Graph Theory and its Applications - UE19CS323
Report: Assignment-2
Metropolitan Transport System using Neo4j

Date: 19/11/2021

Sumukh Raju Bhat	Pratheek P	Pranav Rajnish
PES1UG19CS519	PES1UG19CS348	PES1UG19CS344

Problem Statement:

We represent Bangalore's major transport systems and facilities using a graph database called Neo4j and query data out of it. The transport system includes the metro(green and purple line), bus terminals and railway stations along with parking facilities, points of interest surrounding them and location information.

Choice of DBMS:

Choosing RDBMS or traditional DBMS is not so good for the above problem statement as shortest path between locations, path between locations and other queries are recursive queries(Recursive queries are queries on relationships which are of the type "entity related to itself". Eg: Employee manages Employee). It is a known fact that for recursive queries in RDBMS, we need to know before querying, how many exact number of joins we need to perform which could be a disadvantage of sorts, as we do not know after how many locations exactly we have the target location. Hence graph databases like Neo4j are an ideal choice for the above mentioned problem statement.

In general, why graph databases?

A graph database treats the relationship between the data with equal importance as the data itself. It holds the data without any predefined structure or constraints. With a graph database it is very easy to visualize the relationship between data points.

This model lends itself to particular situations where modelling the data as a graph is easier and more intuitive than trying to separate the data into numerous tables.

In general, why Neo4j?

Neo4j is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for your applications.

Some of the advantages of using Neo4j are:

1) It is very easy to represent connected data using Neo4j CQL, which is a query language that has similarities to traditional SQL.
2) It provides fast and easy traversal of widely connected data.
3) It is easy to represent semi - structured data in Neo4j.
4) No joins.

Defining the Nodes:

Creation of nodes with labels “PurpleStation” and “GreenStation”, each of them having the properties **name**, **hasParking**, **hasRailway**, **hasBusTerminal** and **location:**

CREATE (n:PurpleStation {name:'Kengari', hasParking: 1, hasRailway: 0, hasBusTerminal: 0, location: 'Kengeri'})

CREATE (n:GreenStation {name:'Silk Institute', hasParking: 0, hasRailway: 0, hasBusTerminal: 0, location: 'Talaghattapura'})

Creation of nodes with label “BusTerminal”, each of them having properties **closeTo**, and **location:**

CREATE (n:BusTerminal {closeTo: 'Majestic', location: 'Sevashrama'})

Creation of nodes with label “RailwayStation”, each of them having properties **closeTo**, and **location:**

CREATE (n:RailwayStation {closeTo: 'KSR', location: 'Sevashrama'})

Creation of nodes with label “RailwayStation”, each of them having properties **closeTo**, and **location:**

CREATE (n:PointOfInterest {closeTo: 'Cubbon Park', location: 'Shivaji Nagar'})

Defining the Relationships:

Creating a relationship **“CONNECT”** with attribute **transport** between two stations:

MATCH (s1:Station{name:'Mysore Road'}),(s2:Station{name:'Deepanjali Nagar'})
CREATE (s1)-[r:CONNECT{transport: 'metro'}]->(s2)

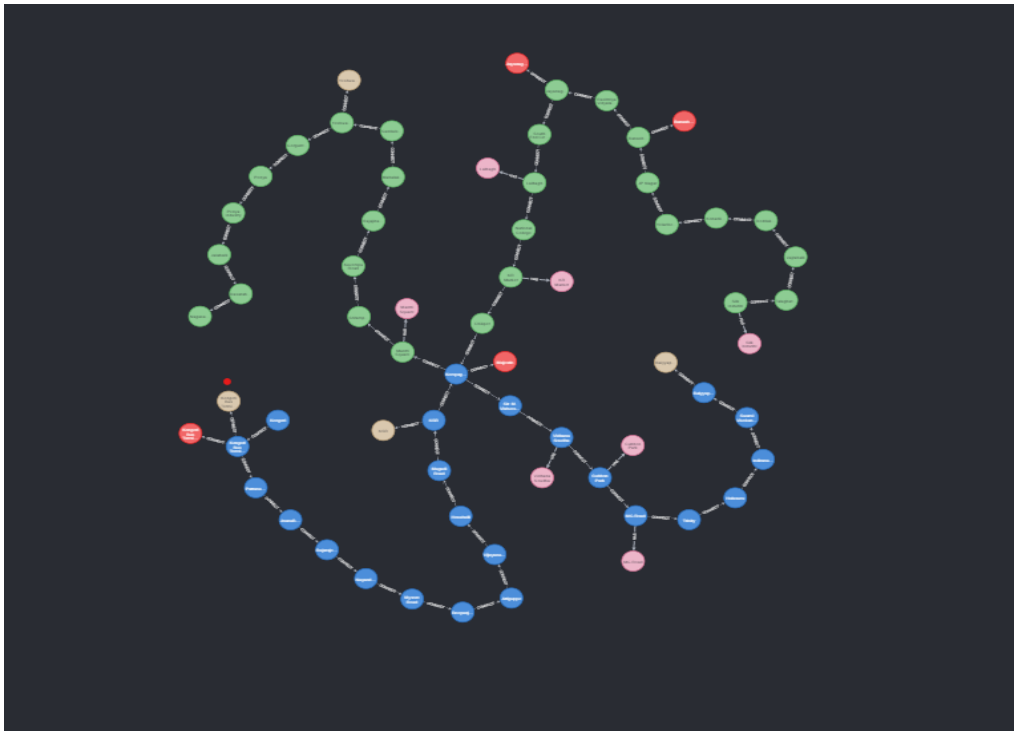
Creating a relationship **“HAS”** between a station and point of interest:

MATCH (s1:Station{name:'Lalbagh'}),(p1:PointOfInterest{closeTo: 'Lalbagh'})
CREATE (s1)-[r:HAS]->(p1)

Displaying the Database:

To view the database we use the command:

MATCH(n) RETURN(n)

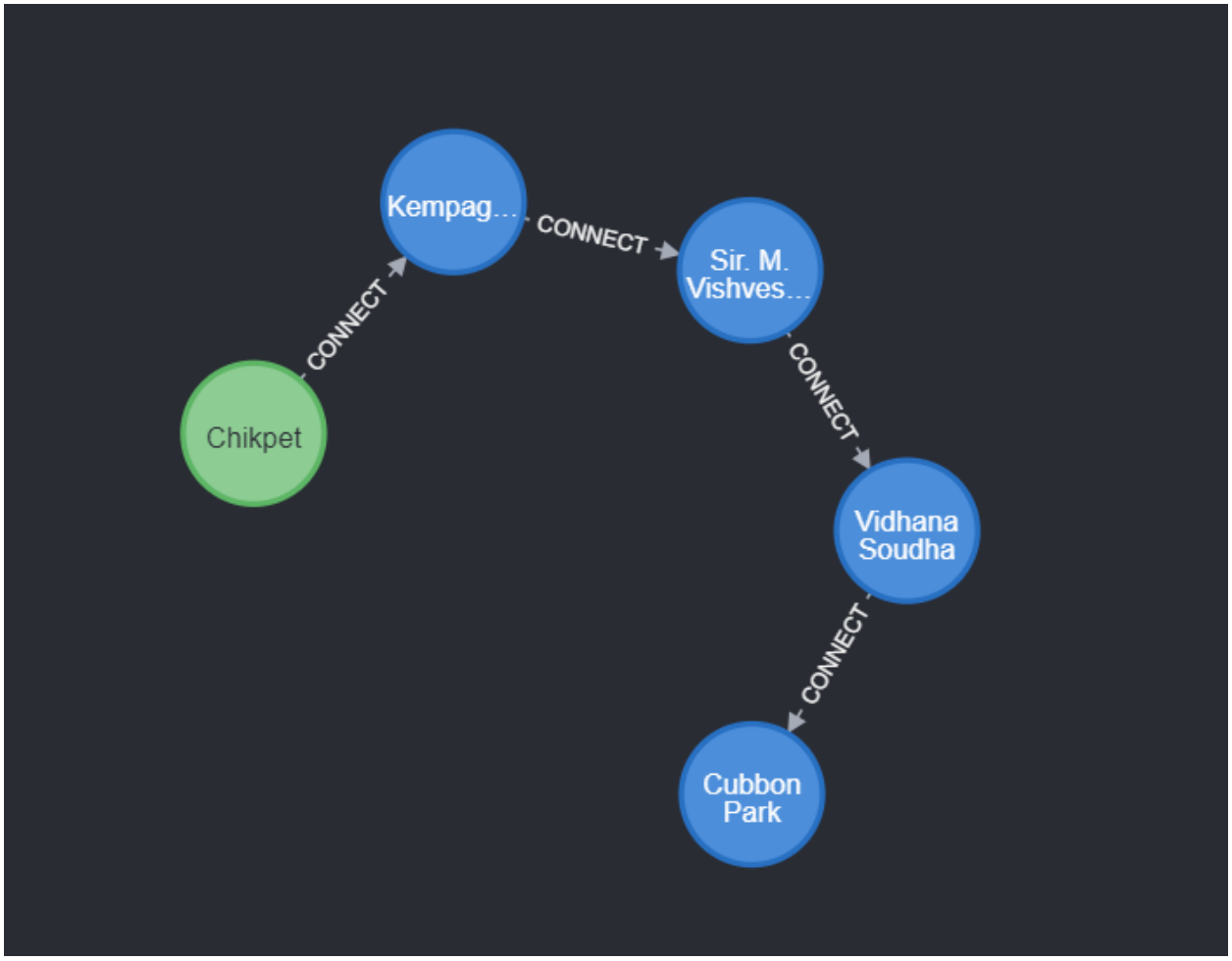


Querying the Database:

1) Finding the shortest path between two metro stations

```
MATCH
        (s1{name:"Cubbon Park"}),
        (s2{name:"Chikpet"}),
        p = shortestPath((s1)-[:CONNECT*]-(s2))
