A Project Report

ON

GRAPH DATABASE INTERFACE BUILDING

BY

T.K. Prashanth

Tarun A.

Guide

Prof. Channa Bankapur

Designation:

Professor of Analysis and Design of Algorithms,

Department of Computer Science and Engineering,

(retired employee of Google),

PES Institute of Technology.

Bangalore.

January 2015 – May 2015

CERTIFICATE

Certified that the project work entitled **Graph Database Interface Building** is a bona fide work carried out by **T K Prashanth (1PI13CS175), Tarun A. (1PI13CS179)** in partial fulfillment for the award of degree of Bachelor of Engineering in Computer Science and Engineering of the Visvesvaraya Technological University, Belgaum during the academic semester January 2015 to May 2015.

Signature of the Guide

Prof. Channa Bankapur

Signature of the HOD

Prof. Nitin V Pujari

T K Prashanth
1PI13CS175

Tarun A.

1PI13CS179

INTRODUCTION:

This project was built to provide a developer interface for a graph database by using a Maven project to map items in the database to our good old Java objects. Graph databases , being non-relational have very powerful applications in areas where relational databases suffer in their mode of operation. We wanted to work on graph databases because graph databases are highly used in ontological projects and providing meaning to big data analytics reports on an interconnected data network.

In a world where social networking has become the breakfast, lunch and dinner of our generation, Neo4j has done well to provide the world with a concise, developer-friendly software to handle non-relational databases. This project was based on a road safety dataset of United Kingdom in 2011.We studied the property-graph model of our database and dealt with the innumerable problems of domain modelling by studying the manual and researching the software.

Since this was the first time we actually used eclipse to create a professional project, we learnt the working and production of a Maven project and also the usage of Spring APIs for linking the entities with our objects. From a bird's eye view, more than an implementation of an algorithm, we ended up performing extensive research on the entire Java – Maven – Spring – Neo4j framework.

FRAMEWORK:

Neo4J:

Neo4j is the leading implementation of a property graph database. It is written predominantly in Java and leverages a custom storage format and the facilities of the Java Transaction Architecture (JTA) to provide XA transactions. The Java

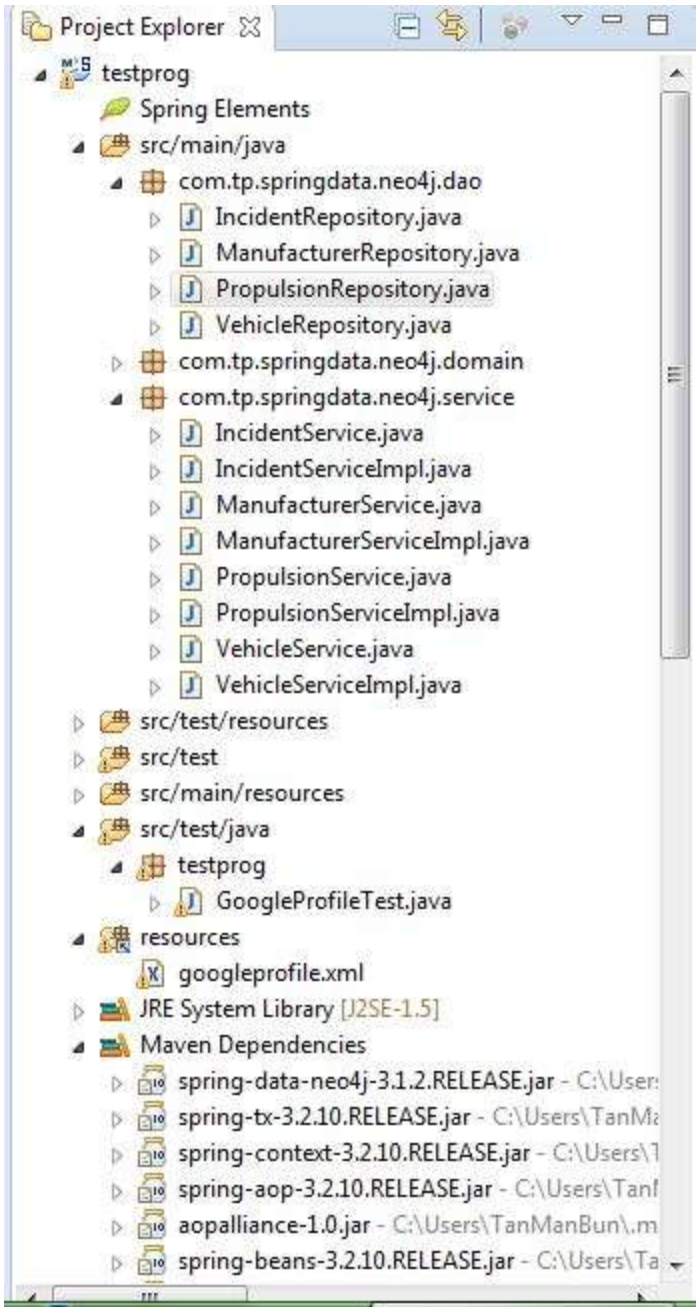
API offers an object-oriented way of working with the nodes and relationships of the graph. Traversals are expressed with a fluent API. Being a graph database, Neo4j offers a number of graph algorithms like shortest path, Dijkstra, or A* out of the box.

