " a1.y as y," +
 " a1.z as z," +
 " sum(a2.pointval) as sumTotal " +
 "from DF1 as a1, DF1 as a2 " +
 "where (a2.x = a1.x+1 or a2.x = a1.x or a2.x = a1.x-1) " +
 "and   (a2.y = a1.y+1 or a2.y = a1.y or a2.y = a1.y-1) " +
 "and   (a2.z = a1.z+1 or a2.z = a1.z or a2.z = a1.z-1) " +
 "group by a1.z, a1.y, a1.x " +
 "order by a1.z, a1.y, a1.x").persist()
neighour.createOrReplaceTempView("DF2");


// gStar = A/B
// A = (numCells - numCells * (Xavg.toDouble) )
// B = S * math.sqrt(((numCells * spatialWeight - numCells *
numCells).toDouble / (cellN - 1)))
// S = sqrt(((sumN**2) / n) - (avgX)**2)
```

get_tisOrdStatistic

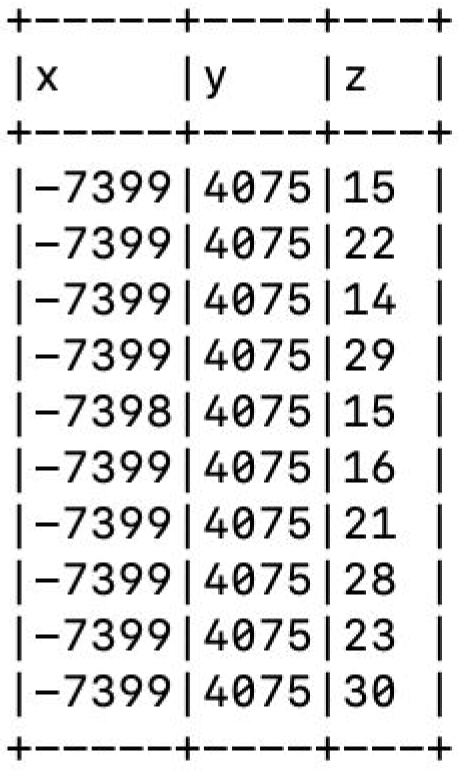
```
def get_tisOrdStatistic(x: Int, y: Int, z: Int, mean:Double,
s: Double, countN: Int, sumn: Int, numcells: Int): Double =
{
 val numerator = (sumn.toDouble - (mean * countN.toDouble))
 val denominator = s * math.sqrt(
   (((numcells.toDouble * countN.toDouble) - (countN.toDouble
* countN.toDouble))/(numcells.toDouble -
1.0).toDouble).toDouble).toDouble
 return (numerator / denominator).toDouble
}
```

Calculating GScore using the above method

```
// gstar
spark.udf.register("GScore", (x: Int, y: Int, z: Int, avgX:
Double, s: Double, countN: Int, sumN: Int, numCells: Int) =>
((
 HotcellUtils.get_tisOrdStatistic(x, y, z, avgX, s, countN,
sumN, numCells))))
```

Output the *coordinates of the top 50 hottest cells sorted by their G score in a descending order. Note, DO NOT OUTPUT G score.*

```
val neighbourG = spark.sql("select GScore(x, y, z, "+ avgX
+","+ S +", countN, sumTotal,"+ numCells + ") as gtstat,x, y,
z from DF2 order by gtstat desc, x desc, y asc, z desc");
neighbourG.createOrReplaceTempView("DF3")
neighbourG.show(20)
```



```
+-----+----+---+
|x    |y   |z  |
+-----+----+---+
|-7399|4075|15 |
|-7399|4075|22 |
|-7399|4075|14 |
|-7399|4075|29 |
|-7398|4075|15 |
|-7399|4075|16 |
|-7399|4075|21 |
|-7399|4075|28 |
|-7399|4075|23 |
|-7399|4075|30 |
+-----+----+---+
```

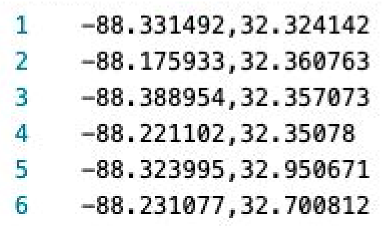
Figure 8. Hot Cell analysis output

7. Input and output

7.1. Milestone 4

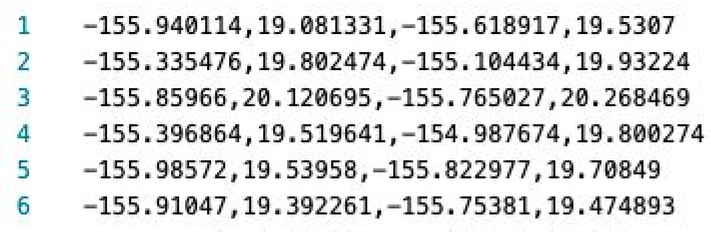
7.1.1. Input

Two input files are given; one contains point data Arealm10000.csv. Another one contains rectangle data Zcta10000.csv. Those files are given as the example dataset for test purposes. The following screenshot is a part of the sample data of those files.



