SPATIO-TEMPORAL DATA ANALYSIS USING APACHE SPARK

Systems Documentation

Craig Manning, Will Cray, Jake Grogan

Contents

Overview

Abstract

The goal of this project is to implement geo-spatial analysis of taxicab rides in NYC. Specifically, this project will identify "zones" with the highest number of pick-ups in the city as well as the Getis-Ord G_i statistic (a geo-spatial z-score) for each zone to identify statistically significant hot spots in the city.

Purpose

This project is designed to showcase the use of Apache Spark in managing and analyzing spatial data. Specifically, it provides insight into the use of clustered computing in determining statistically significant spatial areas for use in predictive analytics.

Goals

This project encompasses a distributed database built on Scala and Apache Spark. The purpose of this database is twofold as described below.

1. Build Spark SQL functionality to support the following:
   - **Range queries:** given a rectangle R covering a certain latitude and longitude and a set of points P, find all the points within R
   - **Range join queries:** given a set of rectangles R covering a certain latitude and longitude and a set of points P, find all the (r_i, s_i) pairs such that the point is within the rectangle
   - **Distance queries:** given a point location P and distance D in km, find all points that lie within the distance D from P
   - **Distance join queries:** given two sets of points$P_1$and P_2 and a distance D in km, find all pairs (P_{1i}, P_{2j}) such that P_{1i} is within distance D of P_{2j}
2. Conduct spatial Hot Spot analysis – This task will focus on applying spatial statistics to spatio-temporal big data in order to identify statistically significant spatial hot spots using Apache Spark and the above functionality

Non-Goals

The below are not addressed in this project:

1. Discuss future applications of this project – it is designed to showcase the technology vs discuss business implications
2. Discuss alternative architectures and their merits in conducting geo-spatial analysis

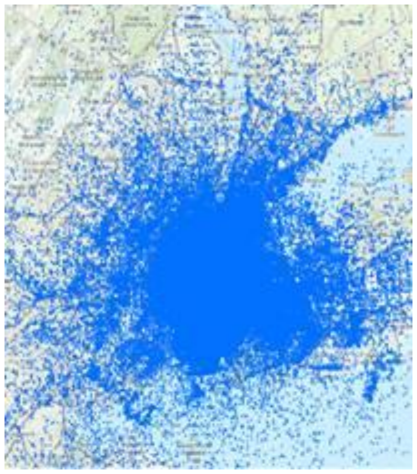
Problem Definition

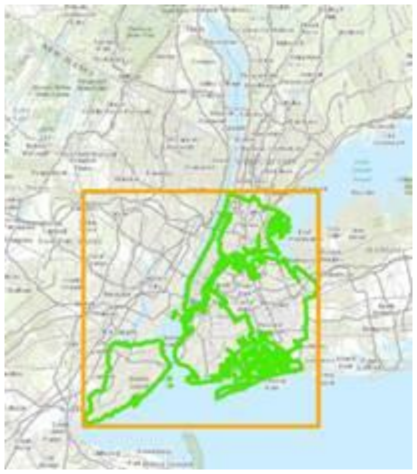
Description

The Getis-Ord G_i^* statistic is commonly used when identifying statistically significant clusters (often termed Hot Spots). It provides a z-score and p-values that allow users to determine where features with either high or low values are clustered spatially. Given the state of our world, the amount of observational data is increasing at an exponential rate. Distributed computing is required to handle these large collections of observational data and Spark has proven to be highly capable in handling these workloads. Thus, conducting spatial-temporal analysis on top of a Spark framework is the focus of this project.

Input

A collection of New York City Yellow Cab taxi trip records spanning January 2009 to June 2015. The source data may be clipped to an envelope encompassing the five New York City boroughs in order to remove some of the noisy error data (e.g., latitude 40.5N – 40.9N, longitude 73.7W – 74.25W).





Output

A list of the fifty most significant hot spot cells in time and space as identified using the Getis-Ord G_i statistic.