With the declarative Cypher query language, Neo4j makes it easier to get started for everyone who knows SQL from working with relational databases. Developers as well as operations and business users can run ad-hoc queries on the graph for a variety of use cases. Cypher draws its inspiration from a variety of sources: SQL, SparQL, ASCIIArt, and functional programming. The core concept is that the user describes the patterns to be matched in the graph and supplies starting points. The database engine then efficiently matches the given patterns across the graph, enabling users to define sophisticated queries like “find me all the customers who have friends who have recently bought similar products.” Like other query languages, it supports filtering, grouping, and paging. Cypher allows easy creation, deletion, update, and graph construction.

Spring:

Spring Data Neo4j was the original Spring Data project initiated by Rod Johnson and Emil Eifrem. It was developed in close collaboration with VMware and Neo Technology and offers Spring developers an easy and familiar way to interact with Neo4j. It intends to leverage the well-known annotation-based programming models with a tight integration in the Spring Framework ecosystem. As part of the Spring Data project, Spring Data Neo4j integrates both Spring Data Commons repositories as well as other common infrastructures. As in JPA, a few annotations on POJO (plain old Java object) entities and their fields provide the necessary metainformation for Spring Data Neo4j to map Java objects into graph elements. There are annotations for entities being backed by nodes (@NodeEntity) or relationships (@RelationshipEntity). Field annotations declare relationships to other entities (@RelatedTo), custom conversions, automatic indexing (@Indexed), or computed/derived values (@Query).

Maven Project Structure:



IMPLEMENTATION:

As mentioned before, this project ironically wasn't just a simple implementation of an algorithm with some input and output data and code which can be found in ample sites online. This was a research project aimed at understanding how meaningful algorithms work on graph elements in a graph database by using a Maven front-end for the database.

