Project Statement for Milestone 2

DATA KEEPERS

Alhareth Aboud, Logan Kloft, Madee Barnwell, Nam Jun Lee, Stephany Lamas

1. Data Model and Database Tools

Describe the data model you will be using to represent the data. Justify why the data model is an appropriate one for the dataset.

The Data Keepers used a graph data model to implement an airline search engine. A graph is the ideal data model for the air travel search engine because it allows airline, airport, and route information to be linked based on their relationships to each other, as opposed to relational databases which require connections to be inferred with primary keys and foreign keys, and do not allow for as much flexibility with the data structure.

Queries related to the trip recommendation feature of the air travel search engine are good examples of how a graph with nodes and relationships can easily be traversed to provide information on routes connecting two cities, 'X' and 'Y'. Node 'X' is accessed, and the relationships and nodes of the graph are searched until a route connecting 'X' with 'Y' is found. There can be multiple routes. When these routes are presented to the user, they can make their travel plans based on, for example, taking the route from 'X' to 'Y' with the shortest number of stops. If the user knows the maximum number of stops they are willing to take on their journey, they can input their starting point (airport node 'X') and destination (airport node 'Y'), along with that maximum number of stops 'Z'. The search engine will provide the user with only the routes that have 'Z' number of stops or fewer. In the graph itself, 'Z' refers to the number of intermediate airport nodes that are traversed to get from airport 'X' to 'Y'.

What database will you be using to store the data?

The Data Keepers decided to use a graph database (Neo4j) as the database to store the air travel data. Neo4j is provided as a managed service through AuraDB, which is built to store and navigate relationships as a NoSQL. It is a database that focuses on relationships between data, which is an excellent tool for finding path patterns- the goal of this project.

2. Dataset Statistics

If you are using a graph dataset, report the following statistics:

- How large is the dataset you will be dealing with?
  Loading the graph data into an AuraDB instance produces 81,522 nodes and 201,539 relations (edges).

- What is the physical storage size (in KB/MB/GB)?
  The total size of storing all nodes and relations is 752KiB as displayed by the AuraDB sysinfo command.
- How many attributes are there for the nodes/edges?
  The Data Keepers have defined eighteen properties (attributes). At this stage, the team feels it is possible to perform queries without using properties for relations; therefore, all eighteen properties are only applicable to nodes.
- How many nodes and edges are there, and how is the graph labeled?
  Even then, there are three distinct nodes expressed by their labels which only contain related properties. Neo4J uses labels to define a node. This database design uses three node labels: Airport, Airline, and Route. Meanwhile, relations are defined by their types (labels). Currently there are three relation types, BEGIN_ROUTE, END_ROUTE, and FLOWN_BY. These relations define directed edges respectively from an Airport node to a Route node, from a Route node to an Airport node, and from a Route node to an Airline node.
- Is it directed?
  It is worth noting that Cypher, Neo4J's query language, can query relations both dependent and independent of direction.
- What is the average degree of the nodes?
  It is estimated that the average degree (in-degree and out-degree) is about 2.4722 edges (|Edges| / |Nodes| = (201,539) / (81,522) 2.4722).
- What is the density of the graph/network data?
  The density of a graph is calculated by the number of relations divided by the possible number of relations. In this case, the formula is (# of relations) / (# of nodes * # of possible relations per node) = (201,539) / (81,522 * 3) .824.

Briefly describe the functions of the parser you have implemented so far.

The team designed a parser to clean the data and establish a connection with an AuraDB instance. The parser also wrote Airport, Airline, and Route data to the database, created relations between nodes, and added indexing. Due to Python's rich module ecosystem, these tasks were accomplished with relatively little code.

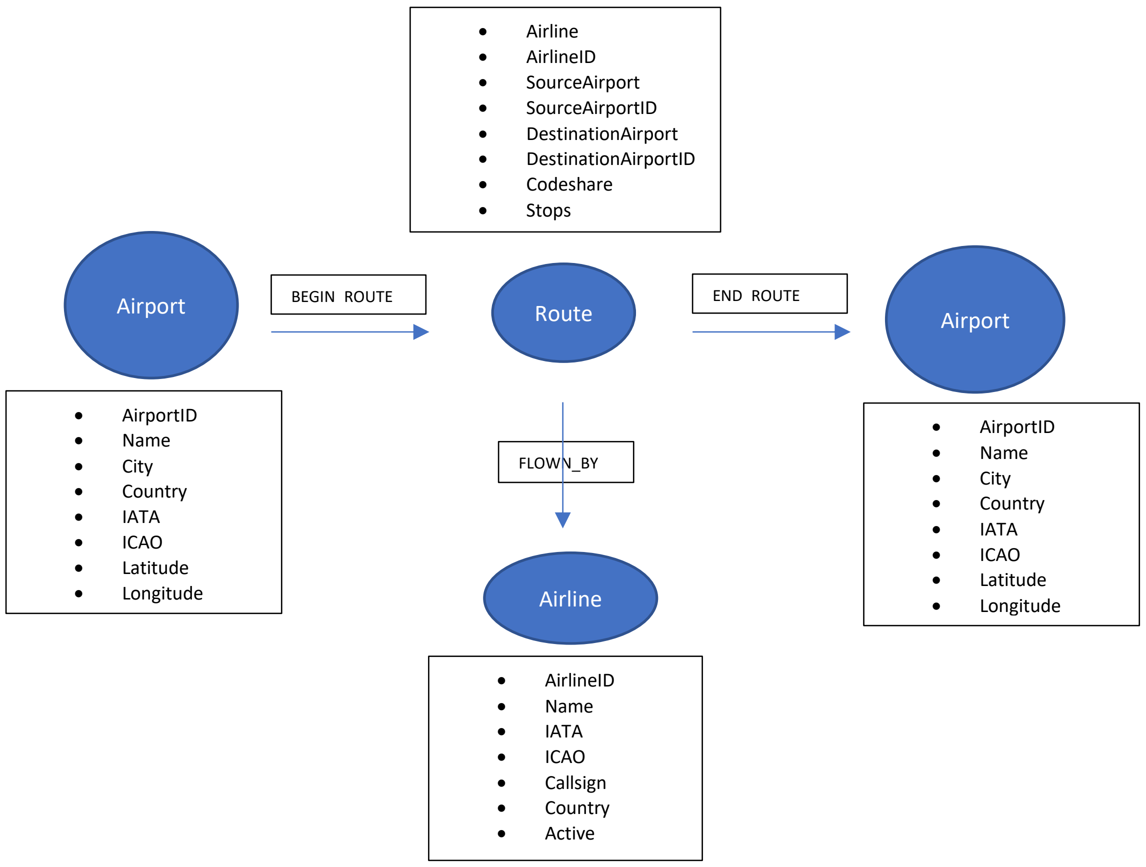
What is the estimated loading time if you have the result?

