# Project Statement for Milestone 3

Airline Search Engine

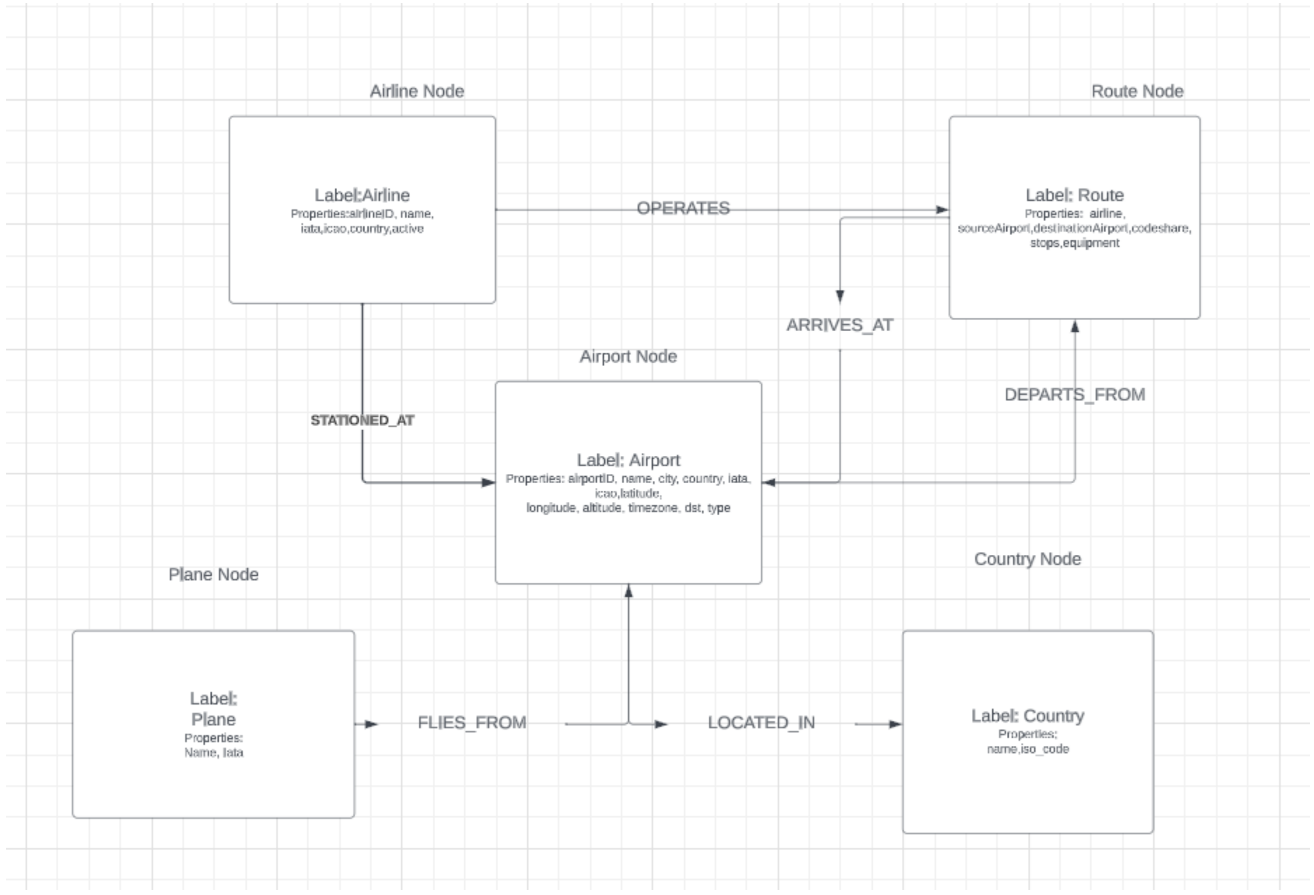
CPTS-415_BMW

Brian Joo, Noah Waxman, Mitchell Kolb

**Report Topics:**

The report should cover the following subtopics and answer the questions listed:

1. Database Schema:
   a. Describe the non-relational schema you have implemented for the reduced data set. Why is the schema an appropriate one for your project?


