Graph Query Portal

David Brock and Amit Dayal
Client: Prashant Chandrasekar
CS4624: Multimedia, Hypertext, and Information Access
Professor Edward A. Fox
Virginia Tech, Blacksburg, VA 24061
May 2, 2018
# Table of Contents

Table of Figures........................................................................................................ 3

**Introduction**........................................................................................................ 6  
Ontology .................................................................................................................. 6  
Semantic Web .......................................................................................................... 7  
Data Collection Platform ......................................................................................... 8  
Current Process ....................................................................................................... 9  

**Requirements**.................................................................................................... 10  
Project Deliverables ............................................................................................... 10  

**Design**................................................................................................................ 10  

**Implementations**............................................................................................... 18  

**Developer’s Manual**.......................................................................................... 21  
About ...................................................................................................................... 21  
System Requirements ............................................................................................ 21  
Node.js API ........................................................................................................... 24  
Neo4j Usage ........................................................................................................... 25  
Routes .................................................................................................................... 25  
Models ................................................................................................................... 27  
Controllers ............................................................................................................. 35  
Running the Node Server ....................................................................................... 46  
Accessing the Node Server .................................................................................... 47  
Running the Neo4j Service ..................................................................................... 48  
Cypher Data Import and Relationship Creation ..................................................... 49  

**User’s Manual** .................................................................................................... 55  
/:graphName/participant ......................................................................................... 55  
/:graphName/participant/:id .................................................................................. 55  
/graph/view ............................................................................................................. 56  
/graph/findByPropertyValue ..................................................................................... 56  
/graph/describe ....................................................................................................... 57  
/graph/addProperty ................................................................................................ 57
# Table of Figures

1. An illustration of an ontology ................................................................. 7
2. Semantic Web-Stack .............................................................................. 8
3. Friendica’s ‘wall’ feature ..................................................................... 9
4. Graph of all vocabularies ................................................................... 12
5. MySQL Workbench Reverse Engineering of Friendica’s Database .... 14
6. Version 1 of custom Ontology .............................................................. 15
7. Final Version of Ontology ................................................................... 16
8. Fictitious example of a graph interaction using Friendica’s Wall Post feature ................................................................. 16
9. Neo4j implementation of the interaction in Figure 8 ......................... 17
10. Api to Graph Database Diagram of Project ...................................... 19
11. Scenario and test queries .................................................................. 20
12. Root directory ..................................................................................... 22
13. Changes in config file ....................................................................... 22
14. Logging configuration ....................................................................... 23
15. Shell Enablement ............................................................................... 23
16. Putty configuration ........................................................................... 24
17. Single Route example ....................................................................... 26
18. Engagement Controller requirement .................................................. 26
19. Fetch Graph Function ....................................................................... 28
20. doesTypeObjectExist Function .......................................................... 28
21. getRegexForEngagementType Function ............................................. 29
22. findAnyNode Function ...................................................................... 29
23. findByPropertyValue Function ............................................................ 30
24. createNewProperty Function .............................................................. 30
25. createNewLabel Function ................................................................... 31
26. runRawQuery Function ..................................................................... 31
27. createNewRelationship Function ....................................................... 32
28. Participant Model ............................................................................... 32
29. Multiple Participant Query ................................................................. 33
30. Engagement Model Class .................................................................. 33
31. Comparative Model Class .................................................................. 34
32. Graph Controller Class ...................................................................... 36
33. findNode Controller processing .......................................................... 37
34. findProperttyValue Controller processing ......................................... 38
35. createNewLabel Controller processing ................................................. 38
36. createNewLabel Controller processing .............................................. 39
37. createNewRelationship Controller processing ................................... 39
38. participantController class ............................................................... 42
39. engagementController class .............................................................. 43
40. comparativeController class
41. logController class
42. HTTP request for participant
43. Sample error message
44. Neo4j browser command line interface
Executive Summary

Prashant Chandrasekar, a lead developer for the Social Interactome project, has tasked the team with creating a graph representation of the data collected from the social networks involved in that project. The data is currently stored in a MySQL database. The client requested that the graph database be Cayley, but after a literature review, Neo4j was chosen. The reasons for this shift will be explained in the design section.

