Project Statement for Milestone 3

## Data Keepers

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

# Introduction

This report explains Data Keeper’s plans regarding custom algorithms. Additionally, it recaps dataset information, system specifications, and the team’s plans to complete the project. The Data Keepers plan to use custom algorithms for questions seven, nine, and ten.

# Note

The Data Keepers initially implemented functionality using Neo4j queries through the Cypher Query Language (CQL). It has since come to the team’s attention that custom functions need to be implemented. Thus, this report outlines plans to swap from being entirely reliant on Neo4j to processing the data in memory on team members’ personal devices. Additionally, the report explains the pseudo-code of the new algorithms.

# Problem statement and algorithm description.

## Give a formal problem statement related to the algorithms. It should consist of (1) input; (2) output, and (3) goal for your solution.

Since the project team tackles a multitude of problems, input and output of each will be given individually, along with the goal of the problem solution. Notice that there is a Cypher Query and Proposed Algorithm header for some problems; that is because the team is currently in the process of switching the project’s implementation from being Neo4j reliant to in-house algorithms. Professor Srinivasulu has stated that teams need only tackle a subset of problems (2-3) for custom algorithms.

## Give a pseudo-code and corresponding description of your sequential or parallel algorithms. If your project targets multiple problems, give the pseudo-code algorithms for each of the problems.

Problem 1

|  |  |  |
| --- | --- | --- |
| (1) | | |
| Input | Output | Goal |
| A country | List of airports operating in the country. | Allow user to filter airports based on selected country. |

Cypher Query Pseudo-code

MATCH (Airport)

WHERE Country = “User Input”

RETURN AirportID, Name, City, Country

Custom Algorithm Pseudo-code

# only Airport records

Map(key, record):

    emit (record.Country, record)

Reduce(Country, [records])

    If Country == TargetCountry

        emit(Country, [records]

Problem 2

|  |  |  |
| --- | --- | --- |
| (2) | | |
| *Input* | *Output* | *Goal* |
| A number of stops | List of airlines having the number of stops | Allow user to filter airlines based on a stoppage amount. |

Cypher Query Pseudo-code

MATCH (Route)-[FLOWN\_BY]->(Airline)

WHERE Stops = “User Input” Route.AirlineID = Airline.AirlineID

RETURN DISTINCT AirlineID, Name, Stops

ORDER BY AirlineID ascending

Problem 3

|  |  |  |
| --- | --- | --- |
| (3) | | |
| *Input* | *Output* | *Goal* |
| A codeshare | List of airlines operating with the codeshare | Allow user to view airlines operating with the codeshare. |

Cypher Query Pseudo-code

MATCH (Route)-[FLOWN\_BY]->(Airline)

WHERE Codeshare = “User Input”

AND Route(AirlineID) = Airline(AirlineID)

RETURN DISTINCT AirlineID, Name, Codeshare

Problem 4

|  |  |  |
| --- | --- | --- |
| (4) | | |
| *Input* | *Output* | *Goal* |
| ‘Y’ or ‘N’ | List of active or inactive airlines | Allow user to view airlines that are currently active. |

Cypher Query Pseudo-code

MATCH (Airline)

WHERE Active = “User Input”

RETURN DISTINCT AirlineID, Name, Active

Problem 5

|  |  |  |
| --- | --- | --- |
| (5) | | |
| *Input* | *Output* | *Goal* |
| ‘Y’ or ‘N’ | List of top ten countries with the highest number of airports. | Allow user to view interesting statistic of dataset. |

Cypher Query Pseudo-code

MATCH (Airport)

RETURN Country, COUNT(num of airports in each Country) AS Count

ORDER BY Count descending LIMIT highest 10

Problem 6

|  |  |  |
| --- | --- | --- |
| (6) | | |
| Input | Output | Goal |
| A number | List of the top cities with the most incoming and outgoing flights in addition to displaying the number of incoming, outgoing, and total flights at the city. | Allow user to view interesting statistic of dataset. |

Cypher Query Pseudo-code

MATCH ()-[incoming)->(ALL)-[outgoing]->() WITH ALL, incoming, outgoing

RETURN City, COUNT(DISTINCT incoming), COUNT(DISTINCT outgoing), COUNT(total incoming and outgoing) AS Total

ORDER BY Total descending LIMIT “User Input”

Problem7

|  |  |  |
| --- | --- | --- |
| (7) | | |
| *Input* | *Output* | *Goal* |
| Two cities | List of routes connecting the two cities | Provide at least one route between two cities. |

