Airline Search Engine

Final Project Report

CPT_S 415 - Big Data

WASHINGTON STATE UNIVERSITY

Team: CPTS-415_BMW

Brian Joo

Noah Waxman

Mitchell Kolb

Fall 2023

Table of Contents

Table of Contents **2**
Introduction **3**
Problem Definition 3
What Is Being Solved / Achieved 3
Why Does It Matter 3
Related Work **4**
How Does this Project Relate to Previous Work 4
Architecture / Design **4**
Overview 4
Dataset 5
Data Model / Database Schema 5
Algorithms / Queries 7
User Interface / User Interaction 9
Results and Findings **10**
Mathematical Analysis 10
Scalability 12
Conclusion **12**
References **12**

Introduction

Problem Definition

The aviation industry consists of a complex network of airlines, airports, planes, and routes that span the globe. With thousands of airports and airlines, and countless routes, finding the most efficient and convenient options for travel is a challenging task for many travelers. To address this, we developed an aviation data search engine, utilizing Neo4j, PySpark, PySimpleGUI, and graph algorithms.

The search engine will use datasets publicly available from openflights.org. The datasets contain details such as airport ID, airport name, main city served by airport, country or territory where airport is located, code of airport, decimal degrees, hours offset from UTC, timezone, and more. With this information, the engine provides the user with a variety of search options, such as searching for airports and airlines, finding the list of airlines operating with code share, finding active airlines in the United States, and determining which city or territory has the highest number of airports.

Moreover, the engine will also provide trip recommendation features, where we define a trip as a sequence of connected routes. It will be able to find a trip that connects two cities with less than a specified number of stops (constrained reachability) and find all cities reachable within a certain number of hops from a starting city (bounded reachability). There will also be a shortest path algorithm which will take in a starting place and destination and will find the shortest available trip that meets that criteria. This algorithm will take advantage of additional graph processing by using the PySpark service.

What Is Being Solved / Achieved

The goal of this project will be to use the given datasets and to use tools such as MapReduce,SQL to implement a search engine. The user should be able to use specific functions and input their own values such as, name of the airport, city, country, etc. One of the key components is the shortest trip path feature. There are many subcomponents of this, such as if the user enters a destination that has no available trips to it or if during the algorithmic analysis it is determined that there is more than one trip that takes the same amount of time. The output that will be displayed in both of these scenarios is an important thing that our team will need to consider when developing this search engine.

Why Does It Matter

The aviation sector plays a vital role in boosting global productivity and fostering economic progress. Our tool doesn't just streamline the process of selecting airlines and airports for travelers; it also optimizes operations for airports, airlines, and governmental entities. For everyday individuals, this means smoother travel experiences, quicker access to reliable information, and ultimately, a more efficient and hassle-free travel experience.

Related Work

How Does this Project Relate to Previous Work

Neo4j and Graph Visualization

One of the key technologies to the success of our project is Neo4j, a graphical NoSQL database management system. Labeled as an ACID-compliant transactional database employing native graph storage and processing, Neo4j stands apart from relational database management systems and other NoSQL databases we evaluated for this project. A document titled "Neo4j as a graph database" offers an extensive summary of Neo4j's features [1].

The author also showcases the performance of Neo4j, explaining that performance remains high even as data scales significantly. For our project, this is crucial as our dataset is expected to expand over time. Moreover, Neo4j makes visualization of complex networks intuitive, facilitating clearer analysis and identification of optimal routes.

Furthermore, we leveraged Neo4j's graph visualization to gain insights from our data. As noted in a paper titled "15 Tools for Visualizing Your Neo4j Graph Database", visualizing graphs can be a difficult task for developers with a database, but numerous tools have been developed to make graph visualization easier [2]. Neo4j Bloom, one such tool, was instrumental in visualizing our graphical database.

PySpark and Graph Processing

Another paper that is particularly relevant to our project is titled "Large scale graph processing with Apache Spark" [3]. This paper discusses the use of Apache Spark, particularly PySpark, for processing large scale graphs.

The paper talks about PySpark being a powerful tool, specifically for distributed data processing. In our project, we utilized PySpark to process some of the large volume aviation data efficiently. PySpark excels in its ability to parallelize operations across multiple nodes, which significantly improves process speed.

