System Documentation Report

CSE 511: Course Project

Group 5

Baihan Chen
Cheng-Yu Chung
Sambhav Jain
Narmada Ravali Namburi
Mariya Varghese
Sanjay Vishal Velmurugan

Introduction

Spatial data analysis is one noteworthy application of big data processing due to the nature of data from this domain. Spatial data usually consists of tremendous amount of data points which are closely bound to a physical space in the real world. For example, a taxi service may produce tens of thousands of data about pickup locations every day. To fully utilize such a dataset, the data processor must be able to handle and analyze a large amount of data at one time and maintain both the efficiency and throughput. Apache Spark is one popular framework which can suffice in this case. In this phase of project, our goal is to reuse the functions implemented in the previous phase of project to design two programs for *Hot Cell Analysis* and *Hot Zone Analysis*. These two analyses are used to understand how popular a certain location is according to its *hotness*, which is defined by a function of the number of records associated to that location. In this report, we provide the detail of our project requirement, our implementation, the interface and some testing results.

Project Requirement

Phase 1: User Defined Functions for SparkSQL

UDF: ST_Contain

ST_Contains function takes in a latitude-longitude coordinate as input and returns a boolean value indicating if the rectangle contains the point or not. The scala implementation of this function contains the logic whether a given point lies within a rectangle or not. The input for this function is nothing but one point-string and one rectangle-string. The input parameter rectangle string is parsed to get four coordinates of the rectangle and point string is parsed to get it's coordinates into a format which is convenient for comparisons. The output for this function is a boolean value to convey whether the point is contained within the rectangle or not.

UDF: ST_Within

ST_WITHIN takes two points on a two-dimensional plane and a distance value as method arguments. It then returns whether the two points are within Euclidean distance to one another as a boolean value, 'true' for if the points are within Euclidean distance and 'false' for if the two points are not within Euclidean distance from each other.

Relevant Query Operations

There are four types of query operations required by the spec:

- **RangeQuery**: Given a rectangle R and a set of points P, find all the points within R.
- **RangeJoinQuery**: Given a set of Rectangles R and a set of Points S, find all (Point, Rectangle) pairs such that the point is within the rectangle.
- **DistanceQuery**: Given a set of points P and distance D in km, find all points that lie within a distance D from P.
- **DistanceJoinQuery**: Given a set of Points S1, a set of Points S2, and a distance D in kilometer, find all (s1, s2) pairs such that s1 is within a distance D from s2 (i.e., s1 belongs to S1 and s2 belongs to S2).

In our implementation, RangeQuery and RangeJoinQuery perform Spark SQL queries with the same ST_Contains method; DistanceQuery and DistanceJoinQuery perform Spark SQL queries with the same ST_Within method. The details of SQL in use is shown as follows:

- RangeQuery: SELECT * FROM point WHERE ST_Contains('$rect', point._c0)
- RangeJoinQuery: SELECT * FROM rectangle, point WHERE ST_Contains(rectangle._c0, point._c0)
- DistanceQuery: SELECT * FROM point WHERE ST_Within(point._c0, '$location', $distance)
- DistanceJoinQuery: SELECT * FROM point1 p1, point2 p2 WHERE ST_Within(p1._c0, p2._c0, $distance)

In these SQL statements, tokens prefixed by "$" (e.g., "$location", "$distance") are string interpolations in Scala which will be substituted by values in corresponding variables in runtime. In addition, by default, a temporary view created by Spark DataFrame API uses the column names prefixed by an underscore ("_") followed by a number. "point._c0" means the first column in the table "point" (or more specifically, a temporary view, according to our implementation).

Phase 2: The Hotspot Analysis

Hot Zone Analysis (Based on ST_Contain)

