Part 1: Basic Querying and Data Analysis

Over 100,000 random cities and their distances were generated in python using the faker library. An additional 30 city relationships were included in the file set, which are the cities of interest in comparing the query times between MongoDB and Neo4j.

1. List Roads from/to 'Atlanta' with Distances and Destinations:

MongoDB:

```
✓ #Task ···

Time taken to execute the query: 1.7097489833831787 seconds
From: Atlanta - To: East Robert - Distance: 941 km
From: Atlanta - To: Joshuahaven - Distance: 920 km
From: Atlanta - To: North Lori - Distance: 405 km
From: Atlanta - To: Adamland - Distance: 882 km
From: Atlanta - To: Shawnview - Distance: 200 km
From: Atlanta - To: Bowersshire - Distance: 701 km
From: Atlanta - To: Loganstad - Distance: 461 km
From: Atlanta - To: East Shelia - Distance: 230 km
From: Atlanta - To: North Heidi - Distance: 151 km
From: Atlanta - To: North Jason - Distance: 697 km
From: Atlanta - To: Richardborough - Distance: 67 km
From: Atlanta - To: Carpenterburgh - Distance: 824 km
From: Atlanta - To: New James - Distance: 812 km
From: Atlanta - To: East Elizabeth - Distance: 612 km
```

Neo4j:

```
✓ # 1 List Roads from/to 'Atlanta' with Distances and Destinations ···

Roads from/to Atlanta:
'Atlanta'       'Pricefort'     '200'
'Atlanta'       'Montogomery'   '150'
'Atlanta'       'West Ericton'  '550'
'Atlanta'       'Phillipsfurt'  '450'
'Atlanta'       'Hinesberg'     '300'
'Atlanta'       'North Arianastad'      '400'
Query Time: 0.038472890853881836 seconds
```

Generating all the roads from Atlanta, Neo4j performed the query faster than MongoDB. Neo4j performed the task in around 0.04 seconds, whereas MongoDB performed the task in1.7 seconds.

2. Find Roads Longer than 150 km, with Details:

MongoDB (left) and Neo4j (right):

```
✓ #2 Find Roads Longer than 150 km, with Details ···

Time taken to execute the query: 4.051257848739624 seconds
From: Jordanfurt – To: Port Colleen – Distance: 979 km
From: Brianside – To: Thompsonborough – Distance: 454 km
From: West Adamfort – To: Johnsonborough – Distance: 489 km
From: Masonstad – To: Michaelville – Distance: 474 km
From: Perezhaven – To: West Christinatown – Distance: 468 km
From: North Karenmouth – To: West Omarton – Distance: 471 km
From: Christineton – To: Lake Kevin – Distance: 360 km
From: Leeport – To: Lake Jonathan – Distance: 325 km
From: New Nathan – To: Matthewmouth – Distance: 885 km
From: North Juliefort – To: New Ronnie – Distance: 775 km
From: North David – To: West Jamiehaven – Distance: 773 km
From: West John – To: West Dakotaview – Distance: 167 km
From: Jessehaven – To: South Melinda – Distance: 912 km
From: Port Ronald – To: Greenville – Distance: 385 km
From: West Richard – To: South Randyborough – Distance: 260 km
From: Hernandezton – To: Amandaburgh – Distance: 583 km
From: Theresashire – To: Ricardochester – Distance: 259 km
From: Nicholasside – To: Jillianshire – Distance: 807 km
From: Lewisbury – To: East Natasha – Distance: 902 km
From: Victoriaton – To: South Amanda – Distance: 743 km
From: Hillmouth – To: South Heatherport – Distance: 753 km
From: New Jamesside – To: East Christinabury – Distance: 950 km
From: Deleonmouth – To: Stephensstad – Distance: 940 km
From: Michaelmouth – To: Lake Denise – Distance: 378 km
...
From: Robertfurt – To: Scottborough – Distance: 337 km
From: Mauricebury – To: Brownbury – Distance: 516 km
From: Lake Karl – To: Cruzmouth – Distance: 218 km
From: South Sherrymouth – To: Rogerstad – Distance: 270 km
```

```
✓ #2 Find Roads Longer than 150 km, with Details ···

Roads longer than 150 km:
'Michelleborough'       'East Kelseyshire'      '605'
'Michelleborough'       'Port Emilyport'        '965'
'Michelleborough'       'Port Justinport'       '918'
'Michelleborough'       'Port Jamesland'        '717'
'Michelleborough'       'East Desireetown'      '943'
'Michelleborough'       'East Valerie'  '806'
'Michelleborough'       'New Meganshire'        '454'
'Michelleborough'       'North Jacquelinestad'  '667'
'Michelleborough'       'Spencerchester'        '784'
'Michelleborough'       'Elizabethton'  '189'
'Michelleborough'       'Caitlinfurt'   '643'
'Michelleborough'       'Lake Dustinshire'      '595'
'Michelleborough'       'Lake Alexander'        '582'
'Michelleborough'       'Charlesstad'   '555'
'Michelleborough'       'Martinbury'    '910'
'Michelleborough'       'Carpenterfort' '324'
'Carpenterfort' 'East Don'      '959'
'Carpenterfort' 'Alanview'      '997'
'Carpenterfort' 'Stephensonville'       '426'
'Carpenterfort' 'Amybury'       '751'
'Barnesmouth'   'Lake Markland' '253'
'Barnesmouth'   'Lake Ashley'   '469'
'Barnesmouth'   'Susantown'     '658'
...
'Amman' 'Garzashire'    '410'
'Amman' 'Peterton'      '156'
'Amman' 'South Ashley'  '454'
Query Time: 3.2383840084075928 seconds
```