Airline Node
Label:Airline
Properties:airlineID, name, iata,icao,country,active
Route Node
Label: Route
Properties: airline, sourceAirport,destinationAirport,codeshare, stops,equipment
OPERATES
ARRIVES_AT
DEPARTS_FROM
STATIONED_AT
Airport Node
Label: Airport
Properties: airportID, name, city, country, iata, icao,latitude, longitude, altitude, timezone, dst, type
Plane Node
Label: Plane
Properties: Name, Iata
Country Node
Label: Country
Properties: name,iso_code
FLIES_FROM
LOCATED_IN

Here are the cypher statements for creating the nodes and relationships for our dataset:

```
LOAD CSV WITH HEADERS FROM 'file:///airlines.csv' AS row
CREATE (:Airline {airlineID: row.`airline ID`, name: row.name, iata: row.iata, icao: row.icao,
country: row.country, active: row.active});

LOAD CSV WITH HEADERS FROM 'file:///airports.csv' AS row
CREATE (:Airport {airportID: row.`airport ID`, name: row.name, city: row.city, country:
row.country, iata: row.iata, icao: row.icao, latitude: toFloat(row.latitude), longitude:
toFloat(row.longitude), altitude: toInteger(row.altitude), timezone: row.timezone, dst: row.dst,
type: row.type});

LOAD CSV WITH HEADERS FROM 'file:///routes.csv' AS row
CREATE (:Route {airline: row.airline, sourceAirport: row.`source airport`, destinationAirport:
row.`destination airport`, codeshare: row.codeshare, stops: toInteger(row.stops), equipment:
row.equipment});

LOAD CSV WITH HEADERS FROM 'file:///countries.csv' AS row
CREATE (:Country {name: row.name, iso_code: row.iso_code});

LOAD CSV WITH HEADERS FROM 'file:///planes.csv' AS row
CREATE (:Plane {name: row.name, iata: row.iata});

MATCH (a:Airline), (r:Route)
WHERE a.iata = r.airline
CREATE (a)-[:OPERATES]->(r);

MATCH (r:Route), (s:Airport)
WHERE r.sourceAirport = s.iata
CREATE (r)-[:DEPARTS_FROM]->(s);

MATCH (r:Route), (d:Airport)
WHERE r.destinationAirport = d.iata
CREATE (r)-[:ARRIVES_AT]->(d);
```

We think that this non-relational schema that we have described above is appropriate because we are using neo4j and it is a graph based system. Our schema has two main parts, Nodes and Relationships unlike relational tabular models that are based around tables and primary keys. Our nodes are airlines, airports, routes, countries, and planes. All these nodes have properties associated with them that hold data values like airlineID or Country. Our relationships are “operates”, “departs_from”, and “arrives_at”. We named these relationships from what we thought linguistically they are according to the data. For example airlines do actually operate on routes that are set up and planes on routes arrive at airports that are traveling. Our schema shown above is appropriate for our project because it effectively models the complex relationships between different entities of flight travel.

In reference to the graph picture shown first each node is shown inside the blocks with its respective properties and label. The name of each node hovers on top. The relationships are defined clearly between the arrows that connect the two related nodes. As the project requirements are to implement a search engine that can query and retrieve information from the given data sets, we thought it best to have these nodes and relationships.

2. Data Ingestion and Query:
    a. Describe how you ingested the reduced dataset to the database. Also, describe how you validated the ingestion step, perhaps through aggregation queries.