```
-88.331492,32.324142
-88.175933,32.360763
-88.388954,32.357073
-88.221102,32.35078
-88.323995,32.950671
-88.231077,32.700812
```

Figure 9. Part of the Arealm10000.csv input rows



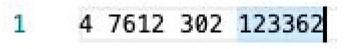
```
-155.940114,19.081331,-155.618917,19.5307
-155.335476,19.802474,-155.104434,19.93224
-155.85966,20.120695,-155.765027,20.268469
-155.396864,19.519641,-154.987674,19.800274
-155.98572,19.53958,-155.822977,19.70849
-155.91047,19.392261,-155.75381,19.474893
```

Figure 10. Part of the Zcta10000.csv input rows

However, eventually, the project file is run with NYC taxi trip data when grading.

7.1.2. Output

The output should contain all of the four queries result: range query, ”rangejoin”, distance query, distance join query. With the sample given input dataset, the result should be the following screenshot.



```
4 7612 302 123362
```

Figure 11. Output values of Milestone 4

7.2. Milestone 5

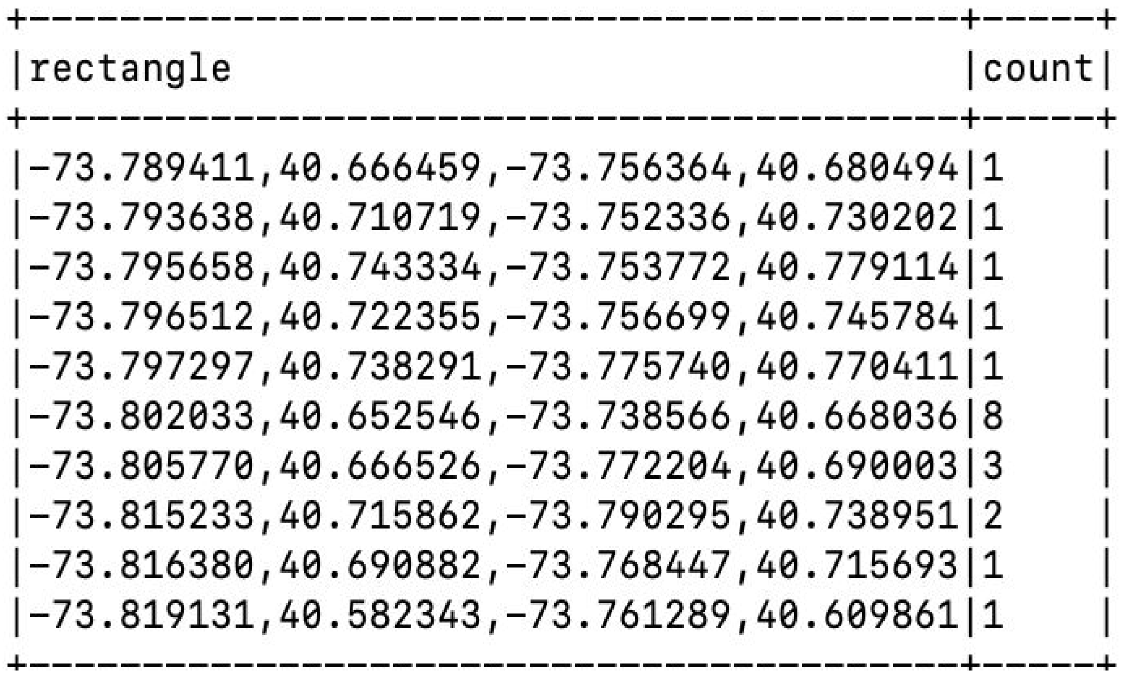
7.2.1. Input

Hot zone analysis: Input parameters contain the path to points file which contains the trip details, pick up location, pick up time, price etc and the path to zone points which contains the coordinates of zones.

Hot cell analysis: Input parameters contain the path to points file which contains the trip details, pick up location, pick up time, price etc

7.2.2. Output

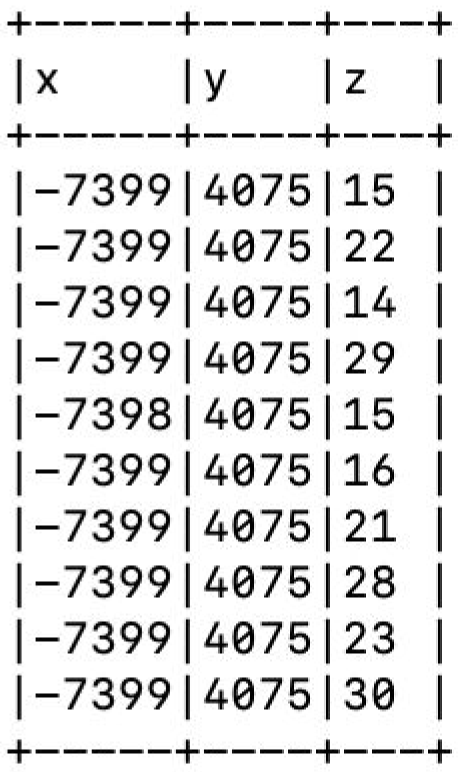
Hot Zone analysis: output the hotness of each rectangle



