Geospatial Data Visualization

Raghav Aggarwal

Structure of Dataframe

In the duration of the project, a dataframe was used to store the trajectory dataset. The content of the dataframe was parsed from the input JSON file. A function spark.read.option() was used to parse the JSON file. The dataframe was consisted in five parts: the vehicle_id, the trajectory_id, the timestamp, the longitude and the latitude.
In the given JSON file, the trajectory was consisted with two parts: location and timestamp. The spark.read.option() parsed the trajectory as three separate parts: trajectory.timestamp as timestamp, trajectory.location[1] as lon(which stood for longitude) and trajectory.location[0] as lat(which stood for latitude).
Here are two diagrams showing what the JSON file and the parsed dataframe looked like.

Figure 1: JSON File

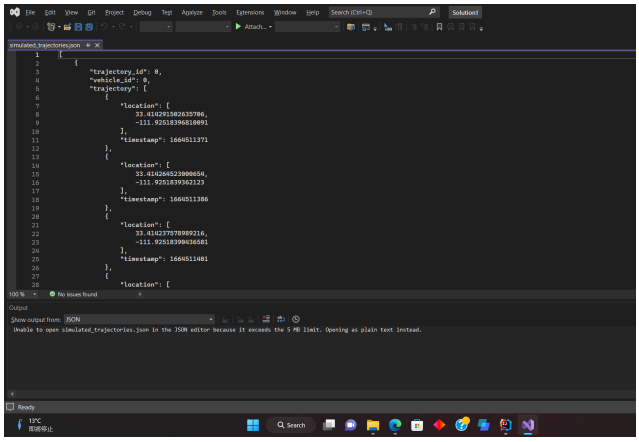
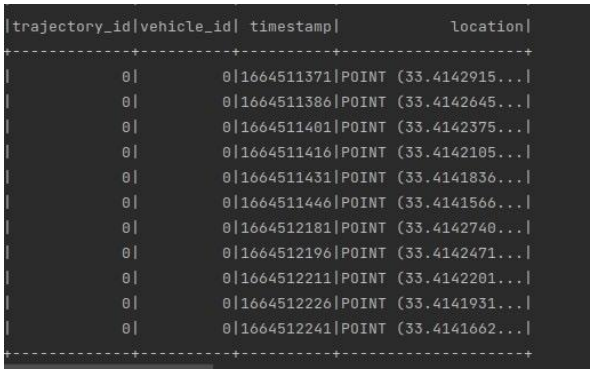


Figure 2: Parsed Dataframe



trajectory_id	vehicle_id	timestamp	location
0	0	1664511371	POINT (33.4142915...
0	0	1664511386	POINT (33.4142645...
0	0	1664511401	POINT (33.4142375...
0	0	1664511416	POINT (33.4142105...
0	0	1664511431	POINT (33.4141836...
0	0	1664511446	POINT (33.4141566...
0	0	1664512181	POINT (33.4142740...
0	0	1664512196	POINT (33.4142471...
0	0	1664512211	POINT (33.4142201...
0	0	1664512226	POINT (33.4141931...
0	0	1664512241	POINT (33.4141662...

Algorithms Implemented

The project required to implement three types of trajectory queries: Spatiotemporal Range Query, Similarity Query and Knn Query. These were implemented using Spark 3.0.3, Hadoop 2.7.7 and Java 1.8. A small background knowledge before diving into the algorithms. In this project, we are working with trajectory of moving objects. It is a spatiotemporal data generated by moving objects, represented by longitude, latitude and a timestamp.

To begin with, we started by implementing Spatial Range Query or getSpatiaRange method(figure 3). We are given dataset, spatial range which includes min longtitude, max longitude, min latitude, and max latitude. And method will return all the trajectory line segments such that all position points of line segment are within the spatial range.

```scala
def getSpatialRange(spark: SparkSession, dfTrajectory: DataFrame, latMin: Double, lonMin: Double, latMax: Double, lonMax: Double): DataFrame = {
  val checkLon = (lon: Double) => {
    var a = 1
    if (lon >= lonMin && lon <= lonMax) {
      a = 0;
    }
    a
  }
  val checkLonUDF = udf(checkLon)

  val checkLat = (lat: Double) => {
    var a = 1
    if (lat >= latMin && lat <= latMax) {
      a = 0;
    }
    a
  }
  val checkLatUDF = udf(checkLat)

  val dfValid = dfTrajectory.select(col("trajectory_id"), col("vehicle_id"), col("timestamp"), col("lat"), col("lon"),
    checkLatUDF(col("lat")).as("latcheck"), checkLonUDF(col("lon")).as("loncheck"))
  dfValid.createOrReplaceTempView("tempView")
  val result = spark.sql("SELECT trajectory_id, vehicle_id, timestamp, ST_Point(CAST(lat AS Decimal(24,20)), CAST(lon AS Decimal(24,20))) as location FROM tempView where  latcheck = 0 and loncheck = 0")


  result
}
```

Figure 3: Code for getSpatialRange

In the getSpatiaRange method (figure 3), we made two user implemented functions (UDF), checkLon and checkLat. CheckLon checks if the longitude is in the range of the Longitudes and CheckLat checks if the latitude is in the range of the Latitudes. Then we have a spark SQL command that is outputting trajectory_id, vehicle_id, timestamp, longitude and latitude if both longitude and latitude are in the given range. The output can been seen in figure 4.