       The data was first parsed, cleaned, and sorted into its own CSV file as per the last milestone. The team validated the ingestion step through these queries such as the following:

       1) 1st part of ingesting starts with importing data from the files and creating nodes

```
LOAD CSV WITH HEADERS FROM 'file:///airlines.csv' AS row
CREATE (:Airline {airlineID: row.`airline ID`, name: row.name, iata: row.iata, icao: row.icao,
country: row.country, active: row.active});
```

```
LOAD CSV WITH HEADERS FROM 'file:///airports.csv' AS row
CREATE (:Airport {airportID: row.`airport ID`, name: row.name, city: row.city, country:
row.country, iata: row.iata, icao: row.icao, latitude: toFloat(row.latitude), longitude:
toFloat(row.longitude), altitude: toInteger(row.altitude), timezone: row.timezone, dst: row.dst,
type: row.type});
```

```
LOAD CSV WITH HEADERS FROM 'file:///routes.csv' AS row
CREATE (:Route {airline: row.airline, sourceAirport: row.`source airport`, destinationAirport:
row.`destination airport`, codeshare: row.codeshare, stops: toInteger(row.stops), equipment:
row.equipment});
```

```
LOAD CSV WITH HEADERS FROM 'file:///countries.csv' AS row
CREATE (:Country {name: row.name, iso_code: row.iso_code});
```

```
LOAD CSV WITH HEADERS FROM 'file:///planes.csv' AS row
CREATE (:Plane {name: row.name, iata: row.iata});
```

2) 2nd part of ingesting starts with creating relationships between nodes

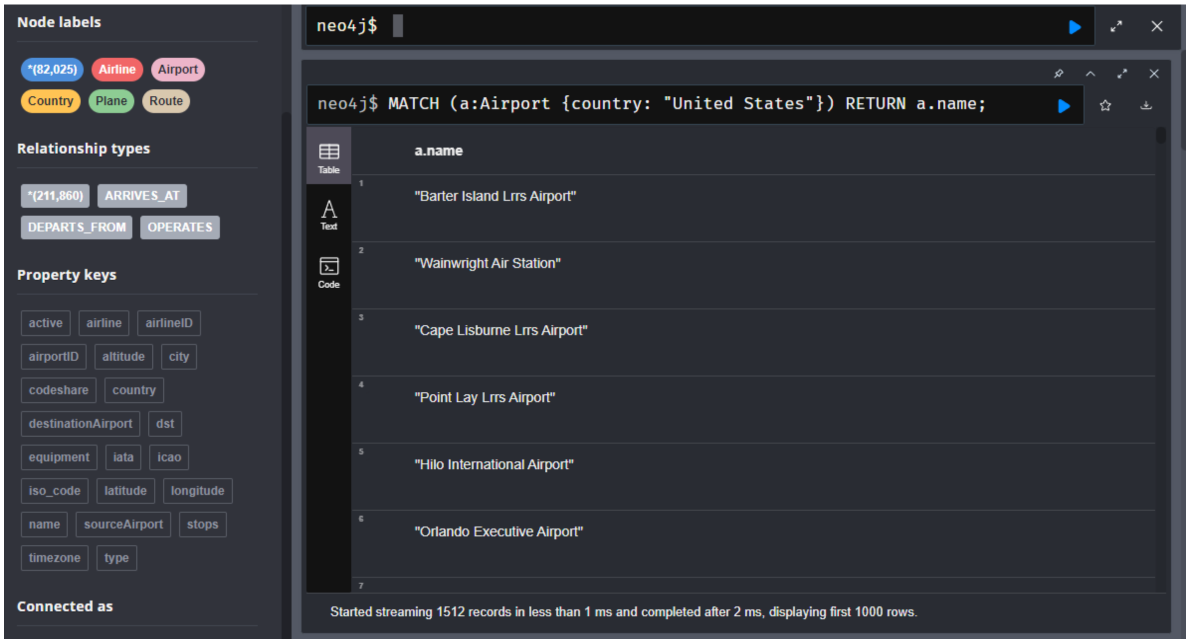
```
MATCH (a:Airline), (r:Route)
WHERE a.iata = r.airline
CREATE (a)-[:OPERATES]->(r);
```

```
MATCH (r:Route), (s:Airport)
WHERE r.sourceAirport = s.iata
CREATE (r)-[:DEPARTS_FROM]->(s);
```

```
MATCH (r:Route), (d:Airport)
WHERE r.destinationAirport = d.iata
CREATE (r)-[:ARRIVES_AT]->(d);
```