Cypher Query Pseudo-code

Using All shortest paths Algorithm

MATCH PATH= allShortestPaths(Airport {Source City: “User Input”}-> Airport {Destination City: “User Input”})

RETURN PATH

Proposed Custom Algorithm

# 1st map -> Airport and Route Records go to same Map

MAP(key, record)

    If record is Airport:

        emit(Airport.ID, Airport)

    If record is Route

        emit(Route.SourceAirportID, Route)

REDUCE(ID, [records])

    For each record in records:

        If record is route:

        emit(ID, [ID, record.DestinationAirportID])

# 2nd map -> expand frontiers until find both cities

Repeat until flag = true

MAP2(key, record)

    Parse record to ID, [path]

        emit(path[0], path)

        emit(path[-1], path)

REDUCE2(ID, [paths])

    For path\_i in paths:

    For path\_j in paths:

        Path’ = Combine path at point where the ID are the same

        If city1 and city2 in path:

            Add Path’ to list of possible paths

            Set flag = true

        Else

            emit (id, path’)

Iterate through collected paths and use ID to get the node information

Display paths

Problem 8

|  |  |  |
| --- | --- | --- |
| (8) | | |
| *Input* | *Output* | *Goal* |
| Two cities and a number of stops | List of routes connecting the two cities with less than the specified number of stops | Allow user to filter their route suggestion. |

Cypher Query Pseudo-code

Using BFS Algorithm

MATCH (start: Airport {Source City: “User Input”}, (end: Airport {Destination City: “User Input”}) CALL apoc.path.expandConfig(start, {labelFilter: -Airline, relationshipFilter: ‘BEGIN\_ROUTE>|END\_ROUTE>’, terminatorNodes:[end], maxLevel: ‘User Input (Stops)’}) YIELD path RETURN nodes of path as Route, length of path as Stops

Problem 9

|  |  |  |
| --- | --- | --- |
| (9) | | |
| *Input* | *Output* | *Goal* |
| A city and number of stops | List of cities that can be reached from the city within the specified number of stops | Allow user to view cities they can reach from one city with a limiter based on the number of stops. |

Cypher Query Pseudocode

Using BFS Algorithm

MATCH (Airport) WHERE Airport.City = “User Input” CALL apoc.path.expandConfig(Airport, {maxLevel: “User Input (Stops)”, labelFilter: ‘>Airport|-Airline’, relationshipFilter:’BEGIN\_ROUTE>|END\_ROUTE>’, uniqueness:’NODE\_LEVEL’}) YIELD path RETURN last nodes of path city (Destination City) as City, length of path as Stops

Proposed Custom Algorithm

Problem 10

|  |  |  |
| --- | --- | --- |
| (10) | | |
| *Input* | *Output* | *Goal* |
| The graph of airports, airlines, and routes | The transitive closure of the airports. | Allow fast query for reachability between two cities. |

Cypher Query Pseudo-code

MATCH PATH = (Source Airport)-[BEGIN\_ROUTE]->(Route)-[END\_ROUTE]->(Destination Airport) WITH ALL, point(Source Airport.Longitude, Source Airport.Latitude) AS source, point(Destination Airport.Longitude, Destination Airport.Latitude) AS destination WHERE Source Airport.City = “User Input” RETURN distance of source to destination as Travel Distance, Source Airport City, Source Airport Name, Destination Airport City, Destination Airport Name, AirlineID ORDER BY Travel Distance Ascending

## Discussion of the optimization techniques you are evaluating or have implemented.

Indexes have been added to all nodes (Airport, Airline, and Route) within Neo4j. The indexes are based on the OpenFlights ID for each airport or airline in the case of the Airport and Airline nodes respectively. Route nodes are a little more complex since they don’t have unique fields. Instead of using one field to identify an index for Route nodes, the Data Keepers used three: SourceAirportID, DestinationAirportID, and AirlineID. Indexes allow Neo4j to make faster queries because they reduce the amount of searching that has to be done when looking for a specific node.

In addition to adding indexes to increase query speed, the team has reduced the properties that a node can contain. It was found that the queries did not make use of certain properties in either the input, output, or query so they were removed for the sake of saving space. Another change that was made was to change the types of the properties. Originally, all of the properties were strings; however, because it is faster for queries to compare and store numbers than it is with strings, the properties were converted to types that made sense for them.