Secondarily, the team was tasked with coming up with three scenarios in which the researchers’ queries would fit. The scenarios that were chosen were the following: Single Participant query (give me all the information about person X), Comparative Study query (let me compare group x with group y), and Summarization query (let me see all the engagements that exist in the network).

These scenarios would be used to form an API that used Node.js. Queries executed within the API would be cached in a Mongo Database in order to save time. The API handles fetching data from the graph, conducting multiple queries to synthesize a comparative study, updating nodes in the graph, and creating nodes in the graph.

The developers migrated the data from a MySQL database to Neo4j. Then, the team created an API endpoint using Node.js that researchers can use to conduct simple repeatable queries of the database. The procedure for this is discussed in the user manual. Finally, the team created a front-end website to further streamline the steps they will have to take. The team delivered to the client a graph-based database, a user and developer manual, a graph interface, a graph querying interface (the API), logging capabilities, and a front-end. The team learned lessons along the way, mostly relating to milestone scheduling and leaving a buffer of time for when things go wrong. They also learned how to work with Neo4j and Node.js.
Introduction

The Social Interactome is a research project that has a goal of using a social network to help people recover from substance abuse [1]. The participants of the project are paid volunteers who are undergoing recovery from substance abuse and have answered surveys and questionnaires as well as participated in other social interactions with other participants. The project, supported by the International Quit & Recovery Registry, is funded by the National Institutes of Health [1]. Dr. Warren Bickel, Director of the Addiction Recovery Center, leads a team of researchers from the Virginia Tech Carilion Research Institute and from Virginia Tech (Department of Statistics, Department of Computer Science). They have been tasked to run experiments and mine the data collected from the Social Interactome project [1].

The client, Prashant Chandrasekar, is a lead developer for the social network and has aided the investigation in creating a graph representation of the data collected from the social networks. Prashant is a third year Ph.D. student at Virginia Tech who is supervised by Dr. Edward A. Fox. His primary research is in Digital Libraries as well as Natural Language Processing and Data Mining. Through Prashant’s extensive experience, he has helped the team in key decisions throughout this project and has aided in identifying technologies and goals that are relevant to this report and the project overall.

The goal of this capstone project is to create a graph representation of the data from the Social Interactome project. Unfortunately, the Social Interactome data is currently stored in an relational database, so to convert to a Graph Database (Graph DB) representation was one of the team’s challenges. The graph database is derived from the Semantic Web and is integral to understanding the team’s design decisions down the line.

Ontology

According to Guarino, “Computational ontologies are a means to formally model the structure of a system” [2]. An ontology represents the knowledge of concepts in a system and the relationships between those concepts [2]. Two standards which govern how ontologies are represented are Resource Description Framework (RDF) and Web Ontology Language (OWL). In RDF, there are two main components, classes and relationships. Classes are represented by ovals, and relationships by arrows. The combination of two classes and the relationship between them is called a triple. A triple contains a subject, predicate, and an object. Ontologies are an alternative to source code. They use models, which allows them to be easily extensible.
The Semantic Web is a World Wide Web Consortium (W3C) standard that promotes a common data format for all applications and webpages around the world. The Resource Description Format (RDF) allows developers to create namespaces or use existing ones in order to richly describe information on a webpage [2]. This, in turn, allows for machine-readable content to be accessed quickly over the World Wide Web [3]. To create an RDF representation, we have to come up with an ontology that best represents the team’s data. In an RDF format, the ontology will dictate the different namespaces required for documentation of the application. In a triplet format, the ontology will identify the different combinations of a ‘subject’, ‘predicate’ and ‘object’. This is shown in Figure 1. By using this format, we can identify information in a graph more easily and describe the team’s linked-data with more predicates. In order to push for the Semantic Web as a standard, W3C has created the Semantic Web stack [4]. Figure 2 shows the different components necessary to create a well-formed linked-data application.
In this stack, it should be noted that RDF is not the only established format, but there are others such as: ‘N-Triples’, ‘Turtle’, ‘Rule Interchange Format (RIF)’, etc. Similarly, the Web Ontology Language (OWL) is not the only available ontology as a user can create their own or use a combination of existing ontologies from different sources. Lastly, the SPARQL query language is also not the only query language but can be exchanged for other languages such as Cypher, GraphQL, etc., based on other technology stacks (as later discussed) [5].