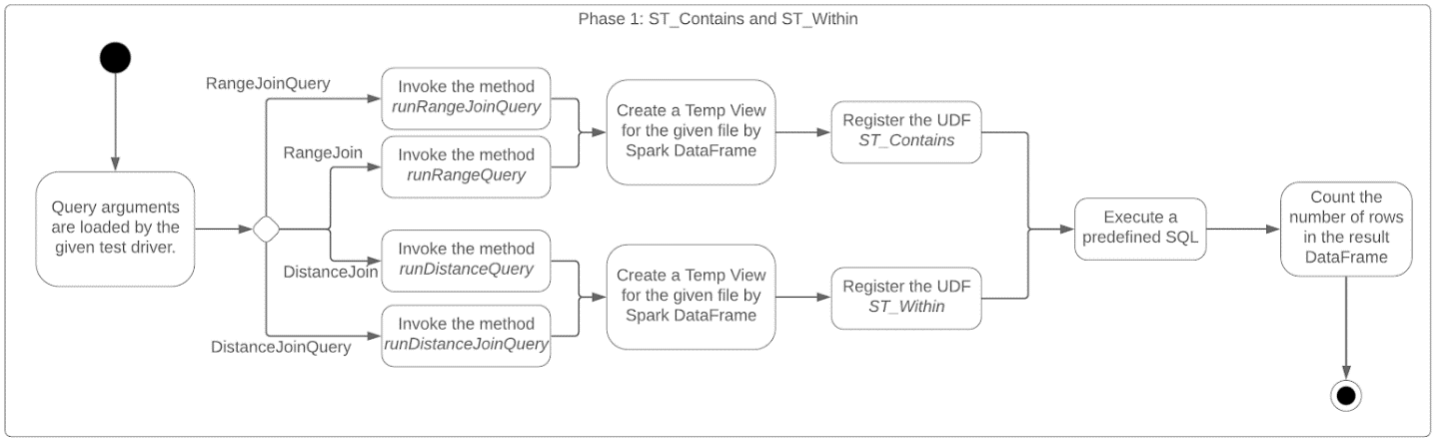
First hotspot analysis task is called HotZoneAnalysis. Hot Zone analysis is performed by loading a csv of points and a csv of rectangles into a Spark data frame. The data frames are then joined where the points reside in the rectangles, and an aggregate count operation is performed to collapse the data frame into a list of unique entries with the count of occurrence. This final list is then sorted by the rectangle attribute to determine the 'hottest' or 'most profitable' rectangle (zone).

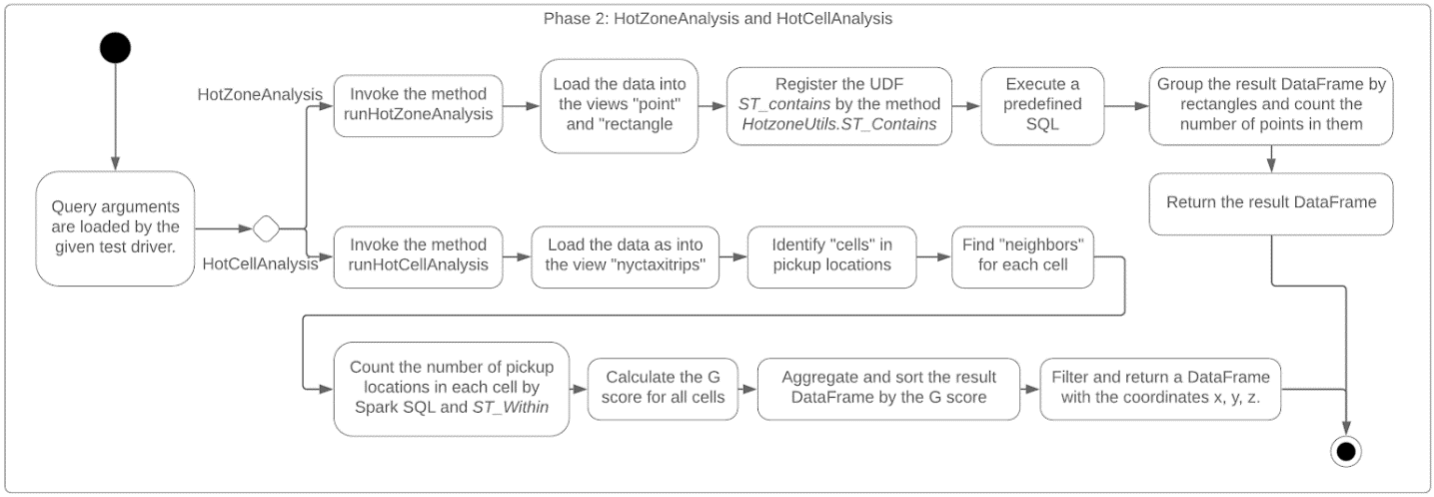
Hot Cell Analysis (Based on ST_Within)

Second hotspot analysis task is called HotCell Analysis. Here we identify statistically significant spatial hotspots. This is done by using the Getis-Ord statistic of NYC Taxi Trip data sets. The input point data consists of a monthly NYC taxi trip dataset provided in a csv file under resources which is parsed to get the coordinates x, y (latitude, longitude) and z (day of the month) for each cell. The G-score for each x-y coordinate of pick up point and time-coordinate z (cell coordinates) is calculated and the final result contains the coordinates in the decreasing order of their G-score.