Inside of the custom algorithms, the Data Keepers are considering the use of dictionaries to store Airport and Airline nodes based on their IDs. This would significantly improve retrieving information when having access to the Airport or Airline ID. Another idea being considered is hashing Airport and Airline nodes into buckets based on their city and country strings. This is due to the fact that many of the queries rely on an initial city or country for input. Creating buckets of nodes will improve search time, which will directly improve queries for questions seven, eight, and nine that rely on an initial city or country.

Another optimization being considered by the team for the custom algorithms is the use of parallelism. In the second, third, and fourth queries, they rely on a linear search through the nodes. Splitting up the search into bounded searches using parallelism will allow for significant speedup.

# Report on the components of the experimental plan.

Since the Data Keepers have already implemented the functionality using Neo4j, the team plans to compare the custom algorithms to those solutions. Neo4j will act as the baseline solution while the custom algorithms will be experimental.

## Dataset description

The air travel data set contains three relevant datastores. They are airlines.csv, airports.csv, and routes.csv. There are 6,162 records in the airlines.csv file, each of which has 8 fields. There are 7,698 records in the airports.csv file, each of which has 14 fields. There are 67,663 records in the routes.csv file, each of which has 9 fields. The dataset contains gaps in some of the records and uses different values to signify a null value. Additionally, as mentioned in the optimization section, there are unnecessary fields. Therefore, the team has constructed a function that fills all empty fields and fields with null values to a single null value that is uniform across the three files, ‘N/A’. The unnecessary columns are also dropped during this process. After running through this process, the files are not considered clean and ready to be loaded into a datastore.

While in AuraDB, the statistics are as follows: The total size of storing all nodes and relations is 752KiB, which can be displayed by the AuraDB sysinfo command. There are eighteen properties (attributes) within the air travel data set which are for the nodes. There are three distinct nodes with labels: Airport, Airline, and Route. Meanwhile, relations are defined by their types (labels): BEGIN\_ROUTE, END\_ROUTE, and FLOWN\_BY. Respectively, these relations define directed edges 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. Cypher, Neo4’s query language, can query relations both dependent and independent of directions. It is estimated that the average degree (in-degree and out-degree) is about 2.2589 edges (|Edges| / |Nodes| = (201,539) / (89,220)). The density of a graph is calculated by the number of relations divided by the possible number of relations. For the air travel dataset, the formula is (# of relations) / (# of nodes \* # of possible relations per node) = (201,539) / (89,220 \* 3) = .7530.

## Algorithm description

The Data Keepers are planning to implement custom algorithms for questions 1 and 7; therefore, their Cypher implementations will be used as the baseline algorithm. In addition, pseudo-code has been created for some other questions and will be implemented given enough time.

### Baseline Algorithms

Problem 1 uses a basic MATCH and WHERE statement in Neo4j’s Cypher language. Since indexing does not help, it’s likely that the query iterates over every node in the AuraDB instance.

Problem 7 uses the all shortest paths algorithm, which is a function embedded in CQL that can find all shortest paths connected between nodes. In order to find the optimization technique of problem 7, the execution speed of all shortest paths algorithm and BFS algorithm were compared to find the optimal technique.

### Custom Algorithms

The team has not yet not implemented the custom algorithms described by the pseudo-code in the problem statements and algorithm description; however, they are described here.

Problems 1, 7 and 9 custom implementations will use MapReduce. For question 1, we will apply MapReduce to the airport’s file. Map will parse an airport record from the file into its city and the record itself. Then, the map will emit the city as the key to aggregate on for the reducer. The reducer will simply emit the (city, [records]) if it matches the input city. Where the list of records are actually the airports.

In the case of problem 7, it is a little more complicated. As a general overview, the first mapper takes in records from both the airport and route file. Then depending on if the record is an airport or route, it will emit something different. For an airport, it emits the airport id as the key and the record as the value. For route, it emits the SourceAirportID as the key and record as the value. Then the reducer will combine routes’ DestinationAirportId with the key. The end result is a path of length two. A characteristic is that all routes are included in the information so we only need to connect these routes together. The next cycle of map emits the front and end id of a path as a key and the entire path as the value so that the reducer can combine two different paths which contain the same nodes (think of these points as a pivot). The reducer combines the paths based on the ‘pivot’ and checks if the input cities are inside of the path. If so, it sets the flag to true and should add the path (between i and j) to a global path variable. While the flag is false, continue running the second version of Map and Reduce on the output file which will have the paths grow until a match is found.