ManageTrajectory.scala • {} part-00000-faf47816-02ee-469b-90ef-3a4584668538-c000.json 1 ×

Users > raghavmac > Desktop > Project-Phase-1 > SDSE-Phase-1 > data > output > get-spatial-range > {} part-00000-faf47816-02ee-469b-90ef-3a4584668538-c000.json > ...


```
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664511371,"location":[1,1,0,0,32,0,0,0,0,96,106,3,-127,7,-75,64,64,-71,113,-47,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664511386,"location":[1,1,0,0,32,0,0,0,0,38,23,-79,-98,6,-75,64,64,64,52,-81,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664511401,"location":[1,1,0,0,32,0,0,0,0,54,68,-85,-68,5,-75,64,64,90,2,-115,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664511416,"location":[1,1,0,0,32,0,0,0,0,-3,-16,88,-38,4,-75,64,64,-31,-60,106,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664511431,"location":[1,1,0,0,32,0,0,0,0,12,30,83,-8,3,-75,64,64,-5,-110,72,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664511446,"location":[1,1,0,0,32,0,0,0,0,-45,-54,0,22,3,-75,64,64,-126,85,38,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664512181,"location":[1,1,0,0,32,0,0,0,0,30,-94,-31,-18,6,-75,64,64,2,86,-69,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664512196,"location":[1,1,0,0,32,0,0,0,0,-27,78,-113,12,6,-75,64,64,-119,24,-103,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664512211,"location":[1,1,0,0,32,0,0,0,0,-12,123,-119,42,5,-75,64,64,-93,-26,118,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664512226,"location":[1,1,0,0,32,0,0,0,0,-69,40,55,72,4,-75,64,64,42,-87,84,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664512241,"location":[1,1,0,0,32,0,0,0,0,-53,85,49,102,3,-75,64,64,67,119,50,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664515751,"location":[1,1,0,0,32,0,0,0,0,38,23,-79,-98,6,-75,64,64,64,52,-81,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664515766,"location":[1,1,0,0,32,0,0,0,0,54,68,-85,-68,5,-75,64,64,90,2,-115,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664515781,"location":[1,1,0,0,32,0,0,0,0,-3,-16,88,-38,4,-75,64,64,-31,-60,106,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664515796,"location":[1,1,0,0,32,0,0,0,0,12,30,83,-8,3,-75,64,64,-5,-110,72,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664515811,"location":[1,1,0,0,32,0,0,0,0,-45,-54,0,22,3,-75,64,64,-126,85,38,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664516546,"location":[1,1,0,0,32,0,0,0,0,30,-94,-31,-18,6,-75,64,64,2,86,-69,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664516561,"location":[1,1,0,0,32,0,0,0,0,-27,78,-113,12,6,-75,64,64,-119,24,-103,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664516591,"location":[1,1,0,0,32,0,0,0,0,-69,40,55,72,4,-75,64,64,42,-87,84,54,54,-5,91,-64]}
{"trajectory_id":0,"vehicle_id":0,"timestamp":1664516606,"location":[1,1,0,0,32,0,0,0,0,-53,85,49,102,3,-75,64,64,67,119,50,54,54,-5,91,-64]}
{"trajectory_id":63,"vehicle_id":63,"timestamp":1664514836,"location":[1,1,0,0,32,0,0,0,0,16,-100,-9,24,7,-75,64,64,113,-54,104,-126,15,-5,91,-64]}
{"trajectory_id":63,"vehicle_id":63,"timestamp":1664514851,"location":[1,1,0,0,32,0,0,0,0,108,71,65,54,6,-75,64,64,102,7,16,0,16,-5,91,-64]}
{"trajectory_id":63,"vehicle_id":63,"timestamp":1664514866,"location":[1,1,0,0,32,0,0,0,0,93,-128,-97,84,5,-75,64,64,5,-27,14,-126,15,-5,91,-64]}
{"trajectory_id":63,"vehicle_id":63,"timestamp":1664514881,"location":[1,1,0,0,32,0,0,0,0,97,50,77,114,4,-75,64,64,-75,-22,-31,-127,15,-5,91,-64]}
{"trajectory_id":63,"vehicle_id":63,"timestamp":1664514896,"location":[1,1,0,0,32,0,0,0,0,4,94,-29,-113,3,-75,64,64,-34,54,-119,-1,15,-5,91,-64]}
{"trajectory_id":73,"vehicle_id":73,"timestamp":1664515076,"location":[1,1,0,0,32,0,0,0,0,54,68,-85,-68,5,-75,64,64,90,2,-115,54,54,-5,91,-64]}
{"trajectory_id":73,"vehicle_id":73,"timestamp":1664515106,"location":[1,1,0,0,32,0,0,0,0,12,30,83,-8,3,-75,64,64,-5,-110,72,54,54,-5,91,-64]}
```


