CSE511 portfolio

Wenzhe Zheng
Arizona State University
1215 E Vista Del Cerro Dr
APT 1100S , Tempe 85281
602-503-7089
wzheng41@asu.edu

2nd Author
2nd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
2nd E-mail

3rd Author
3rd author's affiliation
1st line of address
2nd line of address
Telephone number, incl. country code
3rd E-mail

ABSTRACT

In this society, big data analysis is getting more and more love and attention from people, and more and more people are engaged in big data analysis, and spatial data analysis is also a very important direction for big data.

1. INTRODUCTION

. Due to the nature of the data from this field, spatial data analysis is a noteworthy application of big data processing. Spatial data usually contains many data points that are tightly bound to the physical space in the real world. For example, a taxi service may generate tens of thousands of data about pickup locations every day. In order to make full use of such data sets, data processors must be able to process and analyze large amounts of data at once and maintain efficiency and throughput. Apache Spark is a popular framework that can meet the requirements in this case. In this phase of the project, our goal is to reuse the functions implemented in the previous phase of the project to design two programs for hot cell analysis and hot zone analysis. These two analyses are used to understand the popularity of a location based on its popularity, which is defined as a function of the number of records associated with the location. In this report, we provide detailed information on project requirements, implementation, interfaces and some test results.

2. Description of solution

Phase 1: User Defined Functions for SparkSQL

UDF: ST_Contain

The [1]ST_Contains function takes latitude-longitude coordinates as input and returns a Boolean value indicating whether the rectangle contains the point. The scala implementation of this function contains logic: whether a given point lies within a rectangle. The input of this function is only a dot string and a rectangular string. The input parameter rectangle string is parsed to obtain the four coordinates of the rectangle, and the point string is parsed to convert its coordinates into a format that is convenient for comparison. The output of this function is a Boolean value used to convey whether the point is contained within the rectangle.

UDF: ST_Within

ST_WITHIN[2] takes two points on a two-dimensional plane and a distance value as method parameters. Then, it returns whether the two points are within the Euclidean distance of each other in the form of a boolean value, if the distance between the two points is within the Euclidean distance, it is "true"; if the two points If the distance is not within the Euclidean distance, it is "false".

Relevant Query Operations

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

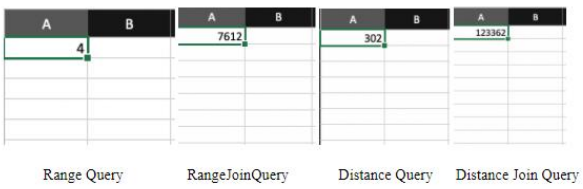
- **RangeQuery**: Given a rectangle R and a set of points P, find all the points within R.
- **RangeJoinQuery**: Given a set of Rectangles R and a set of Points S, find all (Point, Rectangle) pairs such that the point is within the rectangle.
- **DistanceQuery**: Given a set of points P and distance 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 1: Query Results



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.

Phase 2 Query Results:

	A	B
1	-73.897174,40.822518,-73.872773,40.840171	2
2	-73.926629,40.812290,-73.901294,40.824395	1
3	-73.922754,40.667106,-73.900048,40.687222	1
4	-74.010505,40.592545,-73.979640,40.626834	2
5	-73.867911,40.734291,-73.847510,40.748860	1
6	-74.005893,40.707956,-73.991778,40.718025	72
7	-73.984284,40.760945,-73.969802,40.767845	542
8	-73.996242,40.769799,-73.980675,40.781788	267
9	-74.002146,40.644127,-73.980293,40.659678	1
10	-73.840130,40.662481,-73.801134,40.691956	4
11	-73.997506,40.714075,-73.983825,40.724145	308
12	-73.961967,40.758279,-73.947491,40.772348	411
13	-73.884652,40.822974,-73.856153,40.833622	1
14	-73.978371,40.750775,-73.966573,40.762922	830
15	-73.924042,40.691451,-73.877613,40.717801	3
16	-74.009064,40.704880,-73.999196,40.712087	80
17	-73.963079,40.650396,-73.926466,40.664311	2
18	-73.924124,40.712865,-73.887079,40.735202	2
19	-73.988433,40.644821,-73.970843,40.662150	2
20	-73.924905,40.873248,-73.896617,40.890453	1
21	-73.923983,40.765866,-73.887708,40.790954	9
22	-73.938591,40.752146,-73.902027,40.770037	37
23	-73.953486,40.700733,-73.920747,40.728183	15
24	-73.976401,40.792370,-73.960046,40.803364	145
25	-73.983825,40.718849,-73.971628,40.729441	96
26	-73.959648,40.796918,-73.941773,40.811451	44
27	-74.000649,40.681732,-73.992365,40.691203	15
28	-73.993466,40.749795,-73.984119,40.757241	404
29	-73.992564,40.709135,-73.973481,40.721481	135
30	-73.920588,40.884633,-73.896359,40.915541	1