WHERE ALL (x IN RELATIONSHIPS(p) WHERE x.transport='metro')
RETURN NODES(p)
```

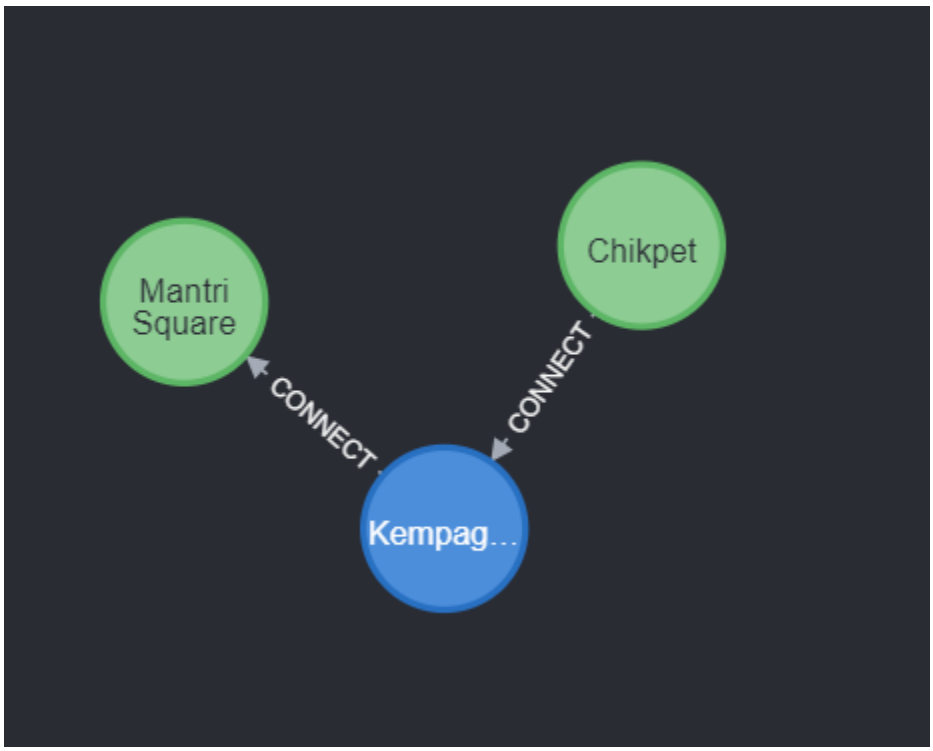
- *Here we recursively find the nodes with connect relationship from s1 until s2 is reached. This can be easily done using -[:connect*]-*



2) Finding the points of intersection of green and purple metros

MATCH(n)
where ((n)-[:CONNECT]->(:PurpleMetro) or (n)<-[:CONNECT]-(:PurpleMetro)) and ((n)-[:CONNECT]->(:GreenMetro) or (n)<-[:CONNECT]-(:GreenMetro))
return(n)