```
+---------------------------------------------+-----+
|rectangle                                    |count|
+---------------------------------------------+-----+
|-73.789411,40.666459,-73.756364,40.680494|1    |
|-73.793638,40.710719,-73.752336,40.730202|1    |
|-73.795658,40.743334,-73.753772,40.779114|1    |
|-73.796512,40.722355,-73.756699,40.745784|1    |
|-73.797297,40.738291,-73.775740,40.770411|1    |
|-73.802033,40.652546,-73.738566,40.668036|8    |
|-73.805770,40.666526,-73.772204,40.690003|3    |
|-73.815233,40.715862,-73.790295,40.738951|2    |
|-73.816380,40.690882,-73.768447,40.715693|1    |
|-73.819131,40.582343,-73.761289,40.609861|1    |
+---------------------------------------------+-----+
```

Figure 12. Part of the Hot Zone analysis output

Hot Cell analysis: outputs the hotspot analysis



```
+-----+----+---+
|x    |y   |z  |
+-----+----+---+
|-7399|4075|15 |
|-7399|4075|22 |
|-7399|4075|14 |
|-7399|4075|29 |
|-7398|4075|15 |
|-7399|4075|16 |
|-7399|4075|21 |
|-7399|4075|28 |
|-7399|4075|23 |
|-7399|4075|30 |
+-----+----+---+
```

Figure 13. Part of Hot Cell analysis output

8. Conclusion

We successfully completed all the milestones of the project as a team. Coming together from different backgrounds we helped each other and collaborated to achieve the end result of the project successfully.
During the initial phase, the team communicated over the Slack channel and zoom conference and discussed the strategy and roadmap of the project. We defined the task and distributed the responsibilities among all team members.
Team members researched and worked on their respective tasks individually and kept on discussing the progress and overall integration of the project. We were all supportive and helped each other to perform as a team. As a team we also made sure that not only we should complete the project but to make sure that everyone acquired the same level of knowledge and understanding of the project.
Our team is very thankful for the opportunities provided by team CSE511. It really helped us grow individually as well as a team.

9. FAQs

9.1. Q: How to solve Error: java.lang.NoClassDefFoundError: scala/StringContext, this is because the SBT version is old.

A: To fix this issue, change the sbt.version=0.12.2 or than above in project/build.properties inside the project.

9.2. Q: How to solve Differences between provided and compile. Provided flag points to the scoped key in the classpath.
A: Use this command to run the project including the debug purpose. Compile flag is better to use during the first compiling time. Other than that, it would be better to use provided flag otherwise the programmer might face the error the NoClassDefFoundError: SparkSession.

9.3. Q: How to solve “java.lang.NoClassDefFoundError: org/apache/log4j/Logger” does not exist.
A: Occurs error message when using the JDK version is less than 1.8.

9.4. Q: “org.apache.spark.sql.AnalysisException: Path does not exist:”
A: Make sure paths to input files are correct in the sbt run command.

9.5. Q: “java.lang.RuntimeException: No main class detected”
A: Run the sbt command from the project root folder.

10. References

- https://aws.amazon.com/big-data/what-is-spark/
- http://sigspatial2016.sigspatial.org/giscup2016/problem