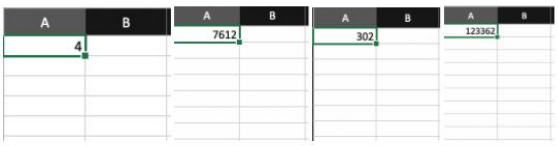
Hotzone Analysis Output

	A	B	C
1	-7399	4075	15
2	-7399	4075	22
3	-7399	4075	14
4	-7399	4075	29
5	-7398	4075	15
6	-7399	4075	16
7	-7399	4075	21
8	-7399	4075	28
9	-7399	4075	23
10	-7399	4075	30
11	-7398	4075	14
12	-7399	4074	15
13	-7398	4075	22
14	-7399	4075	27
15	-7398	4075	29
16	-7398	4075	28
17	-7399	4074	23
18	-7399	4074	22
19	-7399	4074	16
20	-7398	4076	15
21	-7398	4075	21
22	-7399	4075	13
23	-7399	4075	9
24	-7398	4075	23
25	-7398	4075	30
26	-7398	4075	16
27	-7399	4074	30
28	-7398	4076	14
29	-7398	4076	28
30	-7399	4074	29

Hot Cell Analysis Output

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.

3. Results



Range Query RangeJoinQuery Distance Query Distance Join Query

Our implementation succeeded in solving the requirements of hot zone analysis, but not the requirements of hot cell analysis. We admit that due to the limited project time, there are some unsolvable errors and problems. In this section, we describe some of the problems and limitations in the implementation process.

First, our hot cell analysis cannot produce exactly the same output as the sample answer. We believe this is due to the incorrect G score equation we implemented. Although we try our best to explain and understand how to insert all the elements of the equation, it is likely that some elements are wrong or incorrectly calculated.

The second problem/limitation we implemented was the time to run the code. Now, our implementation can complete the calculation in a reasonable time. However, when we try different implementations, sometimes the program runs for more than an hour, which may be due to our hardware and code logic. This problem prevents us from trying many different implementations/tests to figure out what is wrong in the program.

4. Your contribution

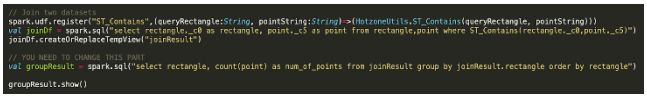
As for the first phase function, I simply separated the longitude and latitude to Double type and then calculated the distance between them based on Euclidean function. The work is shown below:

```
// YOU NEED TO FILL IN THIS USER DEFINED FUNCTION
spark.udf.register("ST_Within",(pointString1:String, pointString2:String, distance:Double)=> {
    // parse point1 and point2
    val point1 = pointString1.split(",")
    var lon1 = point1(0).toDouble
    var lat1 = point1(1).toDouble

    val point2 = pointString2.split(",")
    var lon2 = point2(0).toDouble
    var lat2 = point2(1).toDouble

    // Get the distance using Euclidean between two points
    val euc_d = sqrt(pow(lon1 - lon2, 2) + pow(lat1 - lat2, 2))
    if(euc_d > distance)
        false
    else
        true
})
```