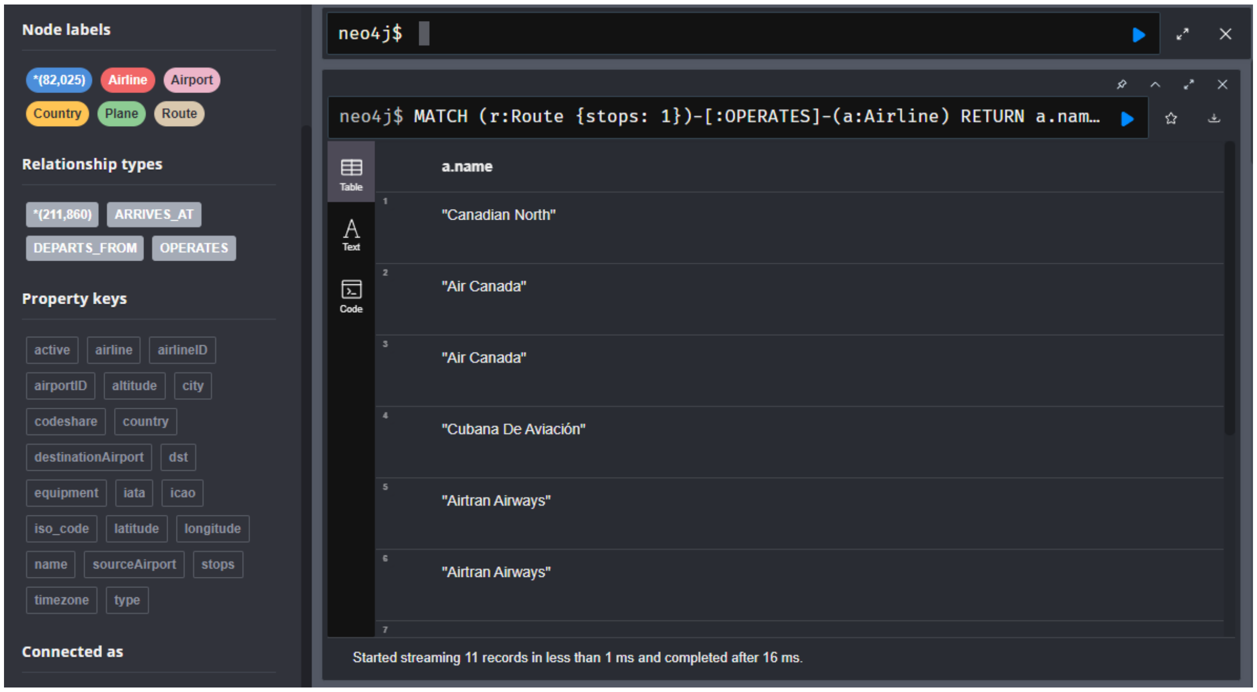
3) We validated the ingestion step using queries that return data using the labels we created from the data that we ingested. Here are some examples along with the results of them.

MATCH (a:Airport {country: "United States"}) RETURN a.name;



```
neo4j$

neo4j$ MATCH (a:Airport {country: "United States"}) RETURN a.name;

a.name
1  "Barter Island Lrrs Airport"
2  "Wainwright Air Station"
3  "Cape Lisburne Lrrs Airport"
4  "Point Lay Lrrs Airport"
5  "Hilo International Airport"
6  "Orlando Executive Airport"
7
Started streaming 1512 records in less than 1 ms and completed after 2 ms, displaying first 1000 rows.
```

MATCH (r:Route {stops: 1})-[:OPERATES]-(a:Airline) RETURN a.name;



```
neo4j$ MATCH (r:Route {stops: 1})-[:OPERATES]-(a:Airline) RETURN a.nam…
```