The total time for running the parser is approximately 26.3389 seconds.

3. Data Structure and Auxiliary Structure

What data structure have you developed of your own to represent the data, if any?

Data Keepers developed a graph data model structure to represent the relationships between airports, airlines, and the routes taken by the airlines to and from the airports. There is a relationship between an airport and a route where an airport is the beginning of a route, and then another relationship exists for the airport at which the route ends. There is a separate relationship between route and airline which describes that a route is flown by an airline:



Data Keepers graph pseudo code:

N1: AirportNode {AirportID: , Name: , City: , Country: , IATA: , ICAO: , Latitude: , Longitude: } –
BEGIN_ROUTERelation {Relationship: ‘BEGIN_ROUTE’} →
N2: RouteNode {Airline: , AirlineID: , SourceAirport: , SourceAirportID: , DestinationAirport: ,
DestinationAirportID: , Codeshare: , Stops: } –
END_ROUTERelation {Relationship: ‘END_ROUTE} →
N3: AirportNode {AirportID: , Name: , City: , Country: , IATA: , ICAO: , Latitude: , Longitude: }

N2: RouteNode {Airline: , AirlineID: , SourceAirport: , SourceAirportID: , DestinationAirport: ,
DestinationAirportID: , Codeshare: , Stops: } – FLOWN_BYRelationship {Relationship:
‘FLOWN_BY} → N4: AirlineNode {AirlineID: , Name: , IATA: , ICAO: , Callsign: , Country: , Active: }

4. Project Plan and Contributions

List each team member’s contribution in Milestone 2:

Logan Kloft’s Milestone 2 contributions included constructing the parser using Nam Jun’s code snippets for data cleaning, data insertion, and data indexing. Logan also uploaded the current state of the parser to GitHub along with screenshots providing information for the completion of Milestone 2. Logan wrote the Dataset Statistics section of the Milestone 2 Report and stated the team’s plans for Milestone 3.

Nam Jun Lee's Milestone 2 contributions included helping to organize parsers on data cleaning, data insertion, data relationships, and data indexing. Nam Jun Lee also actively participated in Discord discussions among team members and wrote the Database Tolls section of Milestone 2. Helped organize parsers on data cleaning, data insertion, data relationships, and data indexing.

Madee Barnwell's Milestone 2 contributions included actively participating in the team's Milestone 2 Discord discussion. She helped Logan and Nam Jun decide on labels and properties of nodes and relations. Madee also wrote the Data Model and Data Structure & Auxiliary Structure sections of the report. Madee edited team member contributions, created and formatted the report Word document, and compiled all team member contributions into the final report document.

Stephany Lamas's Milestone 2 contribution was to complete a final review of the Project Milestone 2 Report and submit on Canvas for the Data Keepers group.

What is your plan for Milestone 3

Milestone 3 requires architecting algorithms to answer the originally proposed inquiries that were stated in Milestone 1. The Data Keepers are tasked with considering ways to improve both Cypher query speed and the analysis of Cypher query results for their airline search engine project.

Neo4j itself is thread secure except where otherwise stated; therefore, to increase query speed, the team will look into implementing multi-threaded queries. This means that parallel algorithms will need to be designed in order to carry out Cypher queries that don't visit nodes more than once and return unique results. Taking advantage of indexing and queries that don't rely on cartesian products will also significantly boost query speed.

After a query is performed, the results must be processed so they are accessible to the interface for displaying filtered data depending on the initial inquiry. The algorithm must be designed to remove or ignore results of queries that may potentially return unrelated data. Another factor to consider is whether the algorithm will process results record by record or split up the results into sections and use threads to process them separately before combining the results.

The Data Keepers have two primary areas to consider for Milestone 3 regarding algorithm design. To achieve these goals, the Data Keepers will start implementation sooner rather than later. The team can measure the efficiency of the algorithms using Big-O notation for time-complexity and size-complexity which addresses scalability.