As for the second phase, the Hotzone Analysis is based on Phase 1's ST_Contain function which is implemented by another teammate. I used the idea and wrote down this line of code: "**select rectangle, count(point) as num_of_points from joinResult group by joinResult.rectangle order by rectangle**" as shown below which means joining groups by counting the specific points landing on the specific rectangle area:



5. Lessons learned and a list of team members

From this one-month project, I had several pieces of ideas and advice to my future projects. First of all, the Spark environment is relatively tricky to build. I had a problem running in the command line. Choosing whether to build it on a local machine or virtual machine is a simple problem. I chose to run on my MacBook only and therefore I did a lot of research on how to build Spark on Mac but could not get it done correctly. It was my teammate who reminded me of the Java version on the computer and I fixed it to Java 1.8 and then it worked. Second of all, the data on the personal Spark website should be shown correctly without any timely issue. This required us to build a Master-Slave environment. We discussed about how this works on a shared Google Doc and here is the solution:

a. Build the environment based on Spark Standalone Mode
b. Run the Master using ./sbin/start-master.sh -h 192.168.1.11 (master's IP)
c. Run the webpage using ./sbin/start-master.sh
d. Run the Slave using ./sbin/start-slave.sh spark://192.168.1.11:7077 (Slave's IP)
e. Refresh the webpage and see if the Worker(s) is(are) there.

Another useful lesson I learnt was the scala code for the Hotzone analysis. I believe it is the common process for data analysis which is Parsing -> Loading -> Joining. Parsing the given data which are points and rectangles and then loading the data from .csv file is essential for the joining part. Counting the points in the specific rectangle is easier otherwise harder if we use two datasets separately.

Team member: Riley Tallman, Paul Butler, Devan Pratt, Sujith Reddy Peddi, Aishwarya Srisankaran, Wenzhe Zheng

Date	Progress	Work done by
10/18/2020	Created a Git repository and installed Apache spark and sparkSQL	Paul Butler, Sujith Reddy Peddi, Devan Pratt, Aishwarya Srisankaran, Riley Tallman, Wenzhe Zheng
10/19/2020	Background research on Apache SparkSQL and spatial queries.	Paul Butler, Sujith Reddy Peddi, Devan Pratt, Aishwarya Srisankaran, Riley Tallman, Wenzhe Zheng
10/22/2020	Phase 1 ST_Contains method implementation and passing the given test case	Riley Tallman, Paul Butler, Devan Pratt
11/31/2020	Phase 1 ST_Within method implementation and passing the given test case	Riley Tallman, Paul Butler
11/10/2020	Phase 2 - Hot zone analysis	Riley Tallman, Devan Pratt
11/16/2020	Phase 2 - Hot cell analysis	Sujith Reddy Peddi, Devan Pratt, Aishwarya Srisankaran, Wenzhe Zheng
11/20/2020	Systems Documentation Report	Sujith Reddy Peddi, Aishwarya Srisankaran, Wenzhe Zheng, Riley Tallman

Our team had a shared doc that whoever encountered a problem could share it with or without the solution while others could see them and try to solve them. I finally knew how to deal with FileNotFoundException. The best solution is putting every file that is being used to a folder which is one level lower than the root folder. Only this way can we figure out which files are missing, and which are redundant. I learnt a lot of lessons and skills based on the project and hopefully I can solve bigger projects easier and faster than this one.

6. REFERENCES

[1] "st_contains." ST_Contains (Oracle Big Data Spatial and Graph Vector Analysis Java API Reference), 3 Aug. 2016, docs.oracle.com/bigdata/bda46/BDVAJ/oracle/spatial/hadoop/vector/hive/function/ST_Contains.html.

[2] "Edit This Page On: Gitlab Github." PostGIS, postgis.net/workshops/postgis-intro/spatial_relationships.html.