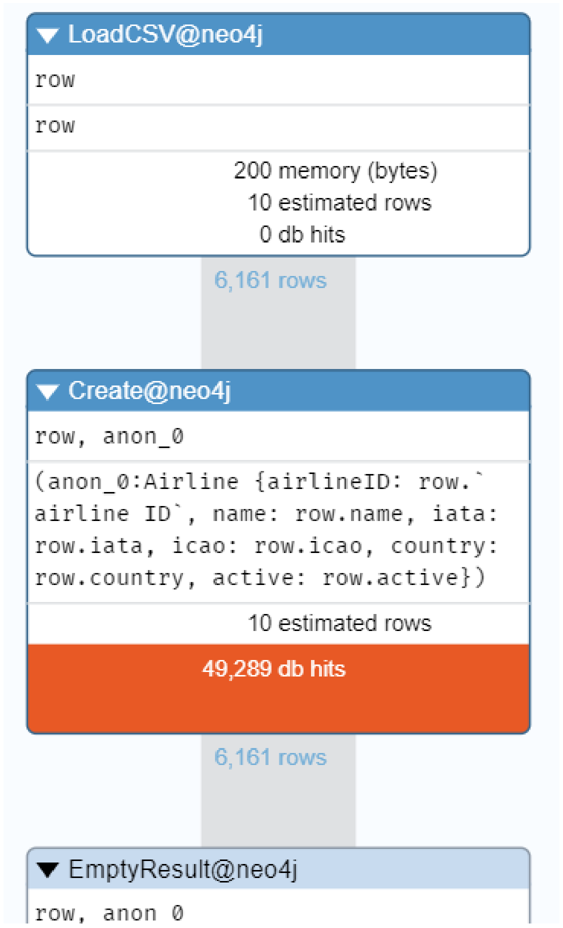
```
   a.name
1  "Canadian North"
2  "Air Canada"
3  "Air Canada"
4  "Cubana De Aviación"
5  "Airtran Airways"
6  "Airtran Airways"
7
```

```
Started streaming 11 records in less than 1 ms and completed after 16 ms.
```

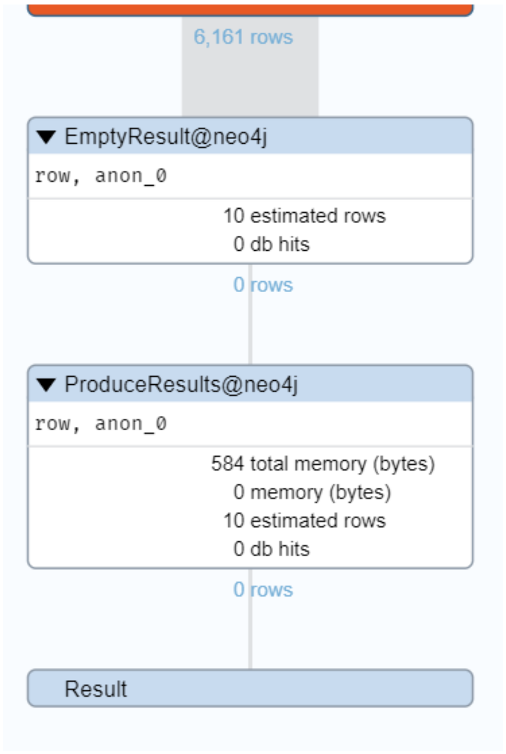
b. Provide performance results of your data ingestion and query operations. How would it change (scale) with increase in dataset size.