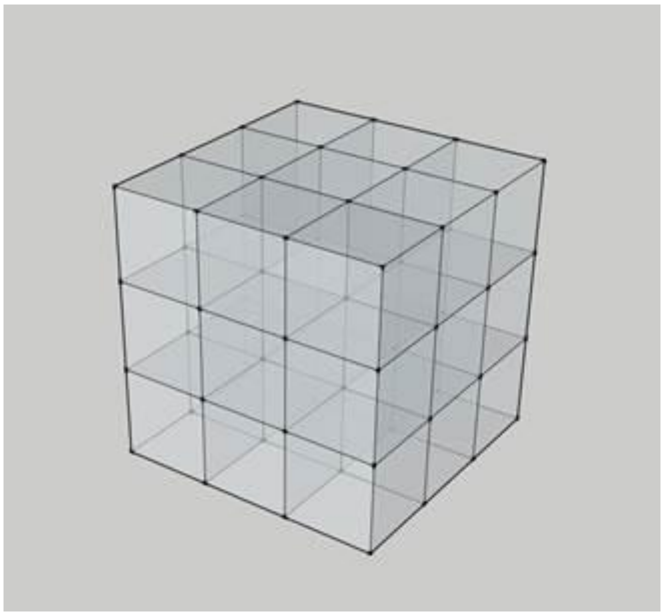
$$G_i^* = \frac{\sum_{j=1}^{n} w_{i,j}x_j - \bar{X}\sum_{j=1}^{n} w_{i,j}}{S\sqrt{\frac{\left[n\sum_{j=1}^{n} w_{i,j}^2 - \left(\sum_{j=1}^{n} w_{i,j}\right)^2\right]}{n-1}}}$$

where $x_j$is the attribute value for cell j, $w_{i,j}$ is the spatial weight between cell i and j, n is equal to the total number of cells, and:

$$\bar{X} = \frac{\sum_{j=1}^{n} x_j}{n}$$

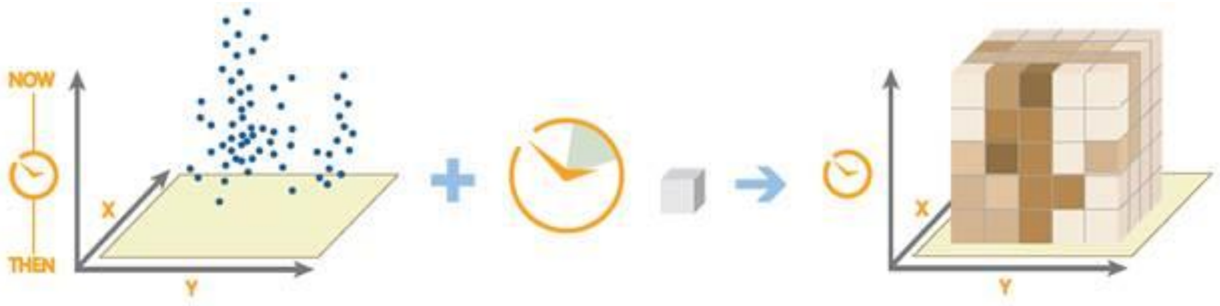
$$S = \sqrt{\frac{\sum_{j=1}^{n} x_j^2}{n} - (\bar{X})^2}$$

The G_i^*statistic is a z-score, so no further calculations are required. The neighborhood for each cell in the space-time cube is established by the neighbors in a grid based on subdividing latitude and longitude uniformly. This spatial neighborhood is created for the preceding, current, and following time periods (i.e., each cell has 26 neighbors). For simplicity of computation, the weight of each neighbor cell is presumed to be equal.



Constraints

Time and space are aggregated into cube cells to form a space-time cube.



Requirements

Assumptions

No.	Assumption
1	The program will run on top of the Apache Spark open source framework
2	The program will be implemented using Scala
3	Data will be stored only in csv format
4	The programs will be tested using a cluster of commodity-level PCs

Functional Requirements

No.	Requirement
1	Load point data from csv files
2	Determine all the points within a given rectangle
3	Find all (point, rectangle) pairs for given points and rectangles
4	Find all points that lie within a certain distance of a single point
5	Find all pairs of points that are within a certain distance of each other
6	Find the number of points in a series of rectangles and display in decreasing order of count
7	Calculate Gi and return a list in decreasing order

Acceptance criteria

The tasks are presumed to be accomplished when the tests are successfully passed. Any work to ensure clean and efficient code is considered follow on work and beyond the scope of this project. Specific evaluation criteria are defined below.