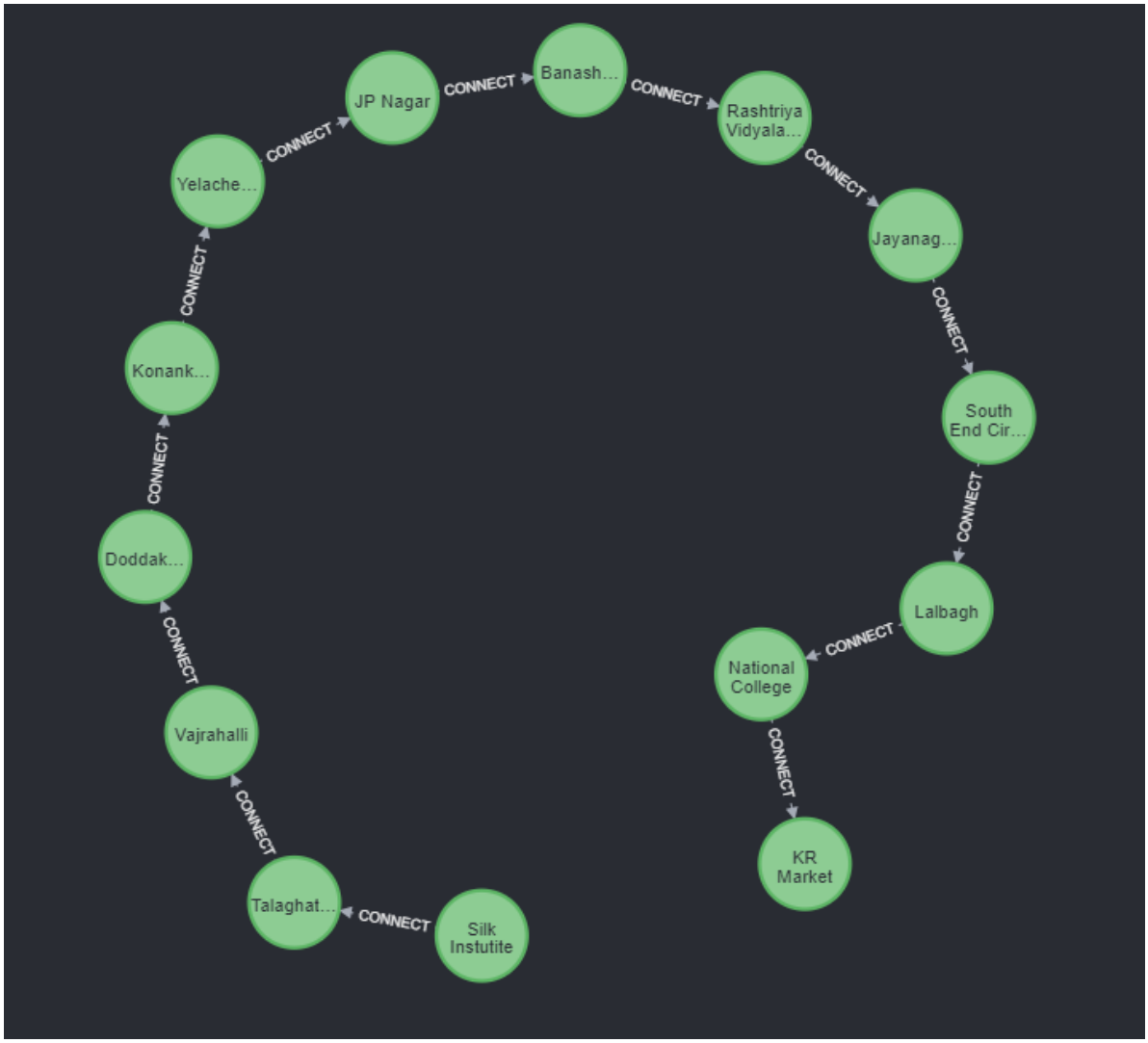
- *Here we check from nodes which is involved in relationship of type PurpleMetro connects GreenMetro or GreenMetro connects PurpleMetro*



3) Finding the shortest path between 2 point of interests

match(n),(m)
where (m)-[:HAS]->(:PointOfInterest{closeTo:"KR Market"}) and
(n)-[:HAS]->(:PointOfInterest{closeTo:"Silk Instutite"})
return shortestPath((m)-[:CONNECT]-(n))*

- *Here we first find the metro close to the point of interest and find the shortest path between those 2 metros using query (1)*



4) Finding all Purple Metro stations with both bus terminals and parking facilities

MATCH (n:PurpleMetro)
where n.hasParking = 1 and n.hasBusTerminal = 1
return n

- *We check for the nodes with 1 set for both properties of hasParking and hasBusTerminal in the PurpleMetro*