The difficulty in enforcing an ontology is that describing the data for all applications requires different formats with different namespaces. This has created a problem in consistency throughout different Semantic Web-based applications [6]. There is also the possibility of deceitfulness when it comes to describing data. There is implicit trust by the user of the application, but in the event that the developer has ulterior motives, this system falls apart.

**Data Collection Platform**

The data collection platform, Friendica, is based on Facebook.com™’s social media platform. Friendica, according to their product website, is “a decentralized social network” that allows for “relationships [to be] made across any compatible system, creating a network of internet scale made up of smaller sites” [7]. The website is based on Hypertext Preprocessor (PHP) and uses MySQL as the database technology [7]. For the purpose of this project, data collection has
already taken place. All the relevant data has been stored in 4 separate databases. Due to the nature of privacy of the data, this report will not show any data, but it can be assumed that the data was collected without any problems.

![Figure 3: Friendica’s ‘wall’ feature](image)

Friendica allows for different types of interactions including: wall posts (see Figure 3), wall comments, activity posts (e.g., user takes a survey and it will be posted on their wall), likes, shares, private messaging, and a mail system.

**Current Process**

The data is represented in a MySQL database. The current process of querying is inefficient.

1. The researcher has a question he wants answered.
2. The researcher asks the developer to query the database for data related to that question.
3. Developer gives the researcher that data.
4. The researcher typically wants to refine the data, so they ask the developer for a more specific query.
5. Developer complies and gives the researcher that data.
6. Steps 4 and 5 can go on multiple times.

This cycle can become more efficient if the time spent communicating with the developer to reiterate smaller goals is reduced. The client wanted to improve this sequence by providing a different approach to querying the data. Graph based databases provide a better way to represent a social network compared with MySQL databases. This is because a participant can be a node in a graph based database, and all their interactions can be edges. This makes the querying more
user friendly. By streamlining the most commonly asked questions using an API, researchers can lessen the time spent generating queries.

**Requirements**

**Project Deliverables**

The client has requested four deliverables to be produced by the end of development and testing.

**Deliverable One**

To increase the extensibility of the project, Prashant has asked the team to write a developer and a user manual. The developer manual will provide instructions for a future developer to be able to set up their own instance of Neo4J and Node.js. It will also provide instructions for how to transfer the data from SQL to Neo4J. Lastly, it will contain instructions for how to continue to add APIs. The user manual will allow an average person to understand how to query the API and understand the results. It will contain step-by-step instructions for how to access the API, how to send an HTTP request, and how to understand the result.

**Deliverable Two**

The client has solicited that we convert the data from a MySQL representation to a rich semantic web representation.

**Deliverable Three**

The team will produce a graph interface that allows the client to add annotations and/or to extend the functionality of the semantic web representation.

**Deliverable Four**

The client has asked for a graph querying interface to conduct specific analyses of studies, which are stored in a graph form.

**Design**

During the team’s first meeting, Prashant set up six requirements for technology that would replace MySQL. Those requirements were the following:

- The technology must allow for an import of the data from MySQL.
• It must allow the user to create an API.
• It must be able to Modify and Add relationships on the fly.
• It must have support for GraphQL.
• It must be able to return relevant subgraphs given a query.
• It must be able to be easily replicable.

