

# CSE 594- Spatial Data Science and Engineering

## Project Phase 3

### Team Members (Group 7)

Sandhya Balu – 1223737762

Kanishk Tanotra – 1222279162

Ajay kumar Janapareddi – 1225475771

### Structure of Apache Sedona DataFrame

We had to complete the loadTrajectoryData() function in phase 1 of the given project, which takes a spark session and the filePath of the dataset as input. After loading the data set, we had to return the final data frame. We created a final dataframe structure using the code below.

Code screenshot

```
def loadTrajectoryData(spark: SparkSession, filePath: String): DataFrame =
{
    val sqlContext = new org.apache.spark.sql.SQLContext(spark.sparkContext)

    var df = sqlContext.read.option("multiline", "true").json(filePath)

    val sampledf = df.withColumn("trajectory", explode(df("trajectory")))
    df = sampledf
        .withColumn("location", sampledf("trajectory.location"))
        .withColumn("timestamp", sampledf("trajectory.timestamp"))
        .drop("trajectory")
        .withColumn("latitude", col("location").getItem(0))
        .withColumn("longitude", col("location").getItem(1))

    df.createOrReplaceTempView("trajectory")
    df = spark.sql("SELECT trajectory_id, vehicle_id, location, ST_POINT(latitude, longitude) AS point, timestamp FROM trajectory")
    df.show()
    df.printSchema()
    df
    // change the null to desired spark DataFrame object
}
```

To begin, we create a sqlContext in order to create a dataframe, register the table as a dataframe, and then perform or execute SQL queries on it. The sqlContext is used to read the given json file into a temporary dataframe. We use the filePath to read the given json as a dataframe. Following that, we restructured our table so that we could run SQL queries on it efficiently.

The trajectory contains the location and timestamp data for each trajectory id. To obtain location and timestamps, we first flatten the nested trajectory array. We drop the trajectory column once we have our timestamps and locations. Finally, we use our new location column to create separate longitude and latitude columns.

There the final structure of the dataframe is

Trajectory_id	integer
Vehicle_id	integer
Location	array
Timestamp	Timestamp
Latitude	double
Longitude	double

Structure of table

Trajectory_id	Vehicle_id	Location	Timestamp	Latitude	Longitude
---------------	------------	----------	-----------	----------	-----------

We later name the view as trajectory. In the end we perform an SQL query on the view and return this fetched dataframe as results.

```
{
  "type": "array",
  "items": [
    {
      "type": "object",
      "properties": {
        "trajectory_id": {
          "type": "integer"
        },
        "vehicle_id": {
          "type": "integer"
        },
        "trajectory": {
          "type": "array",
          "items": [
            {
              "type": "object",
              "properties": {
                "location": {
                  "type": "array",
                  "items": [
                    {
                      "type": "number"
                    },
                    {
                      "type": "number"
                    }
                  ]
                },
                "timestamp": {
                  "type": "integer"
                }
              }
            }
          ]
        },
        "required": [
          "location",
          "timestamp"
        ]
      }
    }
  ]
},
{
  "required": [
    "trajectory_id",
    "vehicle_id",
    "trajectory"
  ]
}
]
```

## Algorithms:

### *Spatial range query*

#### → Preprocessing:

- ◆ We create a Temporary view of the dataframe (dfTrajectory) as Range\_Data\_Table

Spatial range query is used for selecting trajectory id, vehicle id, timestamp as an array when the trajectory is within the polygon range.

#### → Main Spatial range query

- ◆ We are grouping the given spatial data based on vehicle\_id and trajectory\_id
- ◆ And we create a polygon from the envelope points minimum latitude (latMin), minimum longitude (lonMin), maximum latitude (latMax) and maximum longitude (lonMax) [using *ST\_PolygonFromEnvelope*]
- ◆ We check if any of the given points lies inside the polygon formed. [*ST\_Within*]
- ◆ We get the trajectory\_id, vehicle\_id and list of all timestamps and list of locations of cab pickups.

#### → Postprocess:

- ◆ We created a temporary view (spatial\_output\_frame) of the dataframe created from spark sql command.
- ◆ For the output, we have just selected trajectory\_id, vehicle\_id, timestamp, location from the spatial\_output\_frame

**Note:** Spatio range query (*ST\_Within*) uses either R-tree or k-d tree algorithms to get each query in log-n time.

### *Spatiotemporal range query*

→ Preprocessing:

- ◆ We create a Temporary view of the data frame (dfTrajectory) as Temporal\_Data\_Table

Spatio temporal range query is used for selecting trajectory id, vehicle id, timestamp as an array when the trajectory is within the polygon range and timestamp is between max-time and min-time.