Overall MongoDB performed the query at 4.05 seconds, while Neo4j completed the query at around 3.23 seconds. One aspect is that MongoDB accesses the objects stored in JSON documents, which performs slower than Neo4j presetting the attributes for each node.

3. Total Road Length Connected to 'Frankfurt':

MongoDB:

```
✓ #3 Total Road Length Connected to 'Frankfurt' ···

Time taken to execute the query: 5.0743818283081055 seconds
Total road length connected to Frankfurt: 61925 km
```

Neo4j:

```
✓ #3 Total Road Length Connected to 'Frankfurt' ···

Total road length connected to Frankfurt:
 totalRoadLength
-----------------
            7584

Query Time: 0.046210050582885 74 seconds
```

MongoDB takes 5.07 seconds to complete the query whereas Neo4j is an order of magnitude faster in the query around 0.05 seconds. This seemingly large difference in execution is due to MongoDB requiring aggregation of all road sum distances. Unfortunately, there seems to be a discrepancy between the resulting aggregation of roads. We suspect that MongoDB is double counting the roads where Frankfurt is both the starting city, and the destination city. Neo4j produces a more accurate calculation of the total roads since there is directionality embedded in the database of roads with a destination of Frankfurt.

4. Determine Shortest and Longest Road from 'Amman':

MongoDB:

Shortest and Longest:

```
✓ #4 Determine Shortest and Longest Road from 'Amman' ···

Time taken to execute the query for shortest road: 0.007380008697509766 seconds
Time taken to execute the query for longest road: 0.007380008697509766 seconds
Shortest Road from 'Amman':
From: Amman - To: New Kathy - Distance: 64 km
Longest Road from 'Amman':
From: Amman - To: Lake Williamfort - Distance: 704 km
```

Neo4j:

Shortest and longest roads:

```
✓ #4 Determine Shortest Road from 'Amman' ···

Shortest road from Amman:
 fromCity | toCity     | shortestDistance
----------|------------|-----------------
 Amman    | North Sara |              103

Query Time: 0.04681897163391113 seconds
```