Architecture / Design

Overview

The architecture for our project is based on the openflights.org dataset and the Neo4j graph database. The dataset contains detailed information about airports, airlines, routes, planes, and countries, stored in separate CSV files. The graph database model was chosen for its ability to model complex relationships between entities, a characteristic inherent in the dataset. Particularly, this is a strength of Neo4j. Nodes in the graph represent airports, airlines, and cities, while edges represent routes, codeshares, and other relationships. The Neo4j database

system was selected for its strength in displaying graphical data and the ease of importing our database.

Dataset

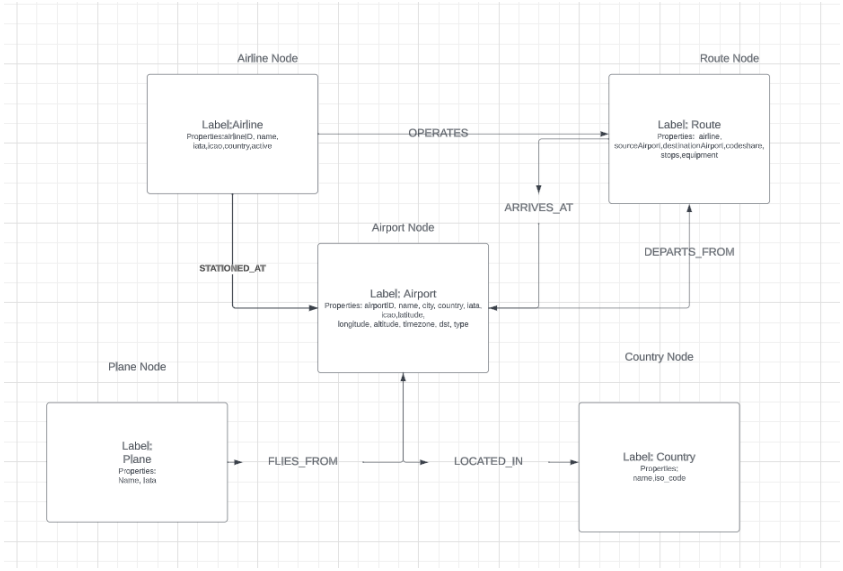
The openflights.org dataset for the “Airline Search Engine” was created to collect data from airports, airlines, and the routes that they take. It contains over 10,000 airports, train stations, and ferry terminals that span the globe. This dataset contains many collections of data that are dedicated to certain aspects of the travel world. Such as airlines, airports, routes, planes, and countries. The dataset contains five comma separated value (.csv) files which are labeled airlines, airports, routes, planes, and country. Here are the size details for each file in the dataset: airlines (400KB) 6162 rows and 8 columns, airports (1.3MB) 7698 rows and 8 columns, routes (2.27MB) 67663 rows and 9 columns, planes (5KB) 173 rows and 3 columns, and country (5KB) 261 rows and 3 columns. Each one of these categories have a single file corresponding to them and the file extension is .dat.

Data Model / Database Schema

The data model that we have decided on to represent our openflights dataset is the graph model. Graphs are useful for modeling complex relationships between entities, which is the case in this dataset. The graph would have nodes representing airports, airlines, and cities, and edges representing routes, codeshares, and other relationships. What made the graph model appealing in the first place is that our dataset is fully composed of named relationships with specific properties which is what graph models are exceptionally good at. Our other choice that was in the running was a document based data model but after analyzing our data closer we decided that our dataset wasn’t formatted in such a way that the key-value collection that the document data model can effectively take advantage of. Although our data does have lots of index-centric values which made the document model appealing in the first place.

When using the graph model we think it is best that each node would have properties such as the airport's ID, name, city, country, IATA code, ICAO code, latitude, longitude, altitude, timezone, and type. For airlines, properties would include the airline's ID, name, alias, IATA code, ICAO code, callsign, country, and active status. For cities, properties would include the city's name and country. Edges would represent routes, codeshares, and other relationships. For routes, properties would include the airline's ID, the source airport's ID, the destination airport's ID, and the number of stops. For codeshares, properties would include the airline's ID, the source airport's ID, the destination airport's ID, and whether the flight is a codeshare.