Architecture Design

In this section, we illustrate the overall architecture of Phase 1 and Phase 2 in order. The illustration is based on the relationship of functionality but not an actual code structure, which is described in detail in the next section.





Task Distribution

Phase 1:

P1-1	Cheng-Yu	Project Setup for P1
P1-2-1	Sambhav	Implement ST_Within
P1-2-2	Baihan	Test and verify the results of ST_Within in the IDE
P1-3-1	Narmada	Implement ST_Contain
P1-3-2	Mariya	Test and verify the results of ST_Contain in the IDE
P1-4	Sanjay	Test P1 on Spark
P1-5	Cheng-Yu	Integration of P1

Phase 2:

P2-1	Cheng-Yu	Project Setup for P2
P2-2-1	Baihan	Implement Hot Zone Analysis
P2-2-2	Sambhav	Test and verify the results of Hot Zone Analysis in the IDE
P2-3-1	Mariya & Sanjay	Implement Hot Cell Analysis
P2-3-2	Narmada	Test and verify the results of Hot Cell Analysis in the IDE
P2-4	Sanjay	Test P2 on Spark
P2-5	Cheng-Yu	Integration of P2

Source Code Document

Phase 1 Interfaces and Usage

SpatialQuery.ST_Contains(queryRectangle: String, pointString: String)
: Boolean

Given a rectangle and a point, this method determines whether the point is in the rect. Note: on-boundary points are considered true.

The parameter *queryRectangle* is a four-item tuple containing two point coordinates, e.g., "-155.940114,19.081331,-155.618917,19.5307". The arrangement of these two points is unknown. In our implementation, there is a validation process which reorders the two given points based on their arrangement.

The parameter *pointString* is a two-item tuple containing the x and y coordinates of a point, e.g., "-88.331492,32.324142". This point string is parsed by a custom helper function *SpatialQuery.parseCoordinateString* which yields items in a String separated by commas.

SpatialQuery.ST_Within(pointString1: String, pointString2: String, distance: Double)
: Boolean

Given two points and a distance, this method determines whether the two points are within the given distance by assuming all coordinates are on a planar space on which we can calculate the Euclidean distance.

The definition of the parameters *pointString1* and *pointString2* is the same as *pointString* in SpatialQuery.ST_Contains.

Phase 2 Interfaces and Usage

HotzoneAnalysis.runHotZoneAnalysis(spark: SparkSession, pointPath: String, rectanglePath: String)
: DataFrame

This method performs a range join operation on the given rectangles and points and find out the number of points within each rectangle. The "hotness" of a rectangle is defined as the number of points it includes.

This method is implemented based on SpatialQuery.ST_Contains in Phase 1. Our implementation can be seen as a variation of Range Query in Phase 1.

HotcellAnalysis.runHotCellAnalysis(spark: SparkSession, pointPath: String)
: DataFrame

This method analyzes hot cells of the given point data and returns a DataFrame ordered by the hotness of cells. This hotness is defined as the Getis-Ord statistic (G Scores) of cells.

The calculation of G Scores is done by Spark DataFrame API.

This method is implemented based on SpatialQuery.ST_Within in Phase 1.

Verification and Testing

Phase 1: Testing in the Intellij IDE

Testing the Phase 1 of the project on Intellij IDE requires the following steps:

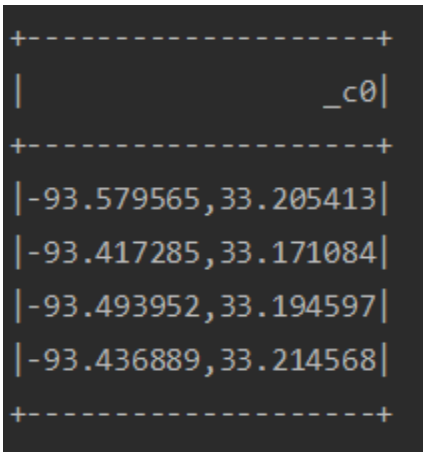
- Import SBT tool kit into IDE
- For running spark on local host, modify .config("spark.some.config.option", "some-value") to .config("spark.some.config.option", "some-value").master("local[*]"). According to core numbers of CPU, the star sign can be changed to numbers.
- There are four functions to test, Range Query, Range Join Query, Distance Query and Distance Join Query.
- Example command lines and corresponding outputs were given in the test case directory. These examples used the CSV files located in the src/resources directory as input datasets for the four functions.
- Finally, setup the program arguments for testing the 4 queries through the edit configuration option.