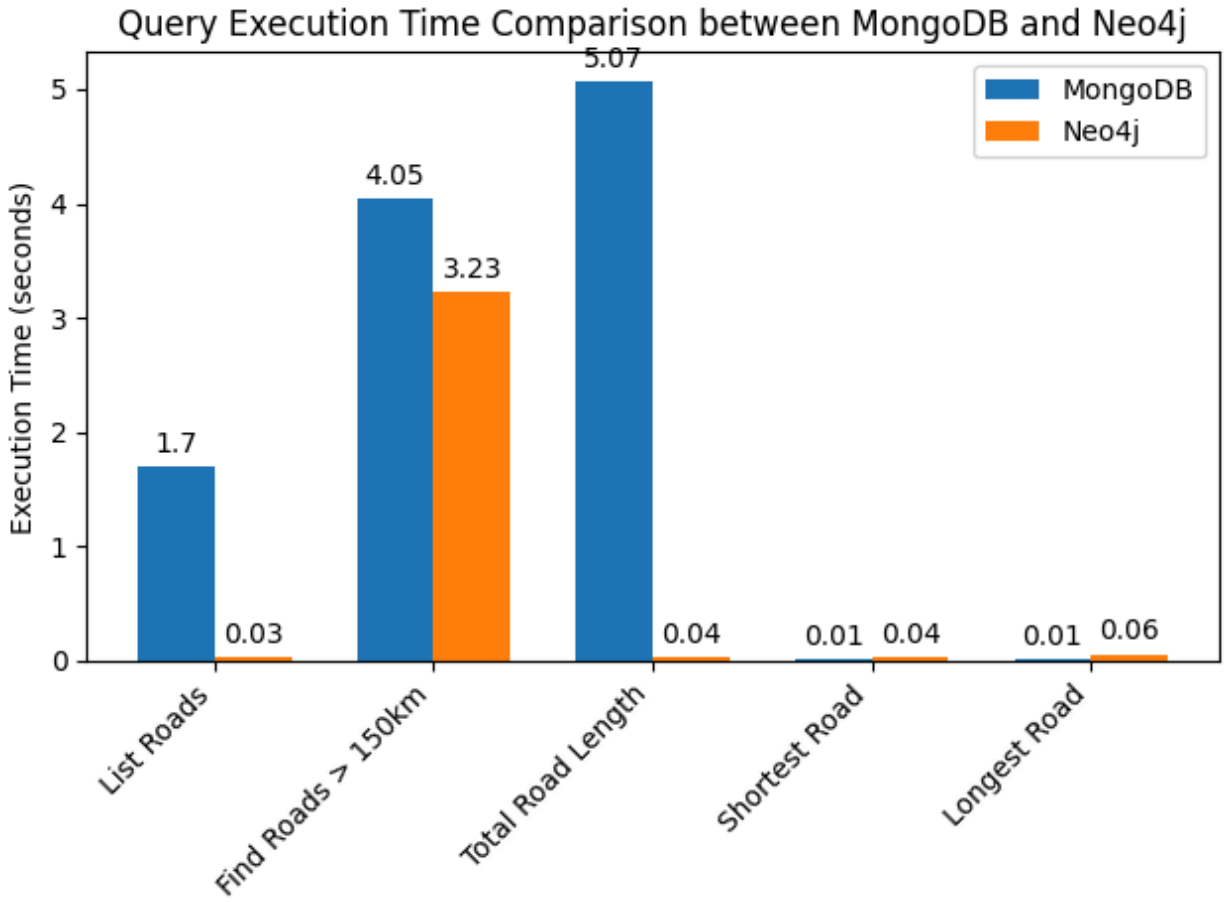
```
✓ # Determine Longest Road from 'Amman' ···

Longest road from Amman:
 fromCity | toCity     | longestDistance
----------|------------|----------------
 Amman    | Lake Jacob |             557

Query Time: 0.06295394897460938 seconds
```

Overall, MongoDB performed both queries for longest roads and shortest roads from Amman, Jordan around the same execution time as Neo4j, but given larger databases beyond 100,000 observations this processing time might change in Neo4j's favor.

Time Comparison Between MongoDB and Neo4j:

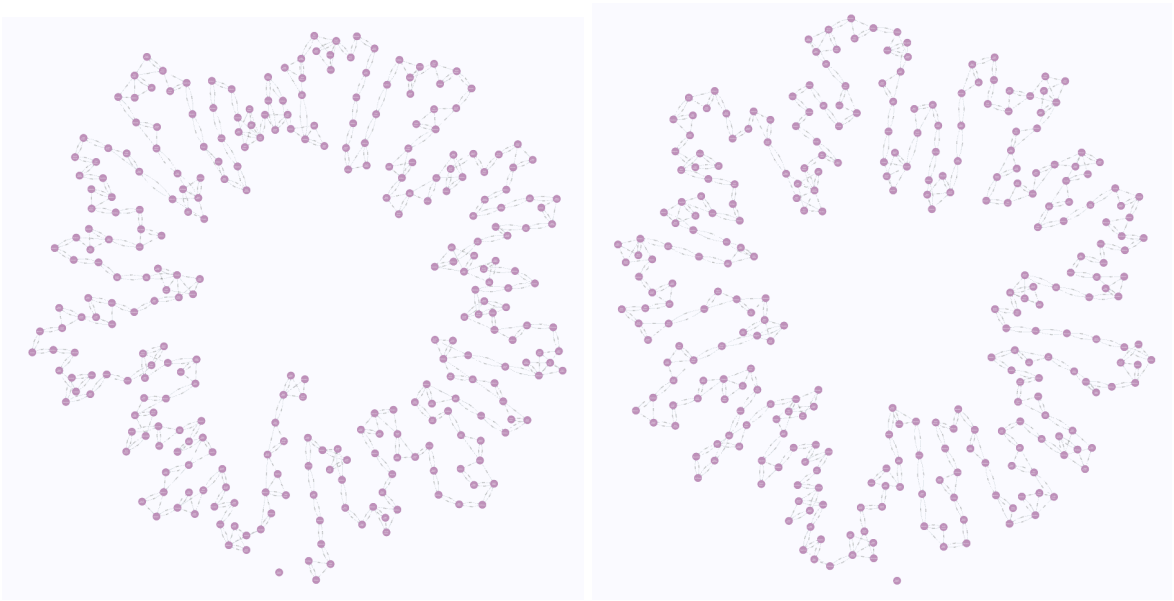


Overall our bar plot compares the execution time between MongoDB and Neo4j for all four queries. MongoDB performs the same executing speed at determining the shortest and longest roads in the dataset. Neo4j excels for nearly all queries, but is exceptionally fast during listing the total roads, the the total road lengths. Part of this speed is that Neo4j is directional in node relationships which offer a predetermined separating criteria in for example calculating road lengths to Frankfurt, whereas MongoDB would need an additional criteria to satisfy this condition. Neo4j takes more time to startup the database than MongoDB, but the processing time of queries are orders of magnitude quicker than the execution time for the same queries in MongoDB.

Part 2: Graph Algorithm in Neo4j

1. Depth First Search (DFS):

Task 1: DFS from 'Atlanta' (left) and ‘Frankfurt’ (right):



The Depth First Search (DFS) for Atlanta initially starts with a node, then searching it’s related neighbor node and further to find the key criteria of interest. Each node is identified whether it passed through the algorithm or not to ensure that nodes are not repeated. When there is a node that is terminal, the algorithm backtracks to the last known node to continue the search. In the above graphs, the node for Atlanta and Frankfurt is found via searching through the related nodes. By-product cycles are discovered in the path to the variable of interest. Cycles are discovered where the node relates directionally to another, and does not terminate. This reflects the intertwined nature of our network of nodes, while being linear to a degree. Another case is that our road networks do not have paths across large geographic distances. The relationships between cities simply connect with nearby cities. This partially explains the donut shaped paths of the nodes, where polarized nodes do not directly connect with each other.

2. First Search (BFS):

Task 2: BFS from 'Atlanta' (left), and ‘Frankfurt’ (right):

So the BFS, explores nodes based on the nearest depth of the algorithm tree priorities. The algorithm explores all nodes at the same depth which then connects them to centralized nodes if a node is higher up in the algorithm tree than another node. This produces clusters and centralizes the data. The crossing of node clusters in the above graphs indicate cities that are likely not connected by roads or any geographical connections. They represent impossible connections such as literally connecting Atlanta to Frankfurt from across the ocean, since both are respective node clusters for their region.

3. Visualization and Timing Analysis:

Task 5.1: DFS Timing and Visualization:

Atlanta:

```
✓ # DFS on Atlanta ···

([<Record traversalSequence=['Atlanta', 'Montogomery', 'Lake Jorgeberg', 'Hallmouth', '
DFS on Atlanta executed in 0.26135802268981934 seconds
```

Frankfurt:

```
✓ # DFS on Frankfurt ···

([<Record traversalSequence=['Frankfurt', 'Tyroneville', 'New Kimberg', 'West Josephfor
DFS on Frankfurt executed in 0.2294158935546875 seconds
```

Task 5.2: BFS Timing and Visualization:

Atlanta:

```
✓ # BFS on Atlanta ···

([<Record traversalSequence=['Atlanta', 'Port Heidi', 'Joshuaport', 'Phillipsfurt', 'Pr
BFS on Atlanta executed in 0.22919511795043945 seconds
```