The team discussed three possible technologies that would fulfill these requirements. The technologies were Apache Jena, Cayley, and Neo4j. Outlined below were the notes that the team took on each technology after conducting a literature review:

1. Apache Jena [8]
   a. Allows for an import of data using an RDF representation
      i. There exists a tool that allowed a conversion of the MySQL database to RDF [9].
   b. Supported an HTTP endpoint through the Fuseki API
      i. Used the querying language SPARQL
      ii. Allowed for updates and addition to the model
   c. Allows inference using predefined rules
   d. Output results in subgraph using the RDF format
      i. These results could be loaded back and searched.
   e. Code only supports Java’s Fuseki API
      i. No support for any other querying language

2. Cayley [10]
   a. Allowed for the import of data using a quad file
      i. A quad file contains a subject, predicate, object, and label for each node.
   b. Supported multiple languages
      i. The team had to custom convert n-triples to n-quad format.
      ii. The team was unsure if it was possible to automate, based on a forum post,
          1. Showed it required lots of work to maintain
   c. Allowed modifications through update statements
   d. Limited online support and documentation

   a. Allowed for an import of the data directly from MySQL [11]
   b. Main querying language was Cypher
      i. Libraries allowed GraphQL to translate queries into Cypher
   c. Allowed for modification of graph using mutation
   d. Uses property containers, which function differently than an RDF model
      i. Must convert data to RDF triples and import into Jena to use reasoners for inferencing
   e. Allowed for the return of a subgraph
      i. Could format the data (e.g., JSON)
f. Extensive online support and documentation

Initially there was a push for Cayley because it was the newest of the graph database technologies and had notable ex-Google employees as the creators. Cayley is a graph database written in the ‘Go’ language that allows for graphs for Linked Data [10]. Cayley supports multiple query languages, specifically a GraphQL-inspired language as well as other newer graph query languages. Cayley’s development is based on the community and thus is heavily dependent on users to collaborate to generate support for any questions or comments regarding its functionality. Cayley stores data using an n-quads format, which is slightly different from the n-triples format. Furthermore, Cayley only supports four well-known RDF vocabularies, which is not sufficient to represent the project’s data.

Looking into the web of different RDF namespaces, it becomes clear that there is very little consistency for each. This makes it much harder to pick specific vocabularies for the project and use them effectively.

There are thousands of different vocabularies, as seen in Figure 4, so to pick the right vocabularies for the work would have been a tedious task. We were left with the choice of either
creating a new namespace and adding to the blob or using a different format. Since the namespace would be extremely specific to the team’s research, we decided against creating a custom namespace. The data is not portable outside this research for privacy reasons, so we decided that using a different technology other than RDF would be a better choice for this situation. Cayley only supports 4 of the hundreds of ontologies but requires that each node (as defined by a subject or object) in the graph has a ‘weight’ value. As mentioned previously, a n-triplet consists of a ‘subject’, ‘predicate’, and ‘object’. The ‘predicate’ describes the relationship between the ‘subject’ and ‘object’. For example, “Prashant is a researcher” can be broken into: ‘Prashant’ as the ‘subject’, ‘is a’ as the predicate, and ‘researcher’ as the object. Cayley’s n-quad requires a 4th relationship that shows the weight of an object in relation to other objects that are related to this specific subject. This was not explicit in the data, which made it difficult to use effectively. It would be possible to assign the weight of all objects to a uniform value, such as 1, but this would render the n-quad format useless. Furthermore, since it was not possible to effectively create the RDF using the 4 supported vocabularies, it became apparent that Cayley was not the right tool for this project.

The next option was to look into Java’s Apache Jena, which is a well-used and supported graph database technology. Since everyone on the team had extensive (2 or more years) experience with Java, it became a very appealing option. Apache Jena also had very good documentation and had much developer support that Cayley did not have [8]. But like Cayley, Jena only had support for RDF [10] and as discussed earlier, this would be a daunting task to convert all the data into an RDF format rather than a generic n-triplet format. Furthermore, Jena came with an out-of-the-box API for the graph database called ‘Fuseki [10]’. The only problem was that Jena only supported SPARQL as the primary query language, which violated one of the goals. Jena also did not have support for any other API technology, which limited the option of using a different middleware [12]. With Jena’s support for only RDF and SPARQL as well as Fuseki as the only API endpoint technology, Apache Jena became a good back-up option but not the best option for this project. This left the team with Neo4j as the primary choice of graph database.