→ Main Spatial temporal range query

- ◆ We are grouping the given spatial data based on vehicle\_id and trajectory\_id
- ◆ And we create a polygon from the envelope points minimum latitude (latMin), minimum longitude (lonMin), maximum latitude (latMax) and maximum longitude (lonMax) [using *ST\_PolygonFromEnvelope*]
- ◆ We select the data that lies inside the polygon formed. [*ST\_Within*] and timestamp from Temporal\_Data\_Table is between the given timeMin and timeMax.
- ◆ We get the trajectory\_id, vehicle\_id and list of all timestamps and list of locations of cab pickups.

→ Postprocess:

- ◆ We created a temporary view (Temporal\_Output\_Table) of the data frame created from spark sql command.
- ◆ For the output, we have just selected trajectory\_id, vehicle\_id, timestamp, and location from the Temporal\_Output\_Table

**Note:** Spatio temporal range query (*ST\_Within*) uses either R-tree or k-d tree algorithms to get each query in log-n time.

**KNN Query:**

→ Preprocessing:

- ◆ We create a Temporary view of the dataframe (Trajectory) as KNN\_Data\_Table

→ Main KNN query

- ◆ We created a temporary view (Trajectory\_Data) of the trajectory dataframe the initialized KNN\_Data\_Table for a given trajectory\_id. Here we basically filter the population with the given trajectory\_id
- ◆ Similarly we created a temporary view (Remaining\_Trajectory\_Data) for the remaining dataset as well. (trajectory\_id is not given value)
- ◆ We calculate the distance between all the points from Trajectory\_Data to the points in Remaining\_Trajectory\_Data and sort them by ascending order and select all the trajectory\_id 's and distance from Remaining\_Trajectory\_Data.
- ◆ We store this as Result\_Trajectory

→ Postprocess:

- ◆ For each trajectory, we take the minimum of all distances calculated earlier.
- ◆ Return all the trajectory ids within the given number of neighbors.

Basic KNN algorithm:

- Calculate the distances between each point and the query point.
- By distance, sort those points.
- Return first K items

The above algorithm is used if the given number of points is within a considerable range. But if the number of points is considerably large, the standard approach is to use an R-tree or k-d tree.

## API, Front end and Visualization Layer

Project implementation has been divided into three parts

### Front end

User Interface of the application is implemented using Angular Framework consists of taking query inputs from the user and displaying the trips layer to the user based on the query inputs. Below is the screenshot of how the user interface looks like

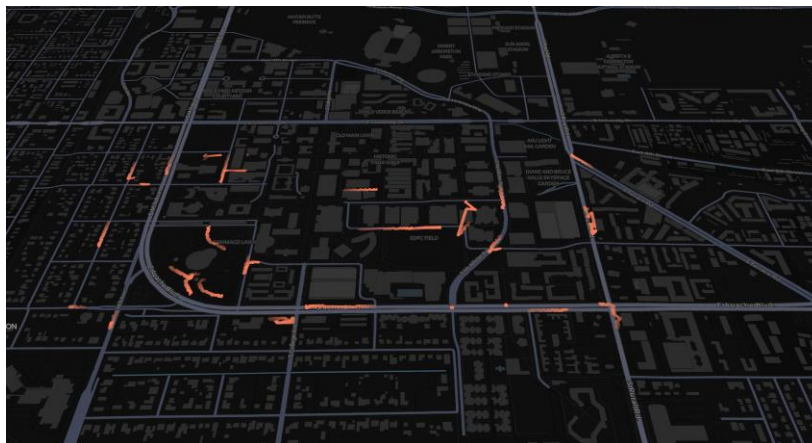
The screenshot displays the 'Trajectories' application interface. At the top, there is a 'Choose Input file' section with a 'Choose File' button and a 'No file chosen' status. Below this, the interface is divided into three main input sections:

- Enter Inputs for Spatio Temporal Query:** This section contains six input fields: 'Minimum Latitude', 'Minimum Longitude', 'Maximum Latitude', 'Maximum Longitude', 'Start Time', and 'End Time'.
- Enter Inputs for Spatial Range Query:** This section contains four input fields: 'Minimum Latitude', 'Minimum Longitude', 'Maximum Latitude', and 'Maximum Longitude'.
- Enter Inputs for KNN Query:** This section contains two input fields: 'Trajectory Id' and 'K Value'.