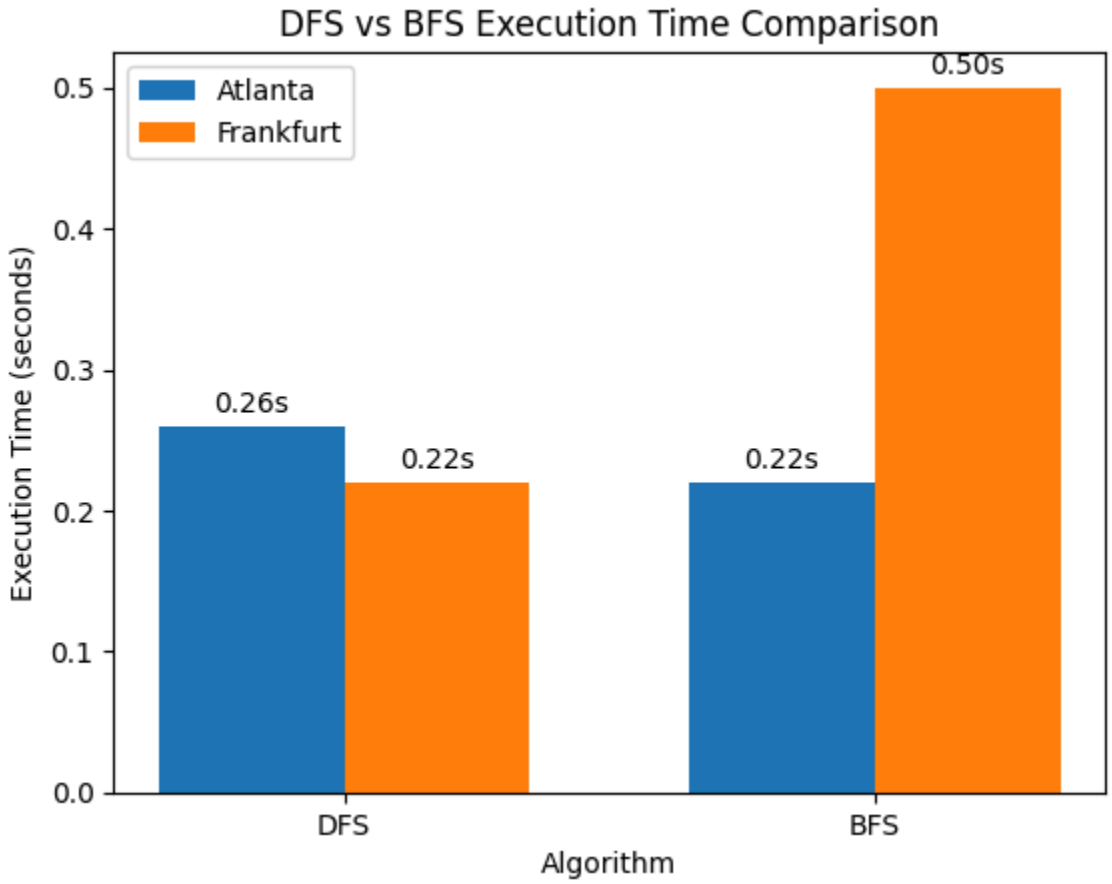
Frankfurt:

```
✓ # BFS on Frankfurt ···

([<Record traversalSequence=['Frankfurt', 'Robertfort', 'Port Andrew', 'Port Mary', 'Su
BFS on Frankfurt executed in 0.5094900131225586 seconds
```

Overall Depth First Search (DSF) has more variability in query time than Breadth First Search (BFS) which will reliably take the shortest amount of time. So although Atlanta and Frankfurt were queried in a similar amount of time using DFS, the starting node has an impact, as well as the amount of cycles in the database. The more cycles, and less terminal nodes the faster DBS will perform the cypher.

Task 5.3: Direct DFS and BFS Comparison:



Since Frankfurt had more nodes clustered as neighbors, the BFS took more time to query the request than DFS. Each neighbor at the same node level in the algorithm tree is verified before continuing to the next node. Atlanta and Frankfurt were queried around the same time for DFS due to interconnected road networks, and that our entire data structure is cyclical without terminal nodes. This allows DBS to pass through each neighbor quickly while minimizing the time to backtrack nodes.