Phase 1: Testing by Spark

Testing the Phase 1 of the project on Spark requires the following steps:

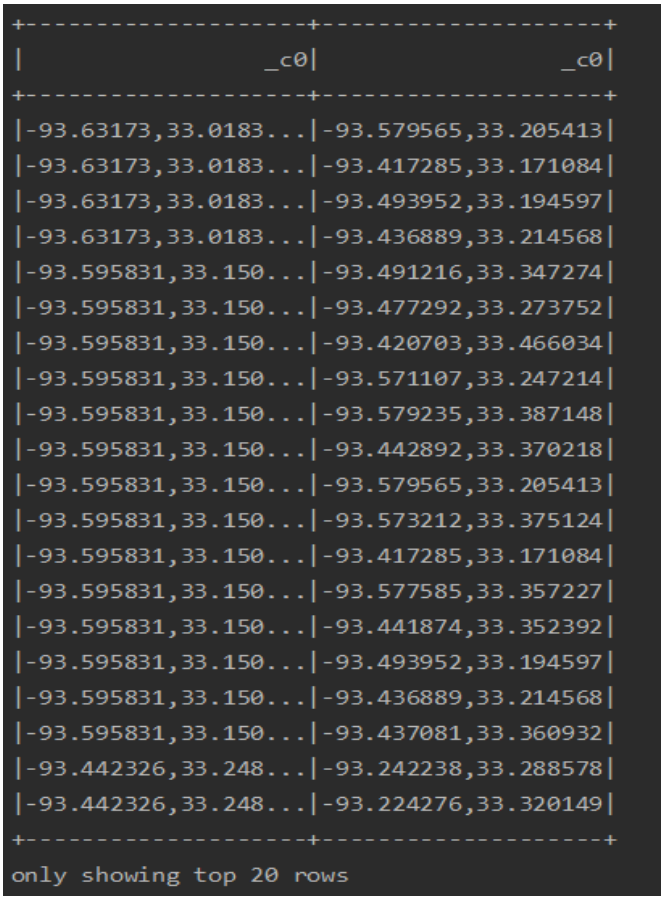
- Install sbt, which is an open source build tool for Scala projects. It can be downloaded from the following link: https://www.scala-sbt.org/download.html
- Run sbt assembly command from the project directory. This will copy the class files from your source code, the class files from your dependencies, and the class files from the Scala library into one single JAR file that can be executed with the scala interpreter.
- The executable jar file would be created in the project folder under target\scala-2.11\ directory.
- Download apache spark from the following link: https://spark.apache.org/downloads.html
- Use the spark-submit command to submit the jar file which we created earlier to the spark cluster.
- The command will look like this when executing for all the 4 queries: ~\spark-2.4.5-bin-hadoop2.7\bin\spark-submit target\scala-2.11\CSE512-Project-Phase2-Template-assembly-0.1.0.jar result/output rangequery src/resources/arealm10000.csv -93.63173,33.0183,-93.359203,33.219456 rangejoinquery src/resources/arealm10000.csv src/resources/zcta10000.csv distancequery src/resources/arealm10000.csv -88.331492,32.324142 1 distancejoinquery src/resources/arealm10000.csv src/resources/arealm10000.csv 0.1
- The first line in the command refers to the spark-submit executable which is present in the apache spark folder. The second line refers to the jar file and the directory in which the output needs to be saved. The third line is the beginning of the input parameters (it is the example input which was provided with the project folder).
- The output data is stored in the result folder which was mentioned in the command.