Neo4j is a self-contained graph database tool that has a well-backed developer community as well as dedicated support agents to help developers on specific projects [11]. Neo4j primarily uses Cypher as its own graph query language but also has native support to translate GraphQL to Cypher. Neo4j does not adhere to an RDF standard [11]. While neo4j does support RDF as a standard for graphs, it primarily works with a self-defined n-triplet format that specifies a custom ‘subject’, ‘predicate’, and ‘object’. This allowed the team to be more expressive with a custom triplet set. Neo4j also allows data to be imported using a comma-separated file format (CSV), which allows the team to bulk export the SQL database in relevant CSV documents [11]. Neo4j has middleware support for most well-known languages such as JavaScript (Node.JS) [12], Python, Java, etc. Neo4j also allows schemas to be defined in Java, which would be a good tool for future developers who want to quickly develop data models for custom imports without
learning extensive Cypher queries. With all these advantages, it seemed that Neo4j was the optimal tool for this project and that the graph developer could work concurrently with the API developer, which would allow the team to finish the work quickly and thus fulfill the deadlines on-time.

The literature review showed that regardless of which choice the team made, the team still had to write the ontology from scratch neither Cayley nor Apache Jena could select the best vocabulary for the graph’s ontology. Neo4j allowed code to define classes, and the ontology was automatically created.

Since we selected Neo4j as our primary graph database tool, we had to create a custom ontology to represent the data. A naive approach was to treat every table in the database as an object. This became a tedious task as there were more than 40 tables in the database.

As Figure 5 shows, Friendica’s database is quite cluttered and has different data stored across different tables. The initial approach is to identify relevant data and find the relevant tables based
on those queries. The issue with this approach is that the custom ontology also became cluttered extremely quickly.

Figure 6 shows a naive implementation of the ontology, but the data is extremely cluttered and there are not many relationships between nodes. This would make it very difficult to search the graph effectively. The team also had drawn false conclusions based on the database design, which led to flaws in this ontology (Figure 6). In designing this custom ontology, the team learned how to deduce important information from the database and attempt to combine objects together despite having the information stored on different tables across the database. The team decided that instead of using the tables of the database as objects in the ontology, it would be better to categorize all ‘private conversations’, ‘wall posts’, ‘wall comments’, ‘shares’, ‘likes’, and other social interactions in an object called a ‘digital object’. This would allow us to generalize all the data in a common object and create meaningful relationships between other users in the graph rather than having information loosely connected in unrelated tables-objects.
In the final version of the custom ontology (Figure 7), we labeled all social interactions with other users of the social platform as a ‘digital object’. This allowed the team to categorize the digital objects as different types of interactions but kept the data in a consistent format. Using this graph allows a potential researcher to ask introspective questions such as ‘Users who used drug ‘x’ and who are friends with each other, how did they interact with each other?’ These questions can be extremely complicated but will be extremely easy to query the database. To model a use-case of this graph, the team created a fictitious sample graph interaction that modeled how a user communicates with another user using a wall post and a wall-comment.

Figure 8: Fictitious example of a graph interaction using Friendica’s Wall Post feature
The dialogue for the conversation in Figure 8 looks like this:

- Person A: “Hey Daniel! How’s it going?” -(post) → Person B’s wall
- Person B - (likes) → Person A’s Post
- Person B: “It’s not Daniel! It’s David” -(comment) → Person A’s original post

In the interaction above, it is clear that the predicate listed in parentheses shows the relationship between how the data is exchanged between person A and person B. The graph also stores tertiary information such as a timestamp, parent post, post type, etc. This example was then ported to Neo4j, which created a graph interaction as in Figure 9.

