Documentation

We will discuss the assumptions made for the data, project setup, data modeling and the future improvements to be done.

1. Data Analysis

We are given data in three files, each one is analysed as below:-

routes.json

This is a big fat json file contaning an array of routes and data providers. Following are the assumptions made about the data in this file.

1. Each route here is and actual route in the geography as shared by the corresponding data providers.
2. A **Route** is divided into multiple segments. This would be an ordered set (json array).
3. A **Segment** is ordered collection of stops. Segment could have a unique identifier - Polyline. A Polyline in geometry means a continues line which is formed by several line segments.
4. So stops here are the endpoints of these line segments.
5. However we have some segments with length 0. I.e without any user movement for e.g.- change, setup, parking etc. Each such segment can be identified uniquely with the start and end stops. However for simplicity, I have taken such segments as

distinct.

6. **Stop** is a json object specifying the geo location and the timestamp of the presence for travelers in a segment for a route. The stop can be uniquely identified as gps coordinates. timestamp here refers that a user traveling under a given route will present at the given stop at this timestamp.
7. Routes have various properties, which I assume to be dependent on route type. For e.g. bike_sharing route will tell the numbers of bikes available on the route.
8. The Route.property has no definite structure and needs to be handled separately whenever a new route type is introduced.
9. Similarly stop has also a property field. However the dataset contains only null property for all the stops. So no further inference can be made about this field
10. Presence of timestamp field in the stop hints that the routes are valid only for the given timestamp. So the new route file has to be processed at least everyday, as the given all routes will be no longer valid for the next day. Assuming that all the timestamps are for the current date. So daily update frequency have been taken for this test.

route_queries.csv

This file includes search query by the user. I assume them to be the queries to search a valid route from origin gps coordinates to destination coordinates. *query_time* is the time for which we want to find the route for, *search_time* is the time user actually made the query. This assumption is made on the fact that *query_time* is never

less than *search_time*.

searches.csv

This file includes search query by the users, but comes from the different source and hence have different structure and less information. I assumed column *'at'* to be *query_time* as giving search_time will have no meaning for the query.

2. Project Design

This section covers setup of the project along with the design decisions.

Database Design

As specified in the task, Redshift has been selected as the Database store. Redshift is fork of PostgreSql-8, which has been modified to do massive parallel processing of queries(MPP DB).

1. We decided to use **Boyce–Codd** normal form for schema design of *Route* Data.
2. We have seven entities namely - Route, Segment, Stop, Taxi Company, and other three are entities for each route type i.e. Car Sharing Route, Bike Sharing Route and Taxi Routes.
3. We have two relationships as - Route-Segments, Segment-Stops,
4. Overall we have nine tables for each entity and relationships.

5. User Query data has been kept in a flat table as this data is historical and we expect to have much higher insert frequency as compared to Route data. Such historical data is used for OLAP systems and it is much easier and faster to run analytical queries on a flat table.
6. *'first_task/scripts/RedshiftCreateSchemas.sql'* contains the schema created on Redshift.
7. We have kept *'created_at'* column in every schema so that same can be used to garbage collect older data. If created as sort keys, this column can work as projection in Vertica DB.
8. Redshift's performance can be greatly improved by selecting the correct **distkey** and **sort key**
9. Rows with same distkey are stored in same node and threads can executed in parallel on different distkey.
10. column which either has high cardinality or which is used as key to join tables best performs as distkey. Keeping BCNF normal form helped us to meet both criteria
11. Inside a node data is kept sorted with sortkeys. **Compound sort keys** performs best when we know that the order of queries will never change. Otherwise it performs worse than having no key.
12. **Interleaved sortkeys** gives equal weight to each specified columns and give better all around performance.
13. We anticipated the type of queries we will have to selected interleaved sort key columns
14. For 'user_search_queries' we have not created any distkey as there was no fitting column. In that case redshift performs

better without any keys rather than having bad keys.

Project Design

1. Parsing of routes.json is done in **Java**.
2. We used Gson library for parsing Json. **Gson** is not the fastest but easy to use and manageable.
3. The file is processed in streaming model rather than deserializing completely in the memory using object model, this is to the fact that we could have much higher number of routes in real use case.
4. We have separate classes for the entities discussed in above section.
5. Main class reads from file. We could have more classes to read from different input streams like Kafka, Kinesis etc.
6. Since Route.Property structure depends on the route type. We need to provide concrete implementation every time a new route is added. There is no easier way of handling this
7. Route.Property parsed separately using custom deserializer for Gson.
8. Wrapper Classes are used instead of primitive types to identify the missing/null values from the zero values.
9. Java module is **mavenized**
10. Java code produces an output file marking each row for the corresponding schema.
11. An encapsulating bash script (*first_task/scripts/load_routes.sh*) filters rows in the output file and *put* the output to **S3**.

12. The script then *copy* the data from S3 to Redshift in the corresponding tables
13. User queries are parsed in a different shell script (*'first_task/scripts/load_queries.sh'*)
14. A python code parse the queries from one of the source(*'searches.csv'*) where as the other source(*'route_queries.csv'*) is parsed as in the bash script itself
15. Redshift copy commands are present at *'first_task/scripts/load_data.sql'* and *'first_task/scripts/load_query.sql'*. However we have committed a different versions of them which are without aws secret key.
16. Individual code contains more detailed comments.

Run Instructions

Two shells script are provided to parse and load route and query data. Scripts can be run by executing following commands in the project home folder.

```
#Loads route data in the respective tables in Redshift
#arg1: path of routes.json file
#arg2: path to generate intermediate output file
first_task/scripts/load_routes.sh routes.json first_task/
data/out

#Loads user query data in the user_search_query table in
Redshift
first_task/scripts/load_queries.sh
```

However to successfully run we will need additional files which are shared on alternate channels(e-mail)

1. Copy *.sql* files in *'first_task/scripts/* directory.
2. Copy *.s3cfg* in your `$HOME` folder.

3. Future Improvements

This project is submitted as part of the test. There is a huge scope of improvement in both system and database design. I have highlighted few of them below.

1. We need to setup a proper pipeline to get data from source to Redshift.
2. Routes data does not seem to need highly frequent update. We could setup a cronjob for the same. **Jenkins** is one such good cron manager.
3. Current user query processing is quite primitive. User Query data could have much higher scale. Recommended option is to parse data at receiving end and stream it to **Kinesis Firehose**. AWS have Redshift delivery Firehose, which will copy data at regular intervals(user controlled rate), while using S3 as intermediate storage.
4. We could create worker threads to offload the I/O.
5. Currently we are copying text file into the redshift. It is better to use binary schema based data encoding like *avro*. Which makes it easier to add/remove a column in the future and also

saves processing time in serialization and deserialization.

6. Redshift’s performance degrades after series of add/delete ops. We need to perform VACUUM and REINDEX jobs. We could setup a cron for such table maintenance while deciding the suitable *To Threshold Percentage*
7. We might need to write Python UDFs for Redshift when working with Geo location data, instead of doing work programatically.
8. Currently access has been given to all the IP addresses for the test DB. We should enable vpc only access for Redshift. IAM roles should be created and users should login with these IAM to create VPN to connect with Redshift