Phase 1: Query Results



_c0
-93.579565,33.205413
-93.417285,33.171084
-93.493952,33.194597
-93.436889,33.214568

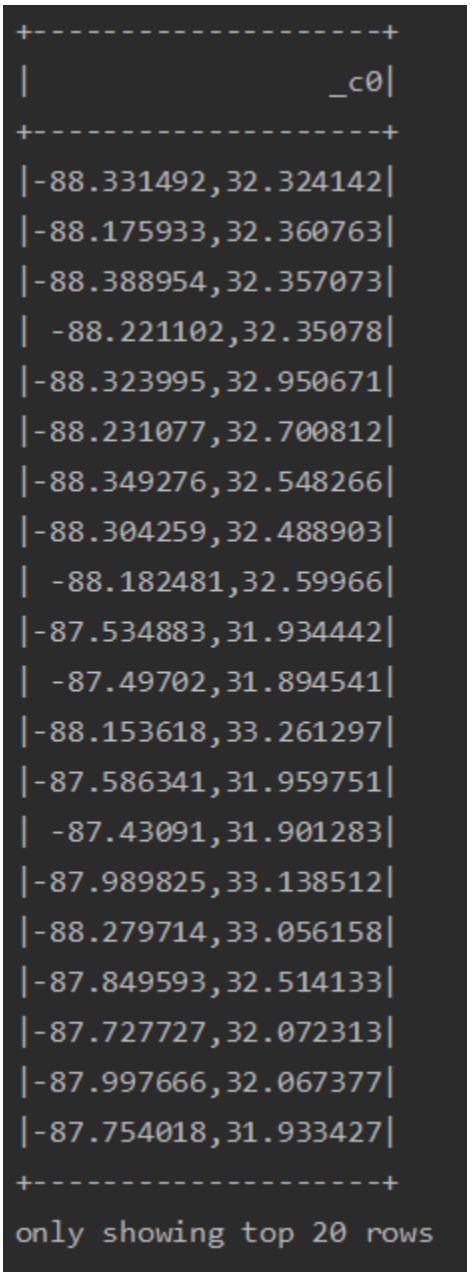
1) <u>Range Query</u>



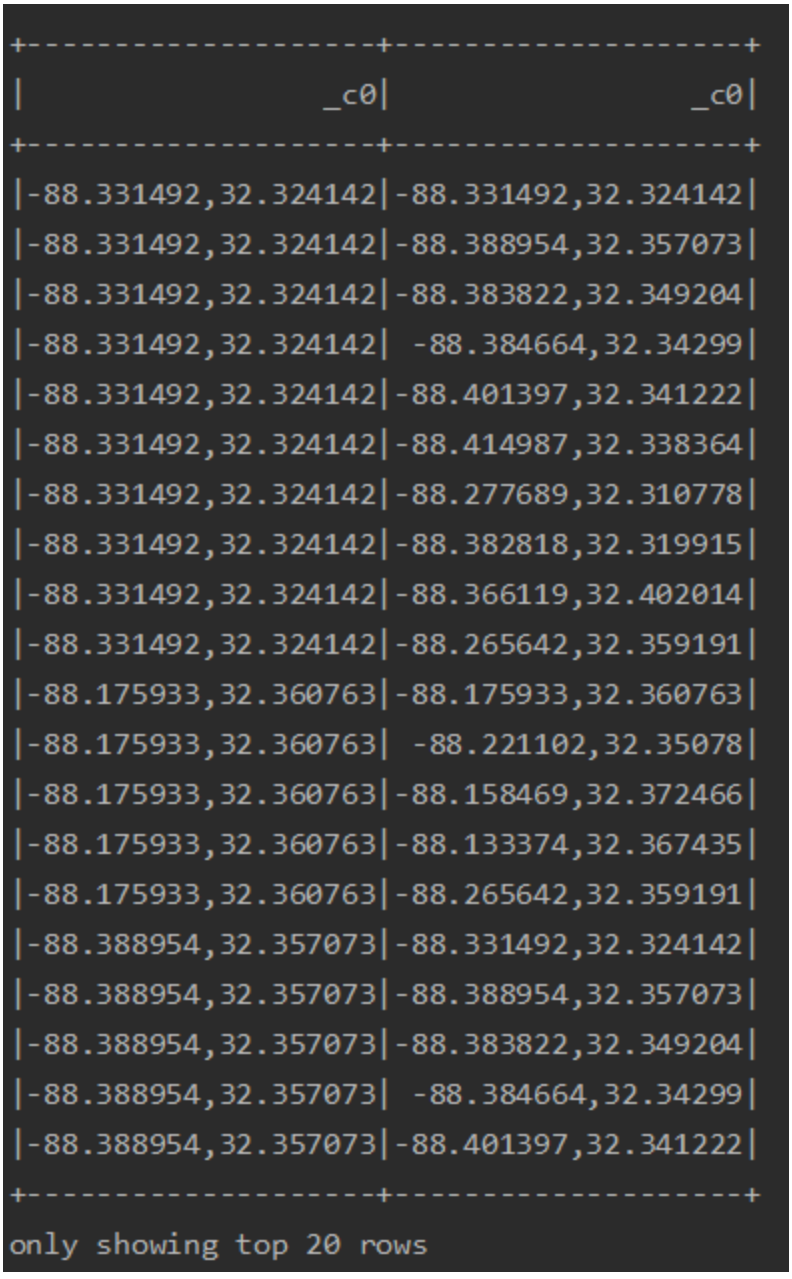
_c0	_c0
-93.63173,33.0183...	-93.579565,33.205413
-93.63173,33.0183...	-93.417285,33.171084
-93.63173,33.0183...	-93.493952,33.194597
-93.63173,33.0183...	-93.436889,33.214568
-93.595831,33.150...	-93.491216,33.347274
-93.595831,33.150...	-93.477292,33.273752
-93.595831,33.150...	-93.420703,33.466034
-93.595831,33.150...	-93.571107,33.247214
-93.595831,33.150...	-93.579235,33.387148
-93.595831,33.150...	-93.442892,33.370218
-93.595831,33.150...	-93.579565,33.205413
-93.595831,33.150...	-93.573212,33.375124
-93.595831,33.150...	-93.417285,33.171084
-93.595831,33.150...	-93.577585,33.357227
-93.595831,33.150...	-93.441874,33.352392
-93.595831,33.150...	-93.493952,33.194597
-93.595831,33.150...	-93.436889,33.214568
-93.595831,33.150...	-93.437081,33.360932
-93.442326,33.248...	-93.242238,33.288578
-93.442326,33.248...	-93.224276,33.320149