For problem 9, (Steph’s Part)

## Settings

## Hadoop (MapReduce)

The Data Keepers are considering the use of Hadoop on members’ personal computers to process the initial data. Results will be derived using a computer with 15.7 GB of usable RAM, and an i7-10510U CPU @ 1.8 GHz which has 4 cores and 8 logical processors.

### Neo4j Aura Platform

Neo4j Aura is a graph platform delivered as a cloud service, making it very easy and fast to search and navigate more connected data, instead of spending time managing infrastructure. Basically, to express data, meaningful insights can be derived by connecting relationships with nodes and edges. The instance size used is Nodes 89220 / 200000 (45%), and Relationships 201539 / 400000 (50%).

## Plan

For problem 7, the custom implementation is being compared to the allShortestPaths algorithm from Neo4j. For problem 1, the basic MATCH and WHERE query used by Neo4j will be compared.

The Data Keepers will use the runtime for processing the same size of data to compare speed, and will provide a complexity analysis for the custom algorithms. Since the team is not sure which algorithms Neo4j uses under the hood for the queries involved in questions 1 and 7, a complexity analysis will not be included for those. In addition to discussing complexity analysis and comparing runtimes, the team plans to analyze the output sizes between the custom and non-custom algorithms with respect to records produced. Accuracy of the algorithms can be determined this way.

Since the size of the data will be kept the same, the efficiency of the algorithms will be tested using queries that theoretically interact with a high number and low number of nodes in the database.

Expected Values Based on Baseline Algorithms:

|  |  |  |
| --- | --- | --- |
| Problem | High | Low |
| 1 | Input: “USA”  Output:  Length: 3024  Time: ~1.2 seconds | Input: “Iceland”  Output:  Length: 44  Time: ~.15 seconds |
| 7 | Input: “Seattle”, “New York”  Output:  Length: 187  Time: ~ 1 second | Input: “Keflavik”, “Mauterndorf”  Output:  Length: 0  Time: ~.2 seconds |
| 9 | Input: “Seattle”, 10  Output:  Length: 3055  Time: ~6.1 seconds | Input: “Seattle”, 1  Output:  Length: 1161  Time: ~1.2 seconds |

## Completion Plan

The team believes that scalability is the most important factor when considering parallel algorithms since an application is limited by its ability to scale. Although additional flight data isn’t available to test, the team agrees that a theoretical analysis of the scalability is sufficient for determining the efficiency of the solution.

In addition to calculating the scalability of the solution, the team believes that computation speed for each query will be the most important figure to present in the experimental study. At the end of the day, users don’t necessarily care about the scalability of the solution, rather they expect solutions which are quick. If the Data Keepers are able to construct a solution that surpasses the use of Neo4j queries alone, then the time invested into development will be considered a huge success.

# Project plan and contributions.

## List each team member’s contribution in Milestone 3.

Stephany Lamas’s Milestone 3 contributions included creating the project Milestone 3 Discord channel and Google Doc, creating the Proposed Custom Algorithm for Problem 9 and adding the approach and pseudo-code for it.

Nam Jun Lee’s Milestone 3 contributions included helping write code for the problem in the project. Nam Jun also actively participated in the Data Keepers’ Discord channels and wrote the pseudo-code algorithms section of Milestone 3.

Madee Barnwell’s Milestone 3 contributions included working on question two and compiling, editing, formatting, and submitting the Milestone 3 report. Madee also actively participated in the team’s Milestone 3 discord channel.

Logan Kloft’s Milestone 3 contributions included helping write code for the problem in the project, participating in Discord channels, writing the introduction section, notes section, part of the problem statements and algorithm descriptions section, optimization section, part of the experimental plan section including adding to the dataset description, the algorithm description section, the settings section, the plan section, and the completion plan section, and plan for presentation and demonstration section.

## What is the team’s plans for the final presentations and demonstrations?

The team has created a UI to display the queries. At this point, the main goal is to implement custom algorithms for problems 1 and 7, with 9 being a bonus given the time. Then, collect analysis data and compare them to the baseline algorithms. Completing the analysis and implementation of the custom algorithms will allow us to start constructing the written report and give us the content required to make a convincing presentation.

For the presentation and demonstration part of the project, the team is considering the use of PowerPoint to describe the problem statement. To showcase the content from the class that has been applied throughout the project, the presentation and demonstration will point to specific instances in the Data Keepers codebase where the feature is being used. If the feature is difficult to visualize, then it will be included as a talking point in the presentation. Having a PowerPoint presentation will make the overall demonstration coherent and organized.

Ensuring that the Data Keepers are working as a team will provide the best chance of success for completing the remaining work.