Evaluation will be done using two metrics:

1. Correctness of the result identifying spatial-temporal hot spots
2. Correctness of the Getis-Ord statistic and identification of statistically significant zones for predictive analytics

Representing a hotspot as a single spatial-temporal cell in the space-time cube, we evaluate correctness using the Jaccard similarity coefficient of the reference result and the candidates result for the subset of the top fifty “hot zones”. This will in effect ignore small perturbations in the ordering of the top results.

Design

This project was implemented by creating two functions to calculate Range/Range Join/Distance/Distance Join queries (ST_Contains and ST_Within) and two functions to conduct spatial-temporal analysis (runHotZoneAnalysis and runHotCellAnalysis) as described below.

ST_Contains

This function takes two parameters, a point and a rectangle, and returns a Boolean value representing if the given point is within the given rectangle.

Input: pointString:String, queryRectangle:String

Output: Boolean (True or False)

Steps:

1. Lat-long for input points are parsed into x and y variables
2. Lat-long for the rectangle represent the diagonal and are parsed into x_1, y_1 and x_2, y_2
3. Check if x is between x_1 and x_2 and return associated Boolean value
4. Check if y is between y_1 and y_2 and return associated Boolean value
5. If both Boolean values return as True – point is contained in the rectangle

ST_Within

This function takes three parameters, two points and a distance, and returns a Boolean value representing if the distance between the two points is less than or equal to the given distance.

Input: pointString1:String, pointString2:String, distance:Double

Output: Boolean (True or False)

Steps:

1. Lat-long for input points are parsed into x_1, y_1 and x_2, y_2
2. Distance value is parsed into variable D
3. Calculate the Euclidean distance between the two input points: $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
4. Compare $d <= D$ and return the associated Boolean value

runHotZoneAnalysis

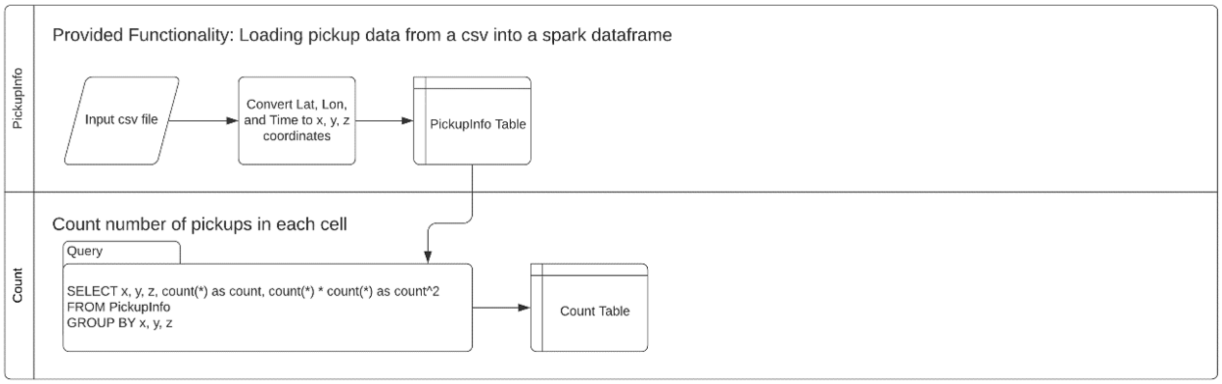
Input: pointString:String, queryRectangle:String

```
test/output hotzoneanalysis src/resources/point-hotzone.csv src/resources/zone-hotzone.csv
```

Output: DataFrame containing all zones identified by their lat-long coordinates and the point count per zone

```
"-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
```

Steps:



PickupInfo
Provided Functionality: Loading pickup data from a csv into a spark dataframe
Input csv file
Convert Lat, Lon, and Time to x, y, z coordinates
PickupInfo Table
Count
Count number of pickups in each cell
Query
SELECT x, y, z, count(*) as count, count(*) * count(*) as count^2
FROM PickupInfo
GROUP BY x, y, z
Count Table

runHotCellAnalysis

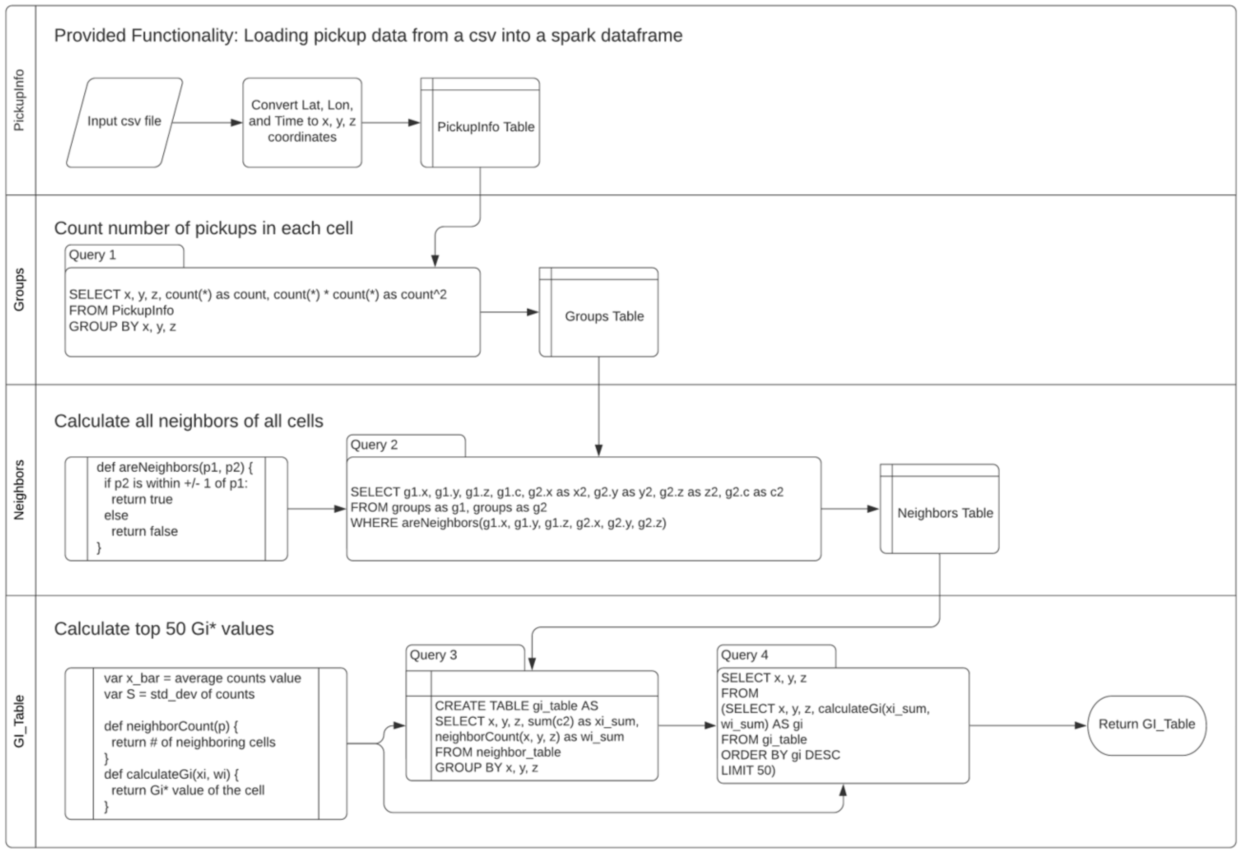
Input: pointString:String, queryRectangle:String

```
test/output hotcellanalysis src/resources/yellow_tripdata_2009-01_point.csv
```