At the bottom right of the input section, there is a 'Display Trips layer' button.

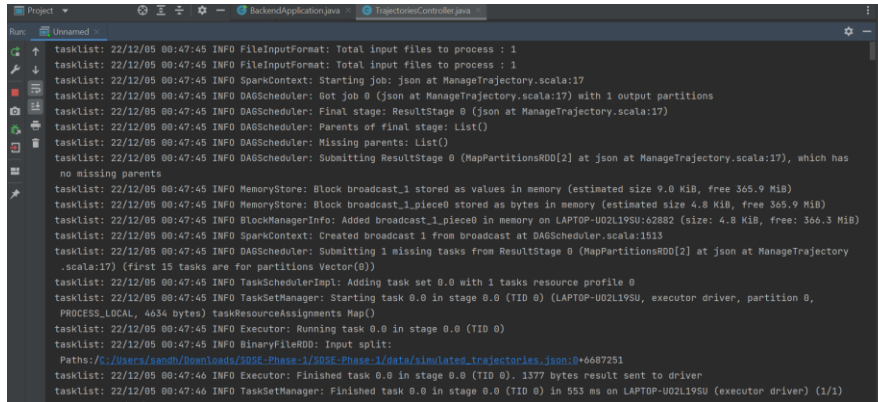
Spatial Range Query takes four inputs maximum and minimum latitudes, longitudes. Spatio Temporal Range Query takes six inputs maximum and minimum latitudes, longitudes, timestamp. KNN Query takes two inputs trajectory Id and k nearest neighbor value. The above screenshot is an example of how the inputs for queries can be entered

Once the user inputs data for any query and requests for displaying the trips layer, a user interface for that query is displayed to the user. An example of how the trips layer is displayed to the user is as below:



## Backend API:

API of the application is implemented using Spring Framework. Once the user clicks request to display the trips layer a REST api request is made to the Spring Framework and the Spring runs the spark submit by taking the parameters from the REST API which are enter in the front end, jar file generated by the spark application and generates files in the path mentioned. Once the spark-submit is run from Spring application, it displays the log in the console which can be used to track any errors or progress of the spark-submit. Below is an example of how the log is being displayed.

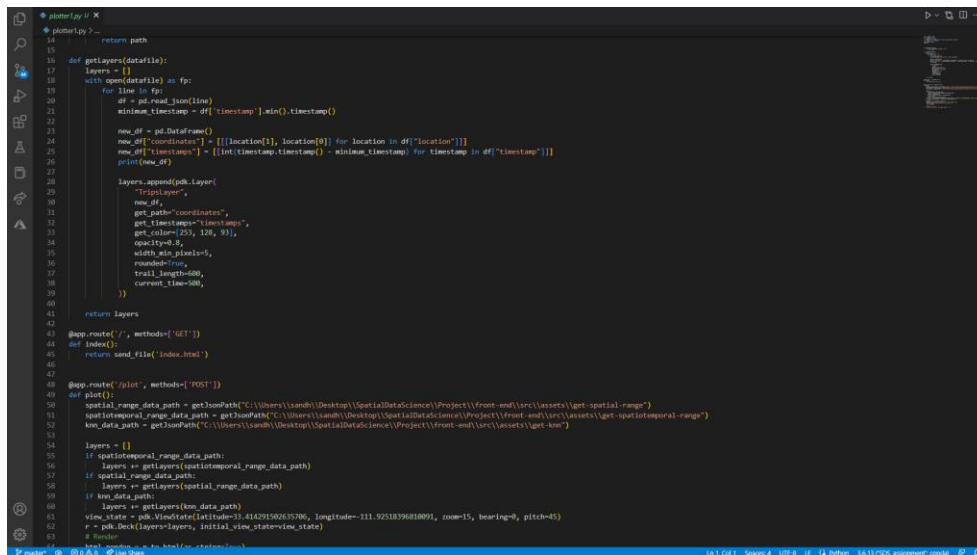


```
tasklist: 22/12/05 00:47:45 INFO FileInputFormat: Total input files to process : 1
tasklist: 22/12/05 00:47:45 INFO FileInputFormat: Total input files to process : 1
tasklist: 22/12/05 00:47:45 INFO SparkContext: Starting job: json at ManageTrajectory.scala:17
tasklist: 22/12/05 00:47:45 INFO DAGScheduler: Got job 0 (json at ManageTrajectory.scala:17) with 1 output partitions
tasklist: 22/12/05 00:47:45 INFO DAGScheduler: Final stage: ResultStage 0 (json at ManageTrajectory.scala:17)
tasklist: 22/12/05 00:47:45 INFO DAGScheduler: Parents of final stage: List()
tasklist: 22/12/05 00:47:45 INFO DAGScheduler: Missing parents: List()
tasklist: 22/12/05 00:47:45 INFO DAGScheduler: Submitting ResultStage 0 (MapPartitionsRDD[2] at json at ManageTrajectory.scala:17), which has no missing parents
tasklist: 22/12/05 00:47:45 INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated size 9.0 KiB, free 365.9 MiB)
tasklist: 22/12/05 00:47:45 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 4.8 KiB, free 365.9 MiB)
tasklist: 22/12/05 00:47:45 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on LAPTOP-U02L19SU:62882 (size: 4.8 KiB, free: 366.3 MiB)
tasklist: 22/12/05 00:47:45 INFO SparkContext: Created broadcast 1 from broadcast at DAGScheduler.scala:1513
tasklist: 22/12/05 00:47:45 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 0 (MapPartitionsRDD[2] at json at ManageTrajectory.scala:17) (first 15 tasks are for partitions Vector(0))
tasklist: 22/12/05 00:47:45 INFO TaskSchedulerImpl: Adding task set 0.0 with 1 tasks resource profile 0
tasklist: 22/12/05 00:47:45 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0) (LAPTOP-U02L19SU, executor driver, partition 0, PROCESS_LOCAL, 4634 bytes) taskResourceAssignments Map()
tasklist: 22/12/05 00:47:45 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
tasklist: 22/12/05 00:47:45 INFO BinaryFileRDD: Input split:
Paths: /Users/samdh/Downloads/SOSE_Phase-1/SOSE_Phase-1/data/simulated_trajectories_json/0+6687251
tasklist: 22/12/05 00:47:46 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1377 bytes result sent to driver
tasklist: 22/12/05 00:47:46 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 553 ms on LAPTOP-U02L19SU (executor driver) (1/1)
```

