PG ROUTING

pgRouting is a feature of the PostgreSQL database management system which offers geospatial routing capabilities. pgRouting utilizes the PostGIS extension, enabling geographic functionality in PostgreSQL. It provides multiple routing algorithms to determine paths on maps, considering factors like distance, cost, and obstacles.

Below are a few essential characteristics and capabilities of pgRouting:

1. Methods for Determining Routes: pgRouting offers various routing algorithms like Dijkstra, A*, and Shooting-Star that are capable of addressing various routing scenarios like shortest route, driving distance, and others.

2. Personalized Cost Functions: Users have the option to create personalized cost functions that can impact routing choices by considering factors like travel duration, traffic situations, and road conditions.

3. Limitations on Turns: pgRouting includes support for turn restrictions, enabling a more accurate routing process by taking into account legal and physical limitations on specific maneuvers (e.g., no left turn).

4. Distance Calculation: The extension has the ability to determine distances for driving purposes starting from a specific point, aiding in defining service areas or catchment zones.

5. Isochrones: pgRouting is capable of producing isochrones, defining regions reachable within a specific time or distance from a given origin.

6. Dynamic Segmentation: With this function, roads can be divided into segments using different criteria, which is beneficial for thorough route analysis.

7. Integration with PostGIS: pgRouting utilizes the spatial functions and data types from PostGIS due to being developed on top of it, allowing for advanced geospatial analysis capabilities.

8. Free Availability: pgRouting is an open-source initiative, so it is freely accessible and can be altered and expanded by the public.

In general, pgRouting is a valuable tool for individuals working with geospatial data and requiring advanced routing features in a database.

Dijkstra Algorithm

Dijkstra's algorithm is a popular algorithm used to find the shortest paths between nodes in a graph, which may represent, for example, road networks. It is a classic example of a greedy algorithm.

❖ Steps of Dijkstra's Algorithm

1. Initialization:

   - Start with a source node.

   - Set the distance to the source node itself to 0 and to all other nodes to infinity.

   - Create a priority queue to store nodes based on their tentative distances from the source.

   - Mark all nodes as unvisited.

2. Main Loop:

   - While there are unvisited nodes:

     1. Extract the node with the smallest tentative distance from the priority queue. This node is considered the "current node."

     2. For the current node, consider all its unvisited neighbors. For each neighbor:

        - Calculate the tentative distance from the source node by summing the current node's distance and the edge weight to the neighbor.

        - If the calculated tentative distance of a neighboring node is less than its currently known distance, update the neighbor's distance and insert it into the priority queue.

     3. Mark the current node as visited.

3. Termination:

   - The algorithm terminates when all nodes have been visited. The shortest paths from the source node to all other nodes are then known.

❖ Characteristics of Dijkstra's Algorithm

Greedy Approach: It always chooses the next node with the smallest known distance.

Optimality: It guarantees finding the shortest path from the source node to all other nodes in the graph.

Non-negative Weights: The algorithm assumes that all edge weights are non-negative.

Complexity

- Time Complexity
- Space Complexity

❖ Usage with pgRouting

pgRouting extends PostgreSQL to provide geospatial routing capabilities, including Dijkstra's algorithm. You can utilize the `pgr_dijkstra` function to find the shortest path in a graph stored in a PostgreSQL database.

General Steps to Use pgRouting for Dijkstra's Algorithm

1. Prepare Your Data: Ensure your graph data is stored in a table with appropriate columns for source, target, and cost.

2. Run the Algorithm: Use the `pgr_dijkstra` function with a SQL query to extract the edges and specify the source and target nodes.

Input

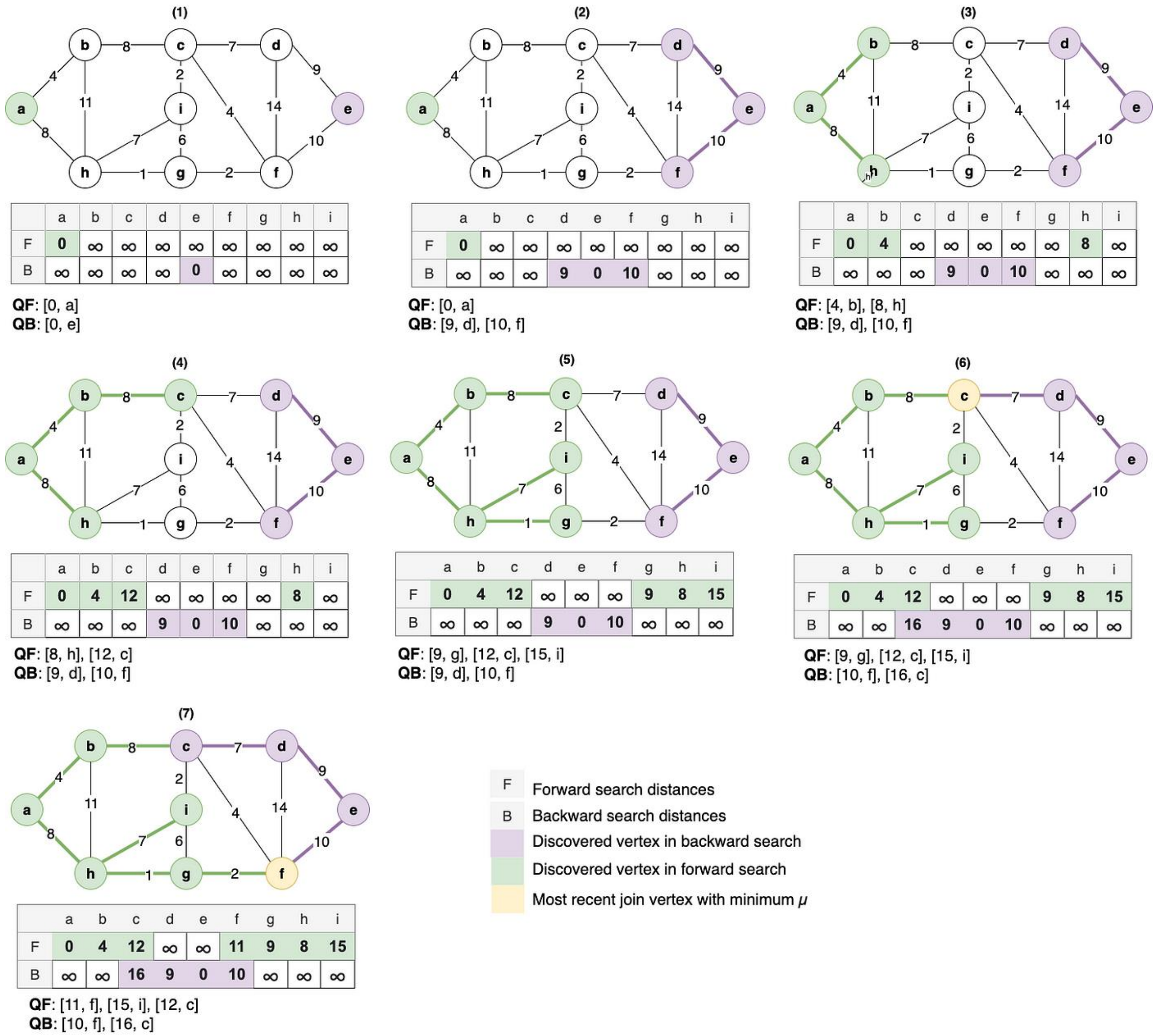
- A SQL query to select edge data (id, source, target, cost).