Output: DataFrame containing all zones identified by their simplified lat-long coordinates (lat-long x 100), the point count per zone, and the G-score in descending order.

```
-7399,4075,15,79.39845481859541
-7399,4075,22,77.06821726123778
-7399,4075,14,76.2526341161626
-7399,4075,29,76.05844765304946
-7398,4075,15,75.61181878788925
-7399,4075,16,75.2817018005969
```

Steps:


PickupInfo
Provided Functionality: Loading pickup data from a csv into a spark dataframe
Input csv file
Convert Lat, Lon, and Time to x, y, z coordinates
PickupInfo Table
Groups
Count number of pickups in each cell
Query 1
Groups Table
Neighbors
Calculate all neighbors of all cells
Query 2
Neighbors Table
GI_Table
Calculate top 50 Gi* values
Query 3
Query 4
Return GI_Table

```
SELECT x, y, z, count(*) as count, count(*) * count(*) as count^2
FROM PickupInfo
GROUP BY x, y, z
```

```
def areNeighbors(p1, p2) {
  if p2 is within +/- 1 of p1:
    return true
  else
    return false
}
```

```
SELECT g1.x, g1.y, g1.z, g1.c, g2.x as x2, g2.y as y2, g2.z as z2, g2.c as c2
FROM groups as g1, groups as g2
WHERE areNeighbors(g1.x, g1.y, g1.z, g2.x, g2.y, g2.z)
```

```
var x_bar = average counts value
var S = std_dev of counts

def neighborCount(p) {
  return # of neighboring cells
}
def calculateGi(xi, wi) {
  return Gi* value of the cell
}
```

```
CREATE TABLE gi_table AS
SELECT x, y, z, sum(c2) as xi_sum,
neighborCount(x, y, z) as wi_sum
FROM neighbor_table
GROUP BY x, y, z
```

```
SELECT x, y, z
FROM
(SELECT x, y, z, calculateGi(xi_sum,
wi_sum) AS gi
FROM gi_table
ORDER BY gi DESC
LIMIT 50)
```

Project Development

Work Distribution

Work for this project was distributed using a round robin approach with certain tasks being assigned based on interest/skill.

Team Member	Focus Area
Craig Manning	Wrote the `ST_Within()` function, resolved the primary problems with the Hotzone Analysis, and pioneered the MapReduce approach used in the Hotcell Analysis.
Will Cray	Wrote the `ST_Contains()` function and two `HotzoneUtils.scala` member functions used to calculate the number of neighbors for a given point, resolved a working algorithm for the required z-scores.
Jake Grogan	Lead documentation initiatives, contributed to the process in `HotzoneAnalysis.scala`, found bugs in `HotzoneUtils.scala`, and helped resolve the key calculations behind the Getis-Ord statistic.

Questions

Question
What are the performance tradeoffs between using a `cross-join` on a Spark dataframe versus a `map` and `reduce` call?
How would this tradeoff be affected on a true distributed compute architecture vs. testing on a development machine?
What would the implementation look like in Hadoop? Would it be more difficult to implement?
How does MapReduce work on Spark?

Lessons Learned

Our largest pitfall was our attempt to use MapReduce with Scala to perform the G-score calculations. We had our functions defined and working inside of HotcellUtils.scala but getting the code that invoked those functions across the distributed datasets in Spark to compile remained problematic. There were several factors that resulted in countless wasted hours:

1. Unfamiliarity with Spark/Scala in general - This one is general, but it definitely complicated the task of how to effectively use Spark
2. Uncertainty about the AutoGrader's environment – several of the solutions we attempted online required new headers and files. The AutoGrader seemed to be very sensitive, and we never knew what we could (or couldn't) include without breaking the environment resulting in an automatic failure.
3. Misunderstanding about how MapReduce works on Spark – every approach we took failed because our understanding of exactly how Spark was dealing with our commands was not well formed.

After sinking many hours into this approach, we decided to work with SQL queries and leverage the distributed/parallel efficiency that SQL already provides. Additionally, we had ample examples of invoking SQL queries in Scala thanks to the provided code templates, as well as our successful code from HotzoneAnalyzis.scala.