Once the REST API returns a success message, front end makes another API request to a server which is implemented in python to get the trips layer plot

## Visualization Layer

A Rest API is implemented in python which takes the files generated in previous step of spark submission and with the use of pydeck a plot is created by taking latitudes, longitudes and timestamps from the files. Below is an example of how trips layer plot is generated using pydeck. Configuration of trips layer such as width, trail\_length can be mentioned while creating the plot



```
def getLayers(datafile):
    layers = []
    with open(datafile) as fp:
        for line in fp:
            df = pd.read_json(line)
            minimum_timestamp = df['timestamp'].min().timestamp()

            new_df = pd.DataFrame()
            new_df['coordinates'] = [[location[1], location[0]] for location in df['location']]
            new_df['timestamp'] = [(int(timestamp.timestamp()) - minimum_timestamp) for timestamp in df['timestamp']]
            print(new_df)

            layers.append(pdk.Layer(
                "TripsLayer",
                new_df,
                get_path="coordinates",
                get_timestamps="timestamp",
                get_color=[255, 128, 93],
                opacity=0.5,
                width_min_pixels=5,
                rounded=True,
                trail_length=500,
                current_time=500,
            ))

    return layers

@app.route('/', methods=['GET'])
def index():
    return send_file('index.html')

@app.route('/plot', methods=['POST'])
def plot():
    spatial_range_data_path = getJsonPath(C:\Users\samdh\Desktop\SpatialData\source\Project\Front-end\src\assets\get-spatial-range)
    spatiotemporal_range_data_path = getJsonPath(C:\Users\samdh\Desktop\SpatialData\source\Project\Front-end\src\assets\get-spatiotemporal-range)
    km_data_path = getJsonPath(C:\Users\samdh\Desktop\SpatialData\source\Project\Front-end\src\assets\get-km)

    layers = []
    if spatiotemporal_range_data_path:
        layers = getLayers(spatiotemporal_range_data_path)
    if spatial_range_data_path:
        layers = getLayers(spatial_range_data_path)
    if km_data_path:
        layers = getLayers(km_data_path)

    view_state = pdk.ViewState(latitude=33.44291502635786, longitude=-111.92518396810891, zoom=15, bearing=0, pitch=45)
    e = pdk.Deck(layers=layers, initial_view_state=view_state)
    e.show()
    return e.to_html()
```

Response of the api is a html which can be embedded in an iframe in the front end to display the maps layer.