- Source node ID.

- Target node ID.

Output:

- A set of rows representing the shortest path, including details such as the sequence of nodes and the total cost.



(1)

	a	b	c	d	e	f	g	h	i
F	0	∞	∞	∞	∞	∞	∞	∞	∞
B	∞	∞	∞	∞	0	∞	∞	∞	∞

QF: [0, a]
QB: [0, e]

(2)

	a	b	c	d	e	f	g	h	i
F	0	∞	∞	∞	∞	∞	∞	∞	∞
B	∞	∞	∞	9	0	10	∞	∞	∞

QF: [0, a]
QB: [9, d], [10, f]

(3)

	a	b	c	d	e	f	g	h	i
F	0	4	∞	∞	∞	∞	∞	8	∞
B	∞	∞	∞	9	0	10	∞	∞	∞

QF: [4, b], [8, h]
QB: [9, d], [10, f]

(4)

	a	b	c	d	e	f	g	h	i
F	0	4	12	∞	∞	∞	∞	8	∞
B	∞	∞	∞	9	0	10	∞	∞	∞

QF: [8, h], [12, c]
QB: [9, d], [10, f]

(5)

	a	b	c	d	e	f	g	h	i
F	0	4	12	∞	∞	∞	9	8	15
B	∞	∞	∞	9	0	10	∞	∞	∞

QF: [9, g], [12, c], [15, i]
QB: [9, d], [10, f]

(6)

	a	b	c	d	e	f	g	h	i
F	0	4	12	∞	∞	∞	9	8	15
B	∞	∞	16	9	0	10	∞	∞	∞

QF: [9, g], [12, c], [15, i]
QB: [10, f], [16, c]

(7)

	a	b	c	d	e	f	g	h	i
F	0	4	12	∞	∞	11	9	8	15
B	∞	∞	16	9	0	10	∞	∞	∞

QF: [11, f], [15, i], [12, c]
QB: [10, f], [16, c]

F Forward search distances
B Backward search distances
Discovered vertex in backward search
Discovered vertex in forward search
Most recent join vertex with minimum μ

Figure 1: Djikstra Algorithm

Steps involved in Pgrouting for our mini-project:

Firstly, all the required files were downloaded and modified as listed or specified.

- osm2po is both, a converter and a routing engine. osm2po's converter parses OpenStreetMap's XML-Data and makes it routable.
- Other applications like java, PostGIS and QGIS were preinstalled.

Modification in Osm2po Config:

File Edit View

```
wtr.flagList = car, bike, foot, rail, ferry, poly

# Only convert ways with these flags. If finalMask is not set,
# any flag is valid. However, a way with clazz<=0 or flags<=0 is invalid
# and will be ignored.
# -------------------------------------------
# !!!      MAJOR CHANGE IN 5.2.72        !!!
# -------------------------------------------
# Use "," instead of "|" to list alternatives.
# Use "|" instead of "&" for binary OR-operations.
# From now on complex masks like "bike|foot&goodgrade" must be written
# as "bike,foot|goograde, which is consequent and more intuitive!

wtr.finalMask = car,foot,bike
#wtr.finalMask = car,bike

# In some rare cases it can be useful to prevent Ways from being
# processed by the segmenter and to exclude them from the road network.
# Converting streets and polygons in one pass is such a case, but this
# should be avoided. Declare the corresponding indicator flag here.

wtr.polyMask = poly

# osm2po needs to know the anchor-tag and some default values if not tagged.
# To avoid ambiguities configure a well chosen priority.
# Main-Tag definitions. Params 1-4:
# 1) priority
# 2) clazz identifier (1-127)
# 3) default speed in kmh
# 4) flags (optional) - allowed transportation types

wtr.tag.highway.motorway =       1,  11, 120, car
wtr.tag.highway.motorway_link =  1,  12, 30,  car
wtr.tag.highway.trunk =          1,  13, 90,  car
wtr.tag.highway.trunk_link =     1,  14, 30,  car
wtr.tag.highway.primary =        1,  15, 70,  car
wtr.tag.highway.primary_link =   1,  16, 30,  car
```