The performance results of the data ingestion is able to be viewed in neo4j using the cypher command "PROFILE" or "EXPLAIN". Here is the result of the first load and create statement.

```
PROFILE LOAD CSV WITH HEADERS FROM 'file:///airlines.csv' AS row
CREATE (:Airline {airlineID: row.`airline ID`, name: row.name, iata: row.iata,
icao: row.icao, country: row.country, active: row.active});
```



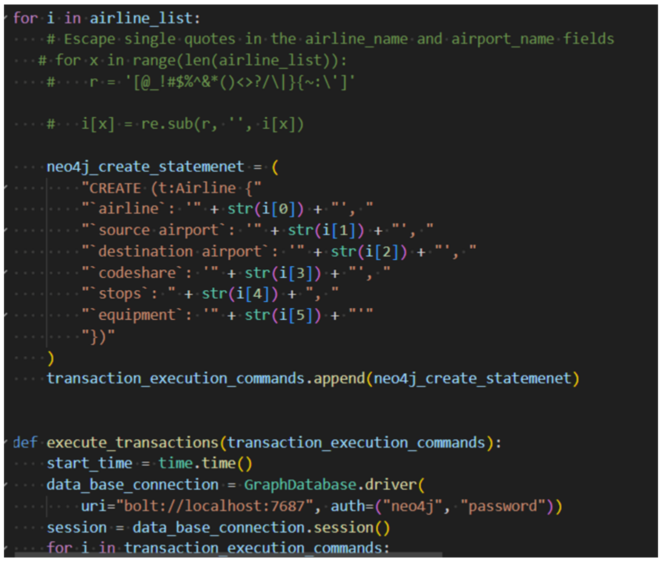
```
▼ LoadCSV@neo4j
row
row
200 memory (bytes)
10 estimated rows
0 db hits
6,161 rows
▼ Create@neo4j
row, anon_0
(anon_0:Airline {airlineID: row.`
airline ID`, name: row.name, iata:
row.iata, icao: row.icao, country:
row.country, active: row.active})
10 estimated rows
49,289 db hits
6,161 rows
▼ EmptyResult@neo4j
row, anon_0
```



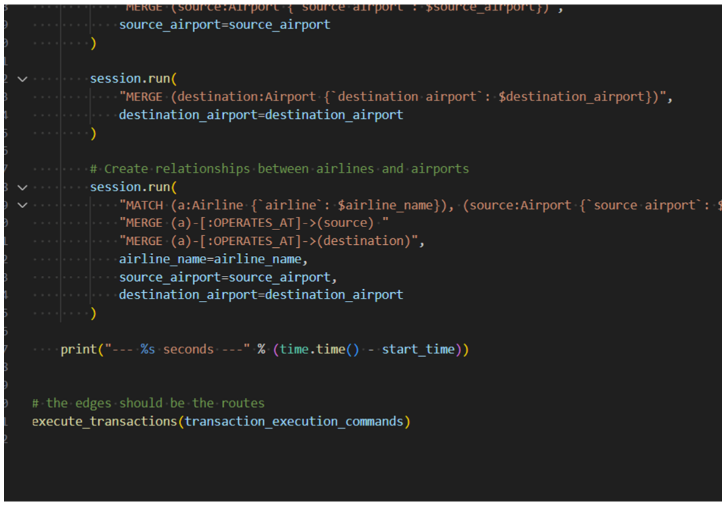
```
6,161 rows
▼ EmptyResult@neo4j
row, anon_0
10 estimated rows
0 db hits
0 rows
▼ ProduceResults@neo4j
row, anon_0
584 total memory (bytes)
0 memory (bytes)
10 estimated rows
0 db hits
0 rows
Result
```

When ingesting the data from the airlines.csv file there are 6161 line items. To perform this command neo4j used about 584 bytes of memory. The documentation talks about how the neo4j underlying architecture is designed to handle a large quantity of data at a given time. THe “LOAD CSV” command is rated to be able to confidently store up to 10 million lines of data while using minimal memory safely. Our dataset won’t even reach a 20th of that maximum size limit. So that means we have plenty of space to use before any we have to worry about potentially using lots of resources. But in the scenario where we need to ingest more than 10 million lines of data neo4j has a command called neo4j-admin import which will create an environment that is controlled that will work for longer amounts of time but can process terabytes of data.