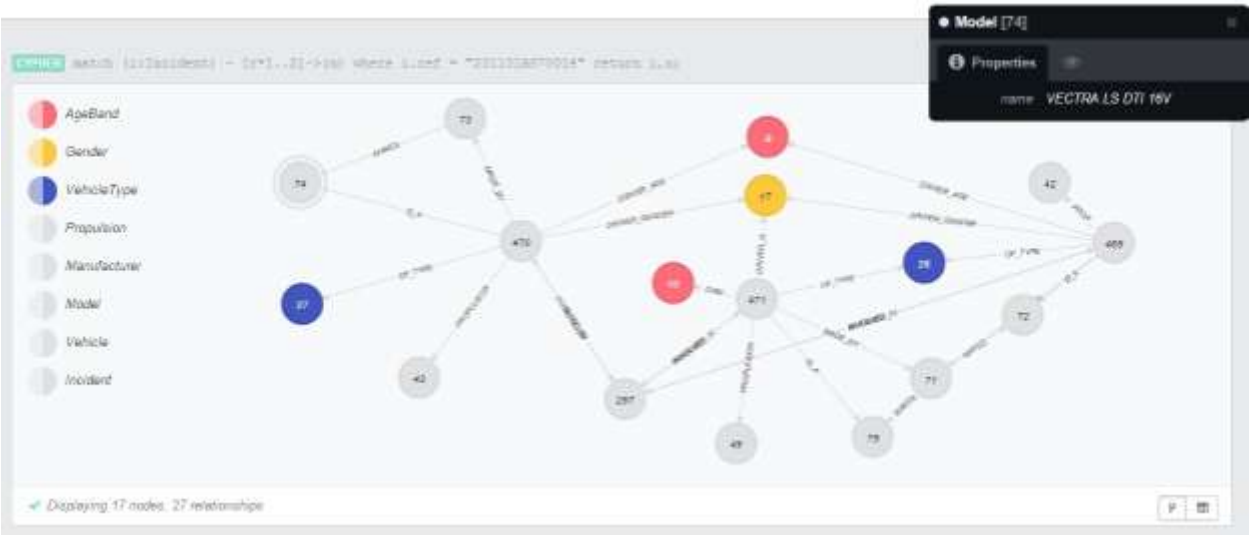
- We first mounted the "Road Safety dataset of United Kingdom in 2011" csv files as a graph database into Neo4J using the "LOAD AS CSV" command. There were 7 files which we had to import and dynamically link the attributes with different entities to provide the needed Node-Property setup for the dataset.
- After going through a lot of online recommendations, we decided that the best UI we could provide for a Neo4J project was through a Maven project. So we set up the infrastructure for a Spring project using Maven as the build system. We had to download, install and configure a lot of Maven dependencies for Spring (as seen in the pom.xml file).
- We then had to scan the whole database and come up with a set of domain classes for each entity in the database. To set up an environment we learnt that we had to use annotations to design the GraphEntity classes. [Domain path: com\tp\springdata\neo4j\domain]
- Then we provided a repository interface infrastructure for the basic functionalities of each of the domain classes. We resorted to the repository infrastructure because Spring removed the need to write boilerplate code for simple operations on each domain class.[Repository path: com\tp\springdata\neo4j\dao]
- We then developed service layer artifacts for the Repositories. We provided service component interfaces and their respective implementations for the DAO layer. [Services path: com\tp\springdata\neo4j\service]
- To run Spring based applications we needed to provide XML configurations for our entire project. The whole project had to defining xml files. One was pom.xml which set up the Maven build and the other was googleprofile.xml which configured the Spring part, in which I had to mention the namespace, Neo4j Schema definition (XSD file) ,database and the graph entities location, DAO interfaces base package, etc.
- Then I ran the test application to check if the database connection was solid (had to be done when the web browser connection port was turned off).
- After testing the basic CRUD operations, we decided that instead of running traversals and meaningless textbook graph algorithms that don't provide us meaningful data, we would run traversals that showed decent analytical results from the data.

- So instead of using the TraversalDescription class and display different traversals, I spent my time learning CQL – Cypher Query Language made specifically for graph databases so that I could perform analytics. In the long run, we may have lost our way as the project ended up being a database analytics project rather than an Analysis and Design of Algorithms project, but nevertheless it was done keeping in mind that this was an opportunity to do research and not a mere coding example to get my grades.
- Finally we ran cypher scripts to find out the most accident prone gender, age-band, vehicle model etc. and obtained the data in tabular form (as reading information in the tabular form is a more compact way of representing it).

RESULTS:

We got the required results in tabular form on the Maven console using the ExecutionEngine class to execute the commands and parse results. As seen from the analysis

1. Age-Bands of 36-45 were involved in the highest number of accidents closely followed by the band of 26-45.
2. Honda is the most accident prone manufacturer followed by Volkswagen.
3. Petrol propulsion engines are the propulsion systems leading to a lot of accidents while electric propulsion systems are not only eco-friendly but also the safest systems.
4. The 4^{th} command describes every accident that involved an AUDI made car.
5. This last command is the cornerstone of graph databases. It describes an entire accident to a graph traversal depth of 2 levels. It was an accident involving a Toyota Prius and Avensis taxi, and a Vauxhall Vectra car. All manufacturers, propulsion systems, vehicle makes, age bands etc. can be derived from the accident network shown in the web console. As you can see, if results came out in a relational format as shown in the console, the reference attribute is redundantly accessed.



CONCLUSION:

The advent of graph databases has affected the world in a dramatic way. From mapping spatial data, to building social networks, to describe chemical compounds and most of all, ontological analysis of data, we chose this project as I saw the potential of this software. The last Cypher command where we can derive all the factors involved in that one particular accident is enough to show the potential of this project. We didn't code out an algorithm in the conventional sense, we built an entire platform using third-party APIs to code ontological algorithms on a highly interconnected database. The most important part of this project was that I learnt how professional software projects are structured and built. My heartfelt gratitude goes out to Prof. Channa Bankapur for accommodating this non-conventional endeavour because I threw myself blindly into the topic as the concept captivated me, and I didn't want to make this about my CGPA, because as an engineer, I live to research and build, not earn and sleep.