Ln 180, Col 26 | 55 of 19,156 characters

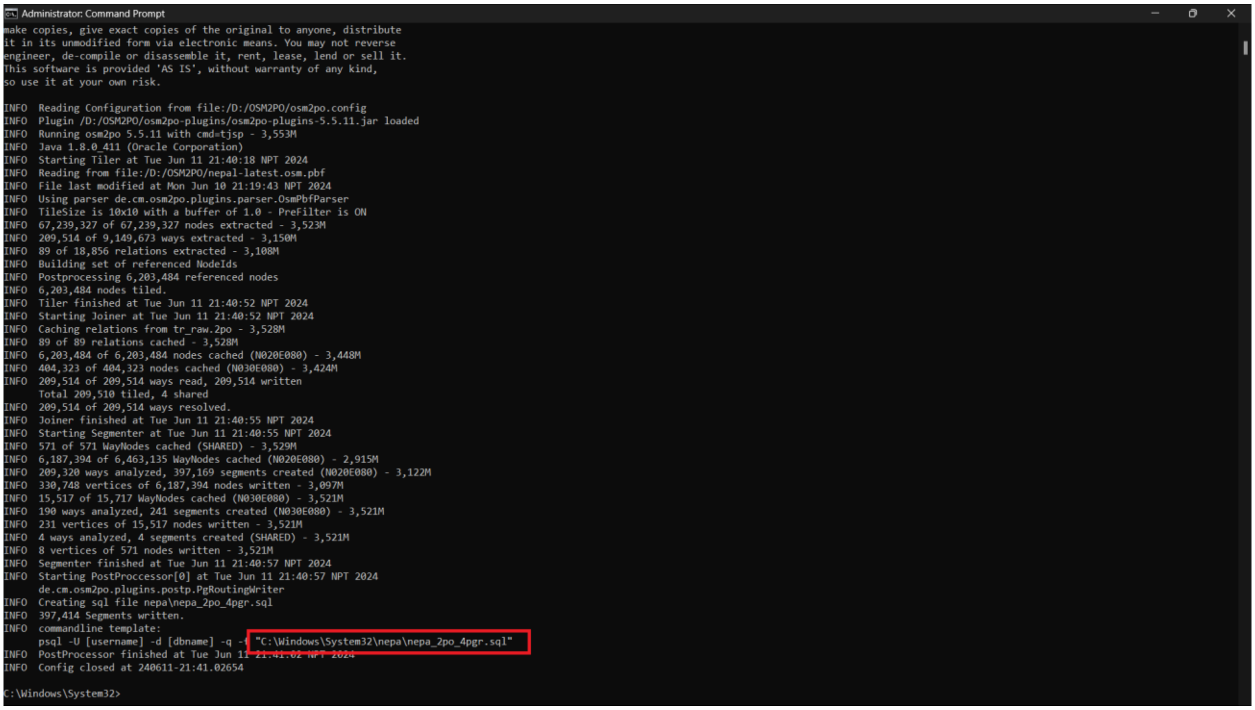
```
File    Edit    View
######################################################################
#
# POSTPROCESSORS
#
######################################################################

postp.0.class = de.cm.osm2po.plugins.postp.PgRoutingWriter
#postp.0.writeMultiLineStrings = true
#postp.1.class = de.cm.osm2po.plugins.postp.PgVertexWriter
#postp.2.class = de.cm.osm2po.plugins.postp.PgPolyWayWriter
#postp.3.class = de.cm.osm2po.plugins.postp.PgPolyRelWriter

#postp.4.class = de.cm.osm2po.postp.GeoExtensionBuilder
#postp.5.class = de.cm.osm2po.postp.BndExtensionBuilder
#postp.6.class = de.cm.osm2po.postp.MlgExtensionBuilder
#postp.6.id = 0
#postp.6.maxLevel = 3, 1.0

#postp.7.class = de.cm.osm2po.sd.postp.SdGraphBuilder

# Pg*Writer usually create sql files. Enable the following
# parameter to redirect them to stdout (console) e.g.:

#postp.1.pipeOut = true

# Tip 1:
# If you want this program to be one link in a transformation chain
# e.g. curl | bzcat | osm2po | psql
# you must set both, log.0.to=err and postp.0.pipeOut=true.
# log.0 is supposed to be a LogConsoleWriter.
# It is recommended to run curl, bzcat and psql in silent/quiet mode.
# Example (one line):
# curl -s -L http://download.geofabrik.de/europe/germany/hamburg-latest.osm.bz2 |
# bzcat -c |
# java -jar osm2po-core.jar prefix=hh postp.0.pipeOut=true log.0.to=err
#                           postp.0.class=de.cm.osm2po.plugins.postp.PgRoutingWriter |
# psql -q -U myuser -s -d mydb

Ln 330, Col 58    58 of 18,155 characters
```

After making the necessary modifications, we use **command prompt** to initialize our project.