We also did some custom performance logging ourselves using a python script that queries the neo4j database and logs the time it takes to return the data to us. The example below is using data from the “routes.csv” file.

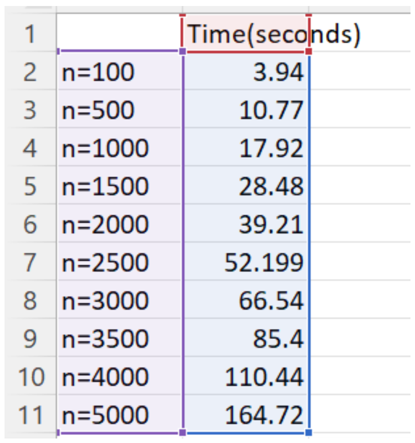


```python
for i in airline_list:
    # Escape single quotes in the airline_name and airport_name fields
   # for x in range(len(airline_list)):
    #    r = '[@_!#$%^&*()<>?/\|}{~:\']'

    #    i[x] = re.sub(r, '', i[x])

    neo4j_create_statemenet = (
        "CREATE (t:Airline {"
        "`airline`: '" + str(i[0]) + "', "
        "`source airport`: '" + str(i[1]) + "', "
        "`destination airport`: '" + str(i[2]) + "', "
        "`codeshare`: '" + str(i[3]) + "', "
        "`stops`: " + str(i[4]) + ", "
        "`equipment`: '" + str(i[5]) + "'"
        "})"
    )
    transaction_execution_commands.append(neo4j_create_statemenet)


def execute_transactions(transaction_execution_commands):
    start_time = time.time()
    data_base_connection = GraphDatabase.driver(
        uri="bolt://localhost:7687", auth=("neo4j", "password"))
    session = data_base_connection.session()
    for i in transaction_execution_commands:
```

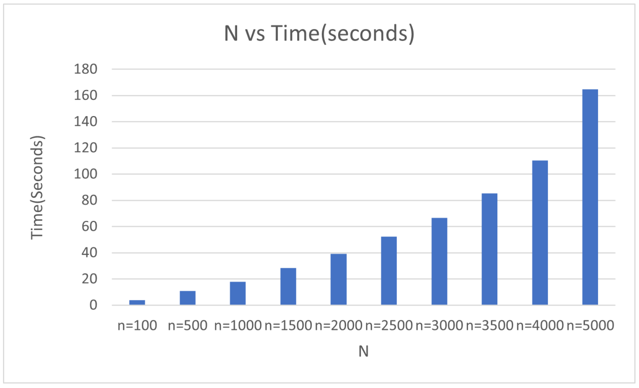


```python
            "MERGE (source:Airport {`source airport`: $source_airport})",
            source_airport=source_airport
        )

        session.run(
            "MERGE (destination:Airport {`destination airport`: $destination_airport})",
            destination_airport=destination_airport
        )

        # Create relationships between airlines and airports
        session.run(
            "MATCH (a:Airline {`airline`: $airline_name}), (source:Airport {`source airport`: $
            "MERGE (a)-[:OPERATES_AT]->(source) "
            "MERGE (a)-[:OPERATES_AT]->(destination)",
            airline_name=airline_name,
            source_airport=source_airport,
            destination_airport=destination_airport
        )

    print("--- %s seconds ---" % (time.time() - start_time))


# the edges should be the routes
execute_transactions(transaction_execution_commands)
```

Our team decided to go in increments of around 500. This would give data that would not be biased by other factors. We wanted the increment to be consistent. Here is what we found.



| 1 |  | Time(seconds) |
|---|---|---|
| 2 | n=100 | 3.94 |
| 3 | n=500 | 10.77 |
| 4 | n=1000 | 17.92 |
| 5 | n=1500 | 28.48 |
| 6 | n=2000 | 39.21 |
| 7 | n=2500 | 52.199 |
| 8 | n=3000 | 66.54 |
| 9 | n=3500 | 85.4 |
| 10 | n=4000 | 110.44 |
| 11 | n=5000 | 164.72 |



As expected, as the amount of data increases so does the time. The growth is not exponential, it follows similar growth; however, this may change as the amount of overall data changes.