Below is the data model diagram for our non-relational database:



Below is the Neo4j Cypher statements for creating the nodes and edges 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);

MATCH (s:Airport), (d:Airport)
WHERE s <> d
CREATE (s)-[:DISTANCE {distance: 1}]->(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. 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.

Algorithms / Queries

- Find airline details
    - This function takes in the name of an airline and returns all the node information regarding that airline in a python dictionary.
- Find active airlines in the US
    - As the name suggests, it simply returns the list of airlines that are currently active AND in the US.This is how the query works.In the find all active airlines, the match query is responsible for matching/finding the nodes that are labeled as Airlines.
- Find the top k cities with the most incoming and outgoing airlines
    - We take in one parameter which limits how many results we return.
    - We use query Neo4j to accomplish the following:
        - Match any routes that arrive at any airport
        - We return the city of each airport and the count of distinct airlines that have routes arriving at that airport
        - We order the results by the number of airlines (descending)

        - Limit the results to the top k cities
    - We then process the result of the query by creating a list of tuples, where each tuple contains a city and the count of airlines for that city. We then create two separate lists, cities (which contains the name of the cities) and airlines (which counts the airlines for each city). After the data is collected, we use a Python library, matplotlib, to create a graph of the top cities. This graph is saved as a PNG and returned to our UI for the user to see.
    - Both the incoming/outgoing functions use the same structure that was just described, except for the outgoing function, we use a query that looks for routes that depart from any airport (using the DEPARTS_FROM relationship that we created), which lets us identify major cities for takeoff. Contrarily, in our incoming function, we examine the ARRIVES_AT relationship.
- Find cities within a specified number of hops from a city
    - This function returns the cities within d-hops by performing a breadth-first search on our neo4j graph. We treat each city as a node and there is an edge between two cities if there is a direct flight between them. The function runs a query to get all paths from the starting city to any other city within d-hops. We use the APOC spanning tree function, provided by Neo4j, to perform the BFS traversal.
    - We also provide a maxLevel parameter to limit the number of hops (d-hops). The query returns a set of paths, where each path is a sequence of nodes, where each node is an airport. We iterate over these paths, and for each path, we grab the nodes and add the city of each node to the cities set. We use a set here to duplicate results.
- Determining which country has the highest number of airports
    - It is a simple function that returns the top X countries with the highest number of departing flights. What the user sees is a list of the top X countries with the number of departing flights in descending order.This is what the function does.
    - This query matches the flight nodes that are connected to the airport nodes. This connection is the relationship called "Departs_FROM". We are using the departs_from relationship because the team is specifically looking for flights departing from airports. In the with part of the query, the query aggregates and counts the flights for each source airport city, and then the query returns city names and number of departing flights. The airport is also assigned to a sourceAirport node. The "with" clause is used to create a new result set. It records the number of flights departing from each airport and assigns it to numberOfSourceAirports. It then sorts the numberOfSourceAirports in a descending order. simply limits the result to the top X cities
    - The final output is "city", "# of departing flights".
- Find a trip that connects two cities X and Y (reachability)
    - At the highest level, we fetch airport data for two provided cities using a helper function. The helper function retrieves all airports in a given city from the database by matching all airport nodes with the provided city. We use a dictionary for each matching airport that contains info like the name, iata code, lat, long, etc. We then use a nested loop that iterates over each airport in city_x