The command used was:

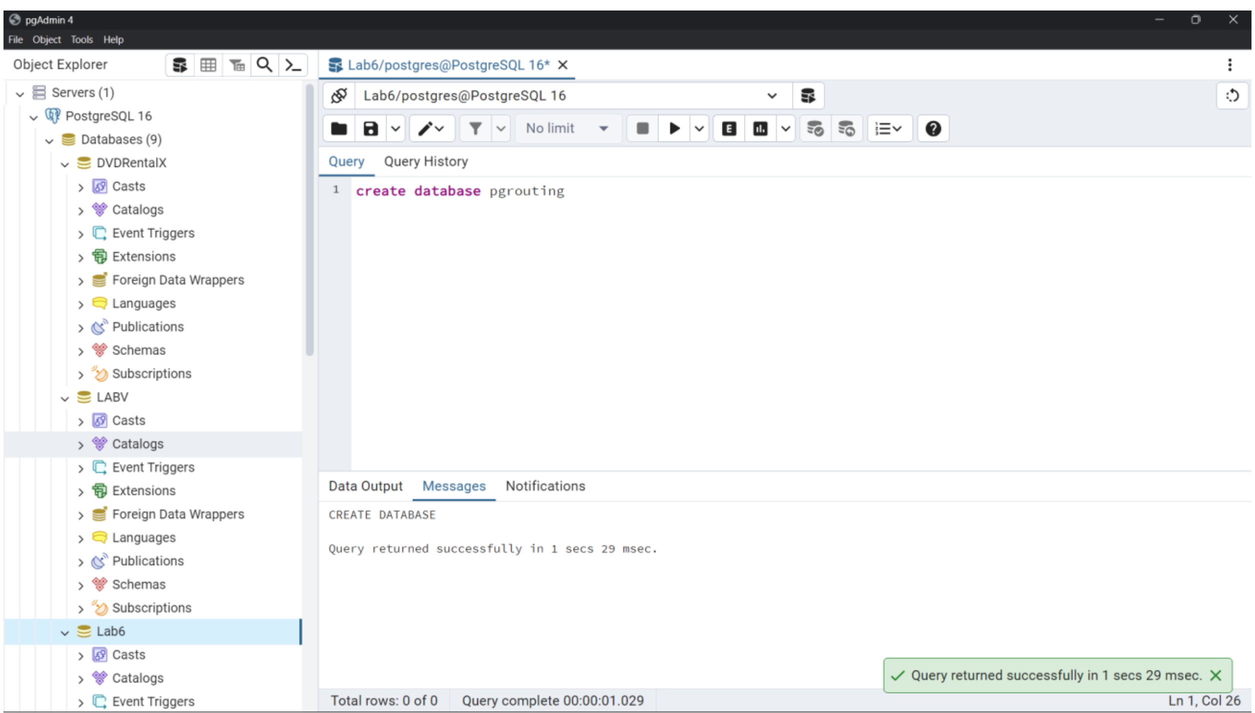
```
C:\Windows\System32>java -jar D:\OSM2PO\osm2po-core-5.5.11-signed.jar cmd=c prefix=nepa D:\OSM2PO\nepal-latest.osm.pbf
```

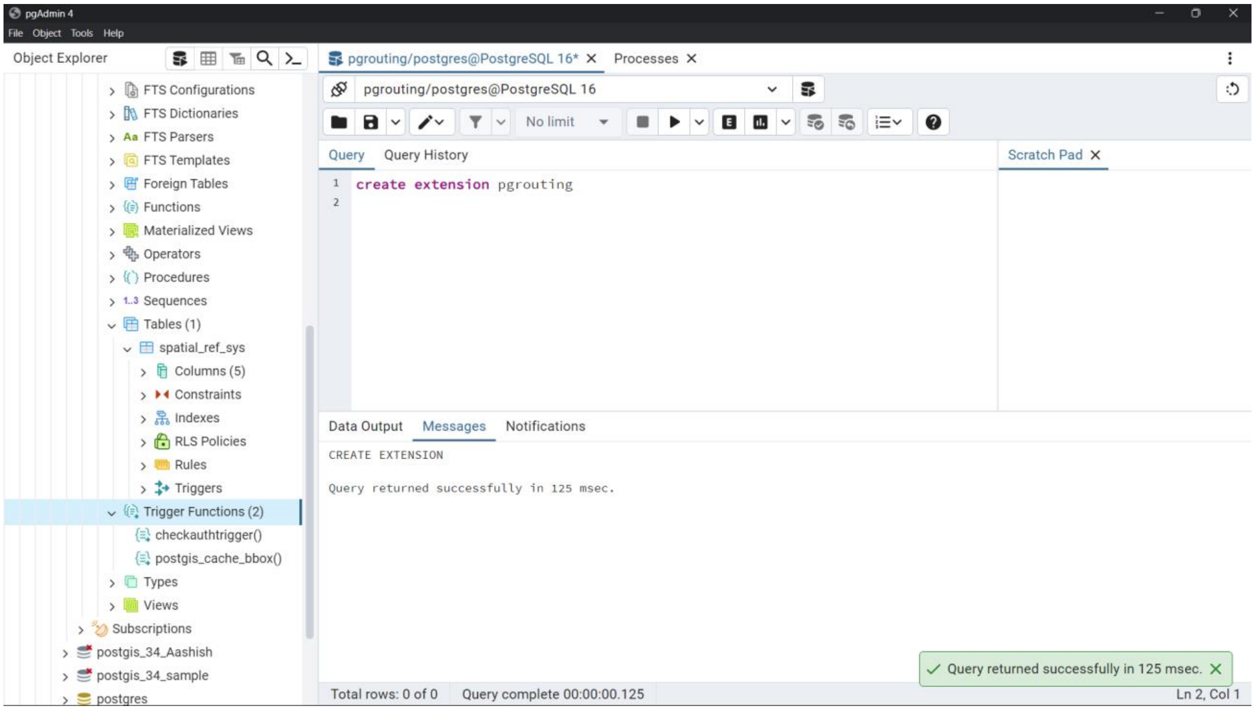
This query initiated a process as:

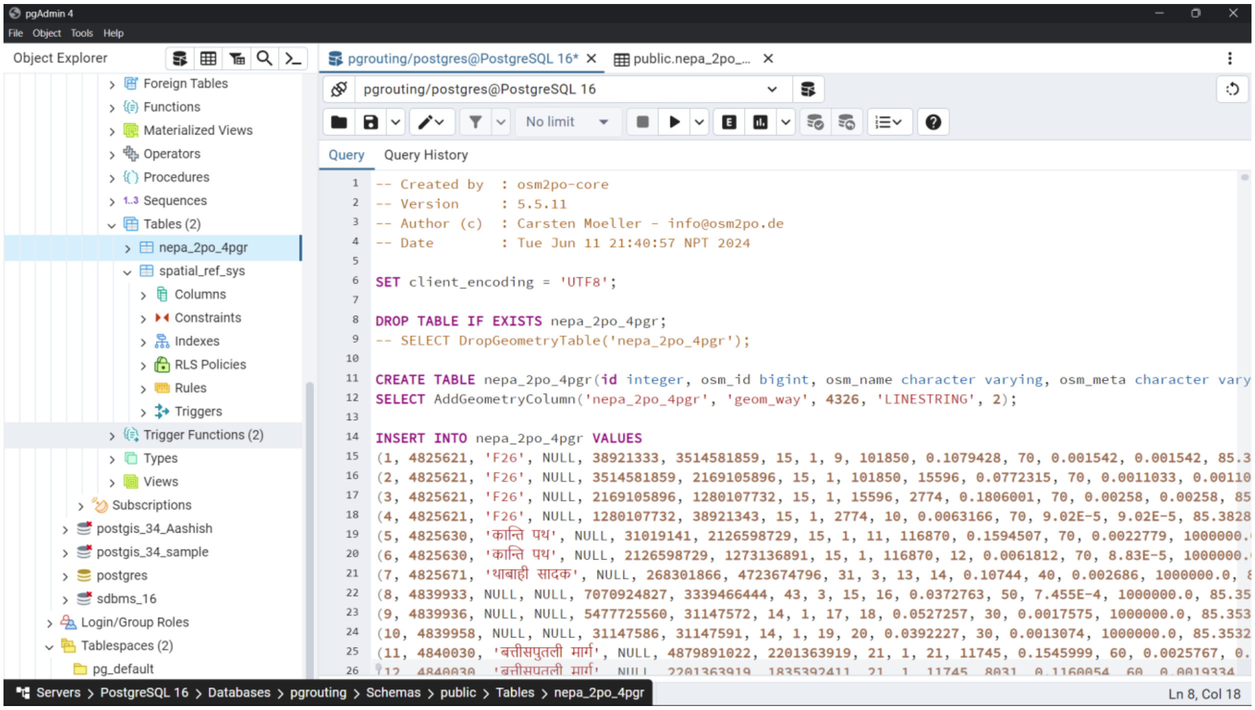


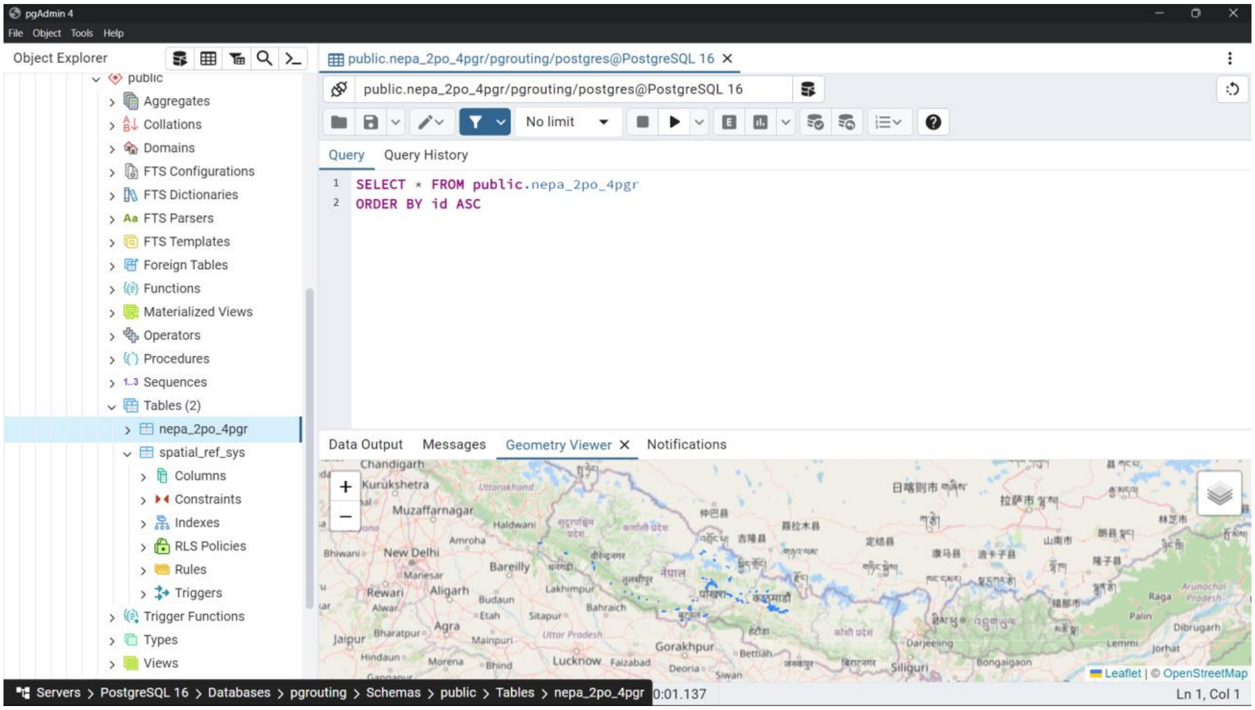
The red filter gives the final save location.