Figure 4:Output for getSpatialRange

In addition, we then implemented getSpatioTemporalRange method (figure 5). We are given the dataset, spatial range and time range. And method will return all trajectory line segment such that all position of the line segment are within the spatial and temporal range.

```scala
def getSpatioTemporalRange(spark: SparkSession, dfTrajectory: DataFrame, timeMin: Long, timeMax: Long, latMin: Double, lonMin: Double, latMax: Double, lonMax: Double): DataFrame =
{
  val checkTime = (time: Long) => {
    var a = 1
    if (time >= timeMin && time <= timeMax) {
      a = 0;
    }
    a
  }
  val checkTimeUDF = udf(checkTime)

  val checkLon = (lon: Double) => {
    var a = 1
    if (lon >= lonMin && lon <= lonMax) {
      a = 0;
    }
    a
  }
  val checkLonUDF = udf(checkLon)

  val checkLat = (lat: Double) => {
    var a = 1
    if (lat >= latMin && lat <= latMax) {
      a = 0;
    }
    a
  }
  val checkLatUDF = udf(checkLat)

  val dfValid = dfTrajectory.select(col("trajectory_id"), col("vehicle_id"), col("timestamp"), col("lat"), col("lon"),
    checkLatUDF(col("lat")).as("latcheck"), checkLonUDF(col("lon")).as("loncheck"), checkTimeUDF(col("timestamp")).as("timecheck"))
  dfValid.createOrReplaceTempView("tempView")
  val result = spark.sql("SELECT trajectory_id, vehicle_id, timestamp, ST_Point(CAST(lat AS Decimal(24,20)), CAST(lon AS Decimal(24,20))) as location FROM tempView where  latcheck = 0 and loncheck = 0 and timecheck = 0")

  result
}
```

Figure 5: code for getSpatioTemporalRange

In getSpatioTemporalRange method (figure 5) we are using the same UDF's from the getSpatiaRange method, checkLon and checkLat with an addition of checkTime, which checks if the time is in the given range or not. In the Spark SQL query we are printing trajectory_id, vehicle_id, timestamp, longitude and latitude if longitude, latitude and time are in the given range. The output can been seen in figure 6.

Figure 6: Output for getSpatioTemporalRange

Lastly we implemented getKNNTrajectory method (figure 7) or KNN query. We are given a dataset, a query Trajectory, positive integer K, and distance function. And the method is returning K nearest trajectories for the given trajectory, based on the minimum distance between trajectories. If this distance is same for multiple trajectories we choose the trajectories with ascending order of trajectory id till we fulfill the limit.

Figure 7: Input for getKNNTrajectory

In getKNNTrajectory method (figure 7) we are making a new view to work with. In the spark sql query we are selecting trajectory id such that the distance of from that trajectory to every other trajectory in the dataframe is minimum and If this distance is same for multiple trajectories we choose the trajectories with ascending order of trajectory id till we fulfill the limit, set by the neighbors. The output can been seen in figure 8.

Figure 8: Output for getKNNTrajectory

Implementation of API, front end, and visualization layer

To finish this demo, we construct the front end with React. React is the most popular JavaScript framework in the world. It was created by Jordan Walke, a software engineer at Facebook, and open-sourced in March 2015. Rather than using the simple HTML, CSS, JavaScript, it works mostly with

JSX. It is a JavaScript syntax extension that describes how the user interface should look. JSX may look like a template language, but it has the full power of JavaScript. Moreover, components are also a very important property of React js. Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML. Components are classified into two types: class components and function components. We can see the base map with the initial view state focused on the Tempe, Arizona area when we connect to the local host. Then, in order to display the trips layer, we must upload the trajectory data to our interface.Therefore, we created two components to implement the file upload and trips layer function. First one is file-uploader.js. In this component, it allows us to upload the trajectory file to our interface. Also, we specified the upload file type must be ended in .Json. We cannot upload other types of files. In the trips-chart.js, we tried to implement the base map (Figure 1) showing on the interface. Also, we can see the 'animation' (moving route) showing on the base map after we upload the data.(Figure 2). In this trip layer, we specified the style, time, and the animation speed. Most important thing is we provide three tabs for users to do the spatial queries. First one is a normal spatial query, the second one is a spatial temporal query, and the last one is the KNN query.