only showing top 20 rows

2) <u>Range Join Query</u>



```
+--------------------+
|                 _c0|
+--------------------+
|-88.331492,32.324142|
|-88.175933,32.360763|
|-88.388954,32.357073|
| -88.221102,32.35078|
|-88.323995,32.950671|
|-88.231077,32.700812|
|-88.349276,32.548266|
|-88.304259,32.488903|
| -88.182481,32.59966|
|-87.534883,31.934442|
| -87.49702,31.894541|
|-88.153618,33.261297|
|-87.586341,31.959751|
| -87.43091,31.901283|
|-87.989825,33.138512|
|-88.279714,33.056158|
|-87.849593,32.514133|
|-87.727727,32.072313|
|-87.997666,32.067377|
|-87.754018,31.933427|
+--------------------+
only showing top 20 rows
```



```
+--------------------+--------------------+
|                 _c0|                 _c0|
+--------------------+--------------------+
|-88.331492,32.324142|-88.331492,32.324142|
|-88.331492,32.324142|-88.388954,32.357073|
|-88.331492,32.324142|-88.383822,32.349204|
|-88.331492,32.324142| -88.384664,32.34299|
|-88.331492,32.324142|-88.401397,32.341222|
|-88.331492,32.324142|-88.414987,32.338364|
|-88.331492,32.324142|-88.277689,32.310778|
|-88.331492,32.324142|-88.382818,32.319915|
|-88.331492,32.324142|-88.366119,32.402014|
|-88.331492,32.324142|-88.265642,32.359191|
|-88.175933,32.360763|-88.175933,32.360763|
|-88.175933,32.360763| -88.221102,32.35078|
|-88.175933,32.360763|-88.158469,32.372466|
|-88.175933,32.360763|-88.133374,32.367435|
|-88.175933,32.360763|-88.265642,32.359191|
|-88.388954,32.357073|-88.331492,32.324142|
|-88.388954,32.357073|-88.388954,32.357073|
|-88.388954,32.357073|-88.383822,32.349204|
|-88.388954,32.357073| -88.384664,32.34299|
|-88.388954,32.357073|-88.401397,32.341222|
+--------------------+--------------------+
only showing top 20 rows
```