Once they have been saved and located, we use pgadmin to create database and load extensions pgrouting and postgis to create a table as:



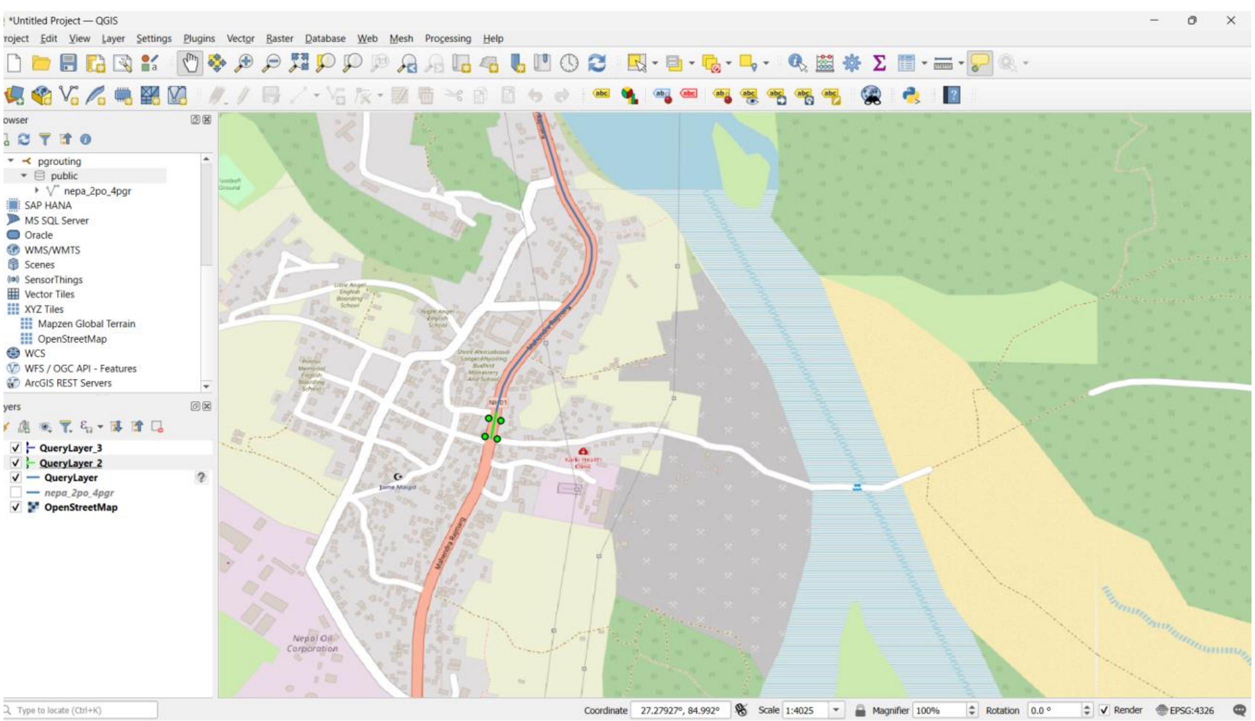




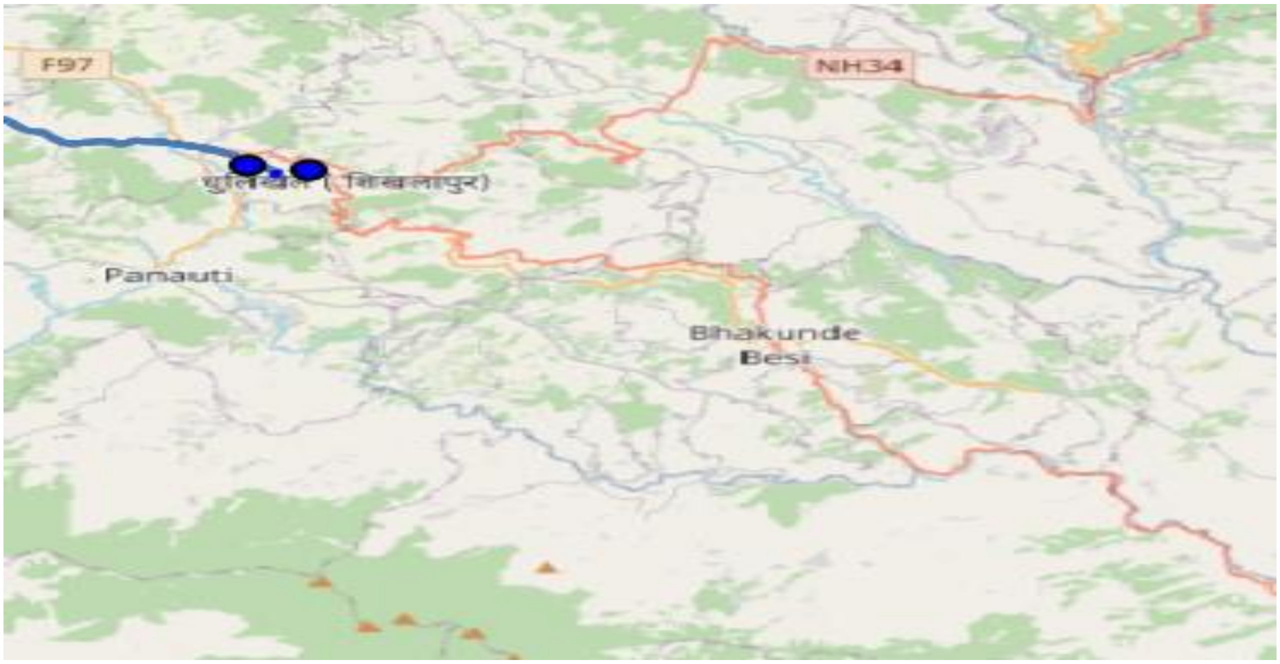


Once all the tables are created, we now perform pg routing by connecting it to QGIS.