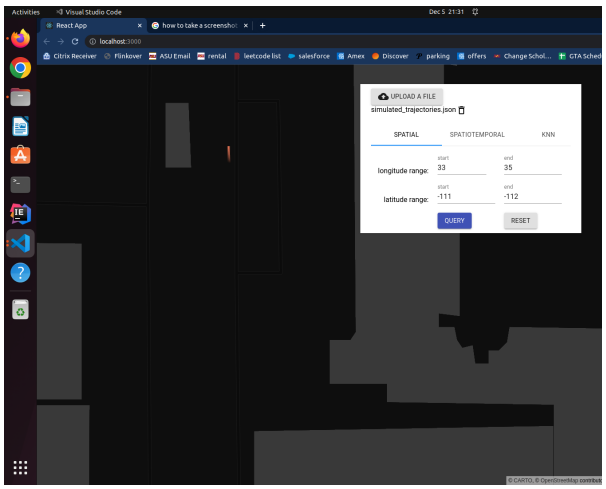
Figure 9: Basic map

Figure 10: After uploading the complete Trajectory Data:

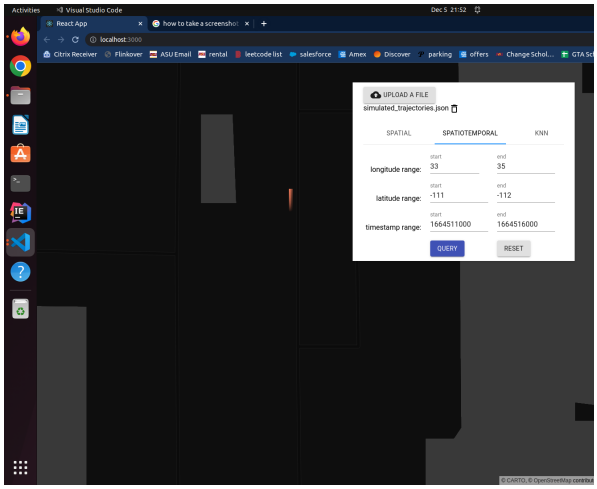
For the Backend and the API, we first tried to find a framework that can work perfectly with the Scala programming language because our phase1 codes were written in Scala. After doing a lot of research, we tried to use the Play framework. It is compatible with the Scala programming language, has a lightweight, stateless, web-friendly architecture, and can be started with a simple sbt command-line tool by the command 'sbt new playframework /play-scala-seed.g8'. However, we are not that familiar with Scala, so

we are not able to create the endpoint to implement Post, Get, Update, Delete. As the second plan, we first convert the phase 1 Scala codes into Python codes. Then, we tried the Flask framework to work as the API endpoint. Flask is a Python API that allows us to create web applications. Since less base code is needed to implement a straightforward web application with Flask than it is with Django, it is also simpler to learn. A web framework, also referred to as a web-application framework, is a group of modules and libraries that let programmers create applications without having to write low-level code for things like protocols and thread management. Therefore, our basic workflow is to create an API with Flask framework, we input the query values for spatial query, spatial temporal query, and the KNN query, and these values will be sent to the well-set endpoint by clicking the query button. It works as the 'Post' and 'Get' function for Restful API. After computing the query results from the backend functions with the parameters sent from the frontend, the results will be sent back to the frontend and display the result trip layer on the base map.

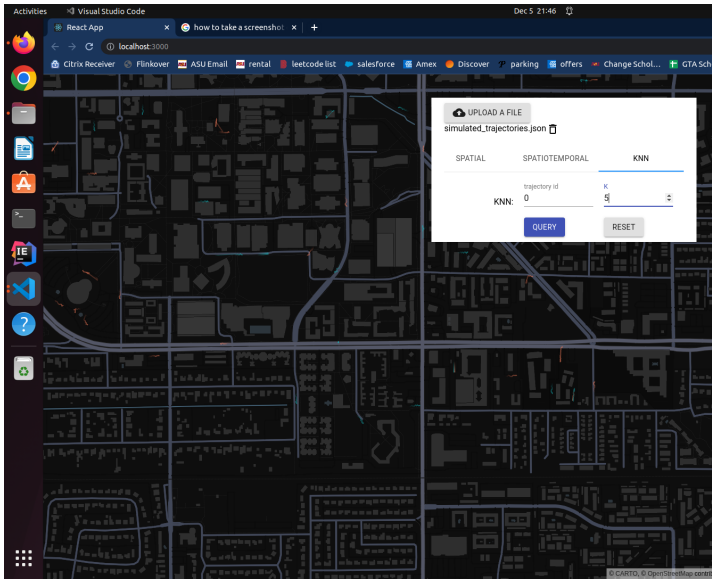
Figure 11: Running Spatial Query



Output of Spatial



Output of SpatioTemporal



Output of KNN