3) <u>Distance Query</u>

4) <u>Distance Join Query</u>

Phase 2: Testing in the Intellij IDE

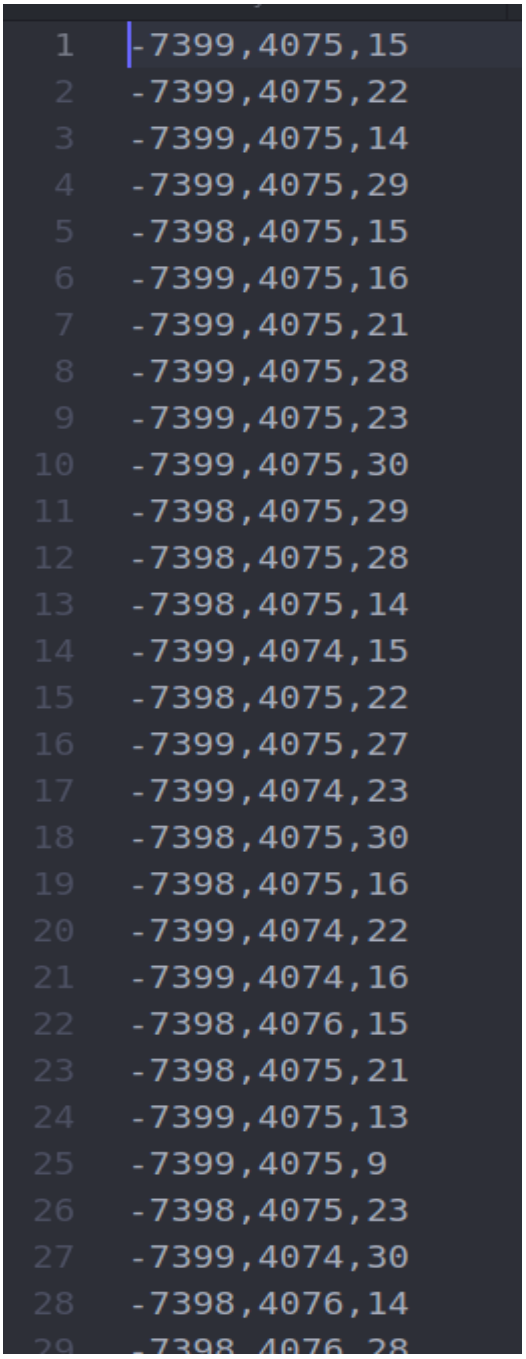
- Import SBT tool kit into IDE
- For running spark on local host, modify .config("spark.some.config.option", "some-value") to .config("spark.some.config.option", "some-value").master("local[*]").
- There are two functions to test, Hot Zone Analysis and Hot Cell Analysis.
- Example command lines and corresponding outputs were given in the test case directory. These examples used the CSV files located in the src/resources directory as input datasets for the four functions.
- Finally, setup the program arguments for testing the 2 queries through the edit configuration option.
- For hot cell analysis, we used the sample monthly NYC taxi dataset uploaded in the following link. Link: https://drive.google.com/open?id=1bN-U4nknvN5p7jiVHO-wduM7oXR5CBji

Phase 2: Testing by Spark

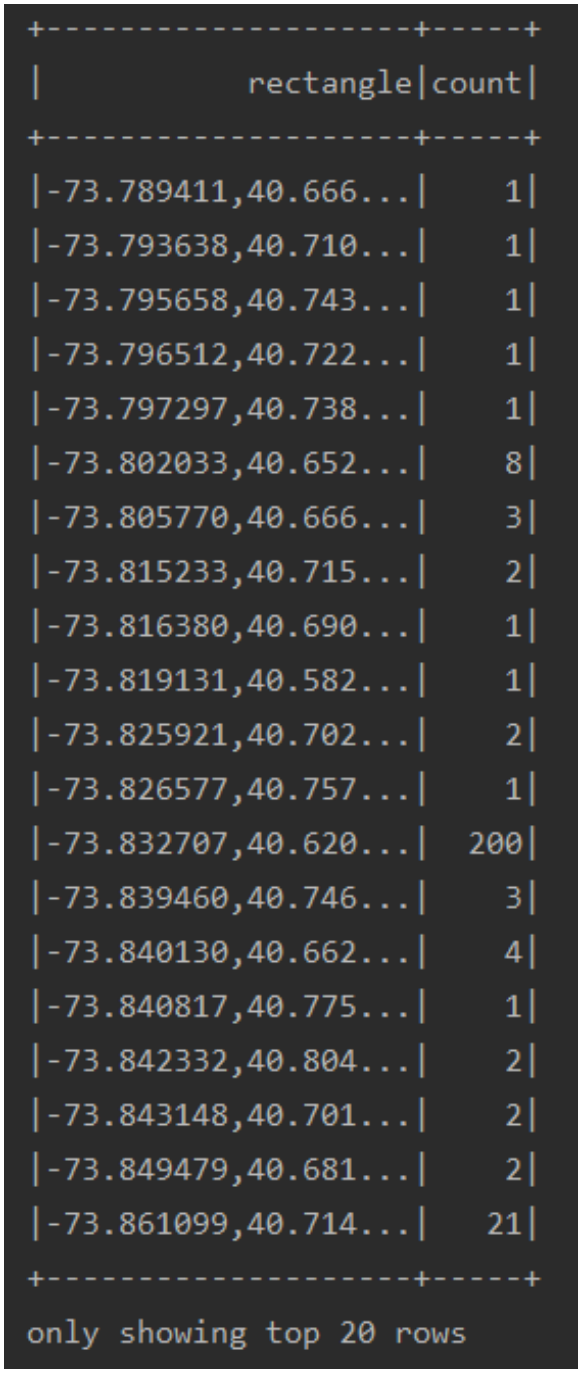
Testing the Phase 2 of the project on Spark requires the following steps:

- Install sbt, which is an open source build tool for Scala projects. It can be downloaded from the following link: https://www.scala-sbt.org/download.html
- Run sbt assembly command from the project directory. This will copy the class files from your source code, the class files from your dependencies, and the class files from the Scala library into one single JAR file that can be executed with the scala interpreter.
- The executable jar file would be created in the project folder under target\scala-2.11\ directory.
- Download apache spark from the following link: https://spark.apache.org/downloads.html
- Use the spark-submit command to submit the jar file which we created earlier to the spark cluster.
- The command will look like this when executing for both the queries: ~\spark-2.4.5-bin-hadoop2.7\bin\spark-submit target\scala-2.11\CSE512-Hotspot-Analysis-Template-assembly-0.1.0.jar test/output hotzoneanalysis src/resources/point-hotzone.csv src/resources/zone-hotzone.csv hotcellanalysis src/resources/yellow_trip_sample_100000.csv
- The first line in the command refers to the spark-submit executable which is present in the apache spark folder. The second line refers to the jar file and the directory in which the output needs to be saved. The third line is the beginning of the input parameters (it is the example input which was provided with the project folder).
- The output data is stored in the result folder named ‘test’ which was mentioned in the command.

Phase 2 Query Results:



```
-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
-7398,4075,29
-7398,4075,28
-7398,4075,14
-7399,4074,15
-7398,4075,22
-7399,4075,27
-7399,4074,23
-7398,4075,30
-7398,4075,16
-7399,4074,22
-7399,4074,16
-7398,4076,15
-7398,4075,21
-7399,4075,13
-7399,4075,9
-7398,4075,23
-7399,4074,30
-7398,4076,14
-7398,4076,28
```



rectangle	count
-73.789411,40.666...	1
-73.793638,40.710...	1
-73.795658,40.743...	1
-73.796512,40.722...	1
-73.797297,40.738...	1
-73.802033,40.652...	8
-73.805770,40.666...	3
-73.815233,40.715...	2
-73.816380,40.690...	1
-73.819131,40.582...	1
-73.825921,40.702...	2
-73.826577,40.757...	1
-73.832707,40.620...	200
-73.839460,40.746...	3
-73.840130,40.662...	4
-73.840817,40.775...	1
-73.842332,40.804...	2
-73.843148,40.701...	2
-73.849479,40.681...	2
-73.861099,40.714...	21

only showing top 20 rows

1) <u>Hotcell Analysis</u>

2) <u>Hotzone Analysis</u>

Note: While the method followed in the spark program for calculating the Getis-Ord Statistic for Hot Cell Analysis is exactly the same as depicted in the Project Instructions, the difference in the expected output is due to mathematical approximations.

Known Issues and Limitations in Our Implementation

Our implementation successfully resolved the requirement of Hot Zone Analysis but not Hot Cell Analysis. We acknowledged that there were some bugs and issues we could not solve due to the limited project time. In this section we describe some issues and limitations in our implementation.

First, our hot cell analysis was not able to produce the exactly the same output as the sample answer. We believe it was due to the equation of the G score we implemented was not correct. Although we tried our best to interpret and understand how to plug in all elements of the equation, it is likely that there were some elements were wrong or not calculated correctly.

The second issue/limitation of our implementation is about the time to run the code. Now our implementation is able to finish calculation in a reasonable amount of time. However, when we were trying different implementations, sometimes the program ran more than an hour and could not finish in several hours, probably due to both the hardware and the logic of our code. This issue made us not able to try a lot of different implementations/testing to figure out what was the bug in our program.