The start point specified was Mahendra Highway:



The end point was specified upto the gate of Kathmandu University as:



The query used were:

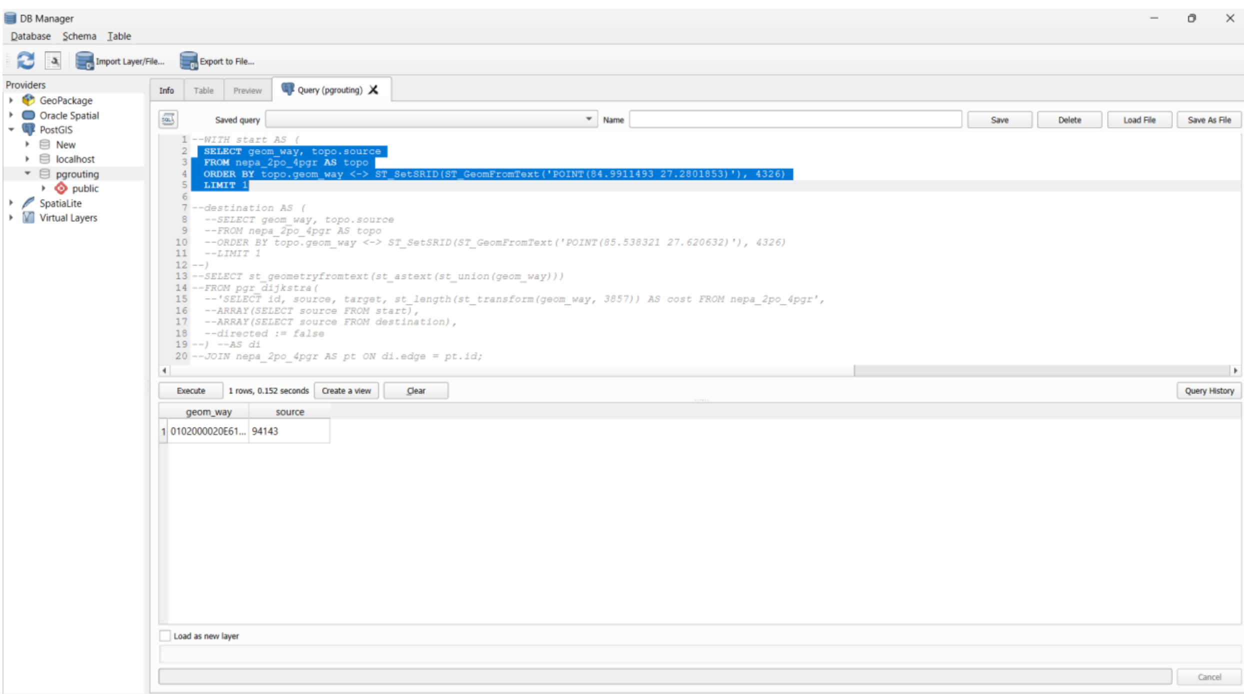


Figure 2: Source Point