3. Data Files:

   a. Are you using any. non-CSV data files (eg. Parquet files, HDFS files)? If so, describe your data transformation step.

      We are not using any non.csv data files. Provided from the original dataset is five data files that are all CSV's, our reduced data file is also a CSV. We tested with some python files to use the neo4j API but that is not a data file so it will not be described here.

   b. Include sample data, if needed, in the non-CSV file format.

      We have no sample data because all of our data files in our dataset are of type CSV.

4. Data structure and auxiliary structure.

   a. What data structure have you developed on your own to store the data in memory? For example, you may be using graph, tree or other collection types to represent the data.

   We used a graph. In the context of our search engine, the graph is the ideal choice because we can represent the different complex relationships with different entities, like airlines, airports, planes, and routes.

   The graph consists of a set of nodes and a set of edges, where each edge connects a pair of nodes. The nodes represent entities like airlines, airports, countries, and planes. On the other hand, the edges represent the relationships between the entities. For example, an airline operates a certain route, a route departs from an airport, an airline departs from an airport, etc.

The graph is saved in memory by using what's known as an adjacency list representation. In this representation, every node in the graph is linked to a list containing its nodes or edges. This is called its adjacency list. This approach is storage efficient as it only requires storage for the values related to existing edges

For example:

```
Airline(A) --OPERATES--> Route(R)
```

This is also explained as, "Airline(A) is connected to Route(R) through the "OPERATES" relationship."

Now, the adjacency list for the Airline node (A) would contain all the Route nodes that the airline operates.

This graph data structure allows us to carry out complex queries and calculations effectively, which will be necessary for the search engine. For instance, it can identify all routes managed by a specific airline or pinpoint all the airlines operating within a particular country. Moreover, the graph allows us to easily incorporate, eliminate, or modify nodes and edges in response to changes in our data.

b. Describe the data structure with appropriate pseudo code.

Neo4j works in a language called cypher but since this question is asking for pseudo code here is a class based description of our node-relation where we just reference each class to show its relations.

```
class Node {
  properties
}
class Edge {
  Node startNode
  Node endNode
  properties
}
class Graph {
  Set<Node> nodes
  Set<Edge> edges
}
class Airline extends Node {
  String airlineID
  String name
  String iata
  String icao
  String country
  Boolean active
}
class Airport extends Node {
  String airportID
  String name
  String city
  String country
  String iata
  String icao
  Float latitude
  Float longitude
  Integer altitude
  String timezone
  String dst
  String type
}
class Route extends Node {
  String airline
  String sourceAirport
  String destinationAirport
  String codeshare
  Integer stops
  String equipment
}
class Country extends Node {
  String name
  String iso_code
}
class Plane extends Node {
  String name
  String iata
}
class Operates extends Edge {
  Airline airline
  Route route
}
class DepartsFrom extends Edge {
  Route route
  Airport airport
}
```

```
class ArrivesAt extends Edge {
  Route route
  Airport airport
}
```

5. Source Code:

   a. Provide source code of your data ingestion and data query steps in a Zip file. Also include source code of any data structures you may have implemented.

      The source code of our project is in the .zip files attached called, "BMW-415-code".

      It contains these files/folders:

      - custom_python_performance_code
      - python_iteration
        - Included this just so that we attempted to use the neo4j api to use python code to make calls to the database. We did most of our work this milestone in the neo4j browser
      - airlines.csv
      - airports.csv
      - countries.csv
      - node_graph_country_plane.PNG
        - A picture to show our dataset relationships
      - planes.csv
      - reduced_dataset.csv
      - route_airline_graph.PNG
        - A picture to show our dataset relationships
      - route_airport_graph.PNG
        - A picture to show our dataset relationships
      - routes.csv