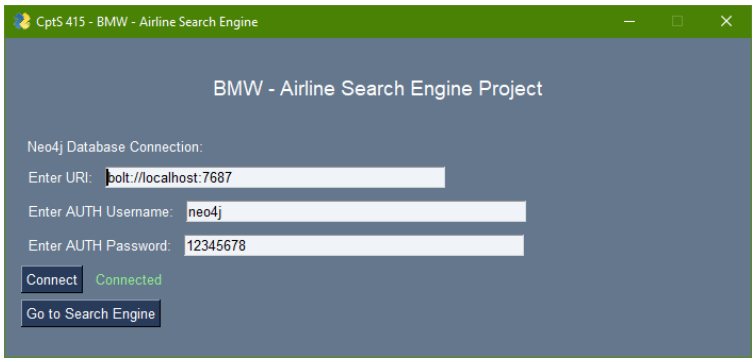
(outer loop) and city_y (inner loop). For each pair of airports, we run a Cypher query to find the shortest path between them (using neo4j's builtin shortest path algorithm, which uses Dijkstra's method).
    - In our query, we consider a maximum depth of 15, or there can be at most 15 hops between airports, to prevent excessively long and impractical routes.
    - This query returns a path, which is a sequence of nodes (airports) and relationships (flights). We take the path and extract the names of the airports and the route. The result is a list of routes, each representing a string which details the route number and the airports at the start and end of the route.
- Determine the Shortest Path from one airline to another airline
    - This function takes in an airport name that the user wants to start at and then an airport name that the user wants to go to. This function takes advantage of PySpark to effectively use data frames to compute the results of the shortest path between both of the airports. Here's what the function does.
    - 1.) Create a spark session
    - 2.) Query the neo4j database to get the information needed to complete this task
    - 3.) Define vertices and edges dataframes: vertices = spark.createDataFrame() and edges = spark.createDataFrame()
    - 4.) Create a graphframe: graph = GraphFrame(vertices, edges)
    - 5.) Find the shortest paths from the vertex to all other vertices using: shortest_paths = graph.shortestPaths
    - 6.) Return the result and close the session.

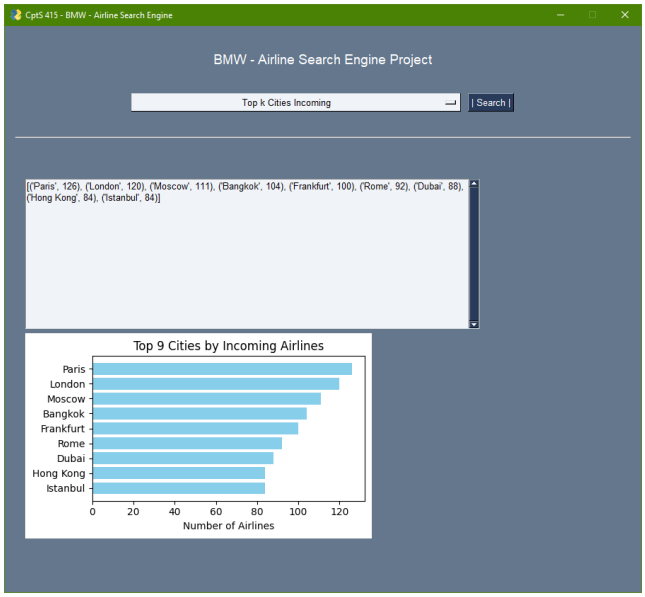
User Interface / User Interaction

Our original plan for the user interface was to provide a console-based menu to the user, where they can select different options to search and retrieve information from the database. After testing this UI out our team concluded that it would be more beneficial to go to a more graphical UI. Our new plan for an interactive user interface is to use PySimpleGUI. This python GUI library is based off of tkinter and allows us to use 40+ widgets to build our UI. For example we can have a menu list that includes options like finding the list of airports operating in a country, finding airlines with a certain number of stops, finding active airlines in the United States, and more. The user can interact with the system by selecting a menu option and entering any additional required inputs. We also have added bar graphs using the matplotlib library to make the interface more interactive for the user.

For our UI we have two screens that the user will interact with. The first screen is the Neo4j database connection screen. This screen allows the user to enter in their local database information and then gives them the ability to move onto the search engine screen once the connection is verified. The second screen is the search engine page that has a similar look and feel to other popular search engines. It includes a drop down menu which lists all of our functions that are referenced above in the architecture section. If the user selects an option that requires input an input box will appear and allow the user to enter in the required information then the result will populate in the text box at the bottom of the screen.

Below is two screenshots of both screens in the user interface:



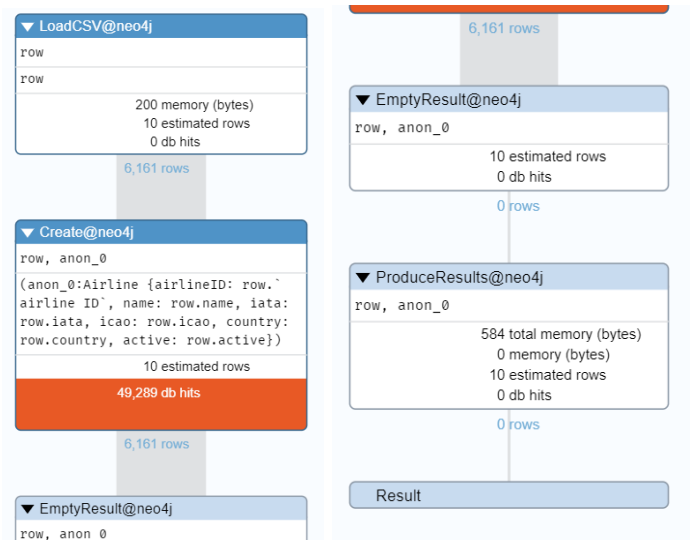


Results and Findings

Mathematical Analysis

Our team's analysis revolved around the data ingestion of our dataset. 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});



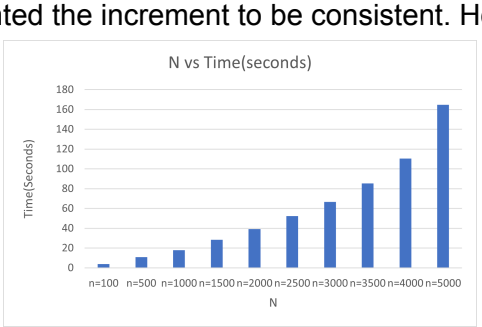
When ingesting the data from the airlines.csv file there are 6161 line items. To perform this command neo4j used about 584 bytes of memory. The documentation talks about how the neo4j underlying architecture is designed to handle a large quantity of data at a given time. The "LOAD CSV" command is rated 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. 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.

```
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:
```

```
            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.

		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.

Scalability

Our team's scalability testing revolved around using different hardware configurations to see how that affects the performance of the program. We found out that sharding may not be beneficial because the dataset is already small and the benefits are negligible. Parallel processing is technically possible but may not result in a significant performance improvement. Horizontal Scaling and vertical scaling will increase user/query quantity and we tested that on two different hardware setups with two different queries that vary in complexity. The results indicated that more resources can help us handle more traffic but will not help our project execute its functionality faster because it is already fast using Neo4j.

On the setup with 1 CPU core and 4 GB of ram.
Query 1 executed streaming 1 record after 1ms and completed after 32ms
Query 2 executed streaming 5 records after 16ms and completed after 284ms

On the setup with 4 CPU cores and 10GB of ram.
Query 1 executed streaming 1 record after 1ms and completed after 26ms
Query 2 executed streaming 5 records after 10ms and completed after 211ms

Conclusion

Our aviation search engine project was a success. We used a graph model to represent the openflights dataset, which has shown to be a useful choice seeing as it allows for efficient querying and representation of complex relationships between entities; particularly the nodes and relationships we created. By using Neo4j and PySpark, our search engine is capable of returning efficient queries and results to the user that can scale in the future if the dataset increases in size. This is evident by the algorithms we implemented, which successfully execute the search options we sought to create. Moreover, our UI turned out to be simplistic, intuitive, and easy to use for the user.

References

1. [1] J. Hoppa, “Graph Algorithms in Neo4j: Shortest Path,” Graph Database & Analytics, Dec. 10, 2018. https://neo4j.com/blog/graph-algorithms-neo4j-shortest-path/ (accessed Dec. 09, 2023).
2. [2] N. de Jong, “15 Tools for Visualizing Your Neo4j Graph Database,” Neo4j Developer Blog, May 25, 2021. https://medium.com/neo4j/15-tools-for-visualizing-your-neo4j-graph-database-ff7315873032 (accessed Dec. 09, 2023).

3. [3] W. Suen, “Large-scale Graph Mining with Spark: Part 2,” Medium, Oct. 16, 2018. https://towardsdatascience.com/large-scale-graph-mining-with-spark-part-2-2c3d9ed15bb5 (accessed Dec. 10, 2023)
4. [4] Garcia-Molina H, Ullman JD, Widom J. Database Systems : The Complete Book. Second edition. Pearson Prentice Hall; 2009.
5. [5] Erl, Thomas, et al. Big Data Fundamentals : Concepts, Drivers & Techniques. Prentice Hall, 2016.
6. [6] Leskovec, J., Rajaraman, A., & Ullman, J. D. (2022). Mining of massive datasets. Cambridge University Press.