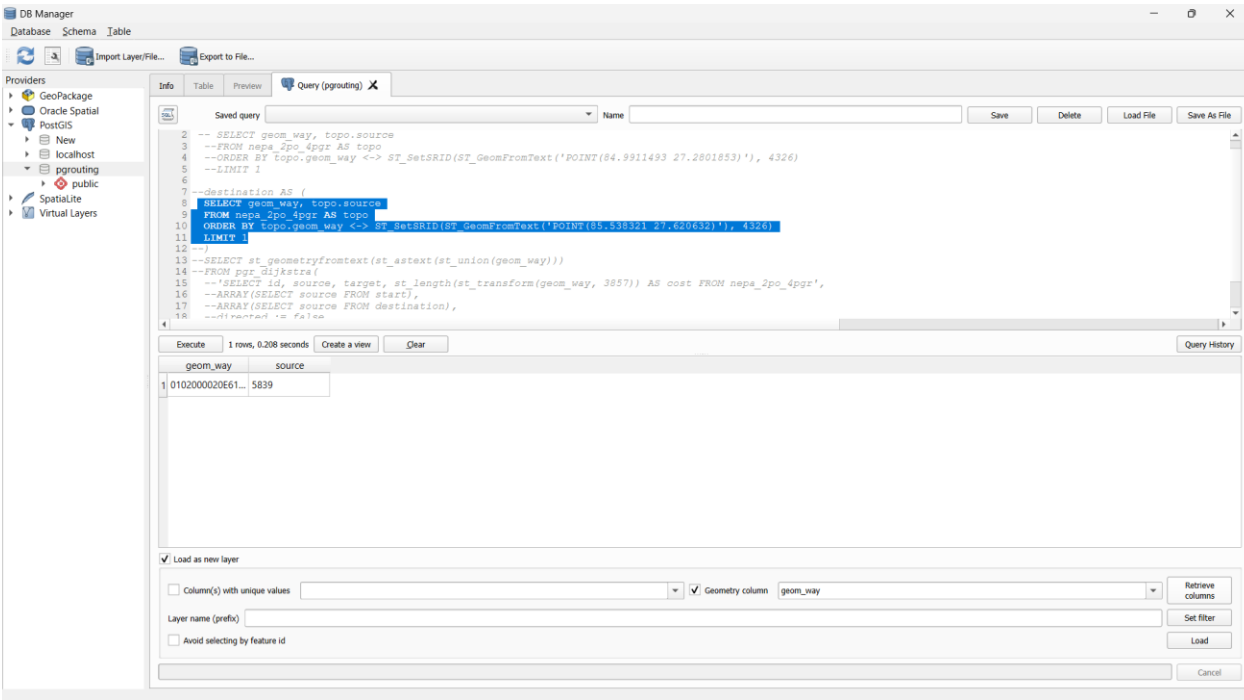


Figure 3: Destination Point

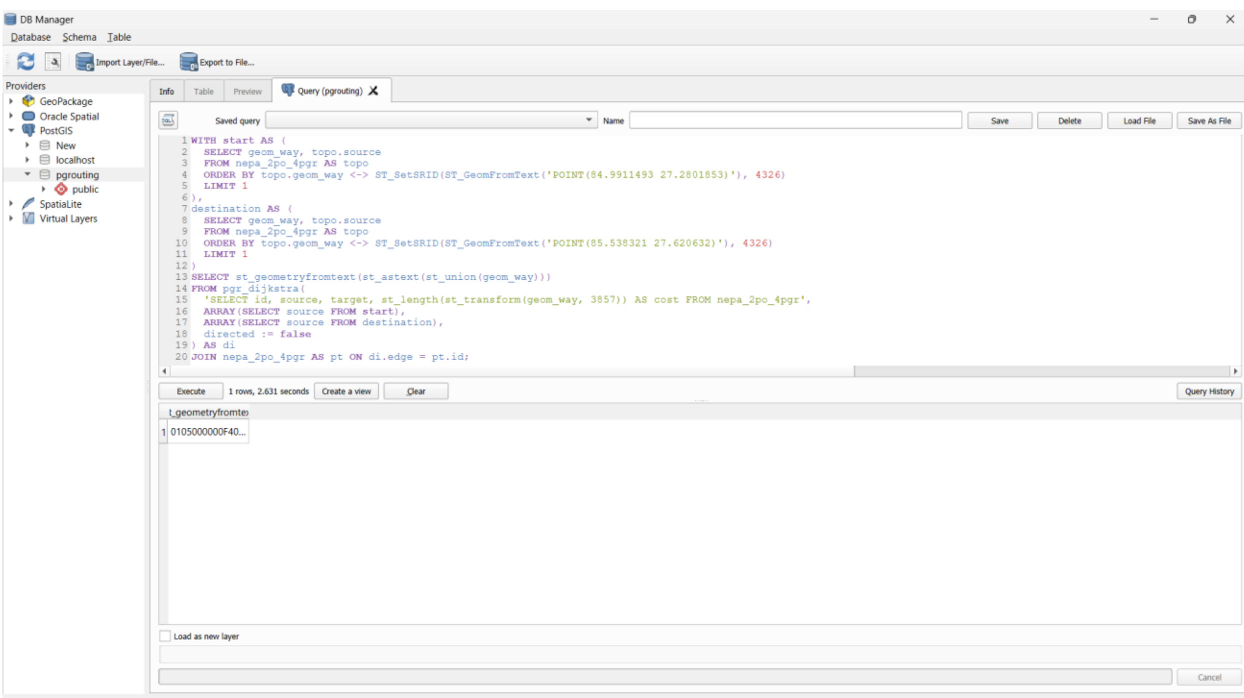


Figure 4: Main Query

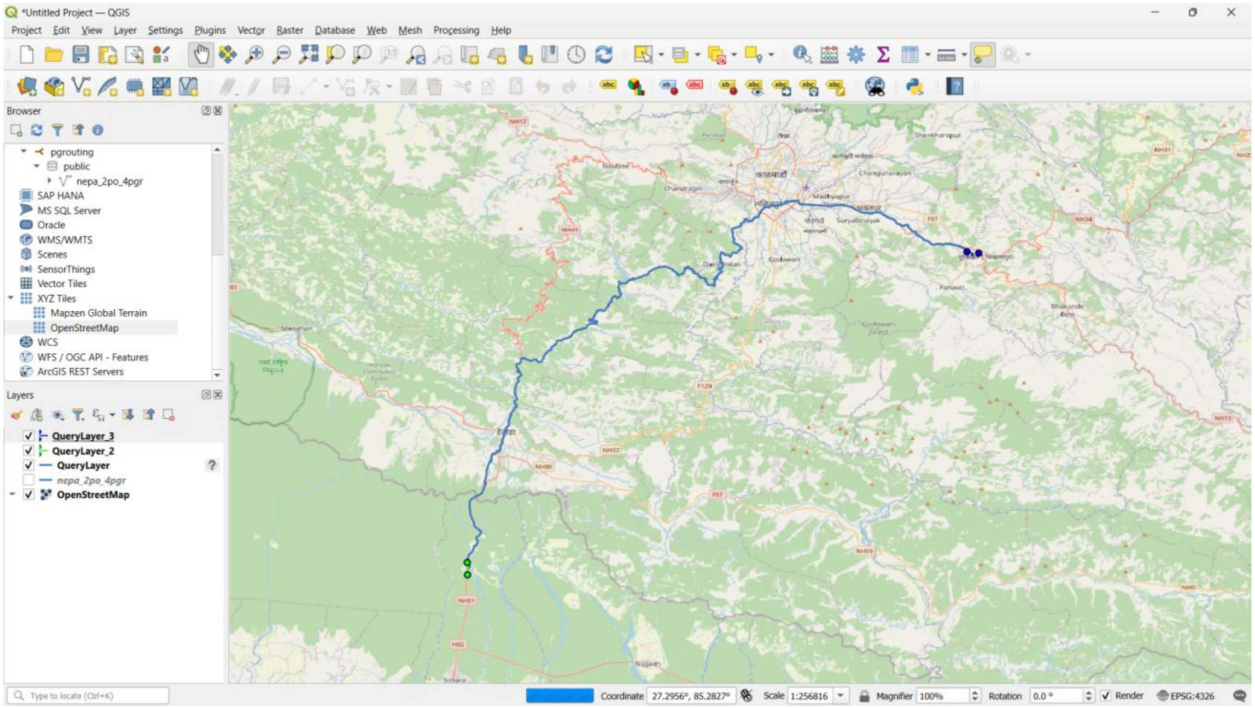


Figure 5: Shortest Route from Mahendra Rajmarga to Kathmandu University