In Figure 9, a developer re-created the interaction using Neo4j. The different nodes represent the different objects as described in Figures 7 and 8. As previously noted, ‘person A’ in Figure 9 is ‘Amit’ and ‘person B’ is ‘David’. ‘Amit’ has an account and has also created a wall post. ‘David’ also has an account (not shown), and also ‘likes’ Amit’s wall post while he also creates a comment on the wall post. Since the comment is in response to the wall post, the comment created by David has a parent that is Amit’s comment. Vice-versa, Amit’s post has a child that is David’s comment.

This simple example is multiplied in complexity with the addition of all the different interactions of all the participants across multiple modes of communication over Friendica. Figure 7 has primarily five different objects: Account, Profile, Gender, Participant, Digital Object. The different objects are determined by the commonalities in categorical datasets across the database. The other information is stored as literals (as marked by a dotted-rectangular shape in Figure 7).
Each literal can be a string or a numerical representation of the data (e.g., epoch timestamp, login data statistics, etc.).

Using the schema as the primary ontology for Neo4j, the team had to decide how to export the data from the database while adhering to the graph in Figure 7. This required some specialized skills in using the structured query language (SQL) to export the database in suitable CSV files that not only maintained data integrity and adherence to the structure as shown in Figure 7, but also contained data that purposefully hid any self-identifying information in accordance with the Institutional Review Board (IRB) at Virginia Tech.

**Implementations**

The scenarios that the team decided to test were the single participant scenario, the summarization scenario, and the comparative studies scenario. The API endpoint for the single participant query will be at /participant. The API endpoint for the summarization query will be /engagement/{item} where item is the engagement item that the researcher would like summarized. The API endpoint for the comparative studies query will be /multiple. In the body of the HTTP request for the comparative studies query will be the items that the researcher wants to compare. The middleware will translate these into separate queries and return the results. Based on the type of HTTP request, a write or a read will occur.
Looking at Figure 10, the Node.JS API will be the middleware that will translate custom queries from the researcher’s input and translate them into pre-formed Cypher queries to run on Neo4j. Except for the Open-ended scenario, all other API endpoints will be pre-defined and will require a parser to read a researcher’s query based on filled-in parameters that the research will provide in a JavaScript Object Notation (JSON) object. The API will then fill in the proper parameters in predefined Cypher queries to then run on the graph. To connect to the graph, the Node.JS service will connect to Neo4j’s Bolt interface. Bolt is the connection technology for Neo4j and is provided by Neo4j as part of the graph database application [11]. In Figure 10, the graph database is like a black box. A user can see the input and output of the service, but the details are obfuscated. This is a deliberate decision, as the graph, which is stored in a custom n-triplet format, is not in a readable format for users and thus should only be accessed by the API. In case the user wants to run custom read-only queries, the user can use the ‘open-ended’ API to fulfill their request.

To implement the graph database, a developer must follow several steps to identify the best model and representation of the database and its data. As previously discussed in the design section, the team generated a custom ontology for this project. By modeling the data using this
ontology, the team then created custom SQL queries to export the data into a CSV format. There are at least 7 different queries that were used to transform the relevant datasets into CSV files, and required non-standard SQL statements to generate, due to the confusing format of the database. Once the data was exported, the developer used Cypher to import the relevant CSV files into neo4j. Cypher is able to create objects based on the headers of the CSV files, which made it extremely easy to create the relationship models based on the ontology (Figure 7). Once the data was imported, the developer created relationships within the graph database based on the categories as previously explained in the design portion of this report. Once the graph and the relevant API were created, the developer connected the API to the graph database and thus completed the data transaction from the API to the graph database.

To test this application, the team developed the following use-case model, given in Figure 11.

Figure 10 illustrates the different APIs as questions that we were able to translate into a pipeline from Figure 11. We can see in Figure 10 the possible APIs that we could consume to answer the questions for single and group participant queries from Figure 11. Each query can be executed on the graph using one or more of the APIs. To test the team’s implementation, the API was used to pull out information about single and group participant data, which was then compared to the results obtained from SQL. Once the client was satisfied with the results, we repeated this process for all other scenarios. This also allowed the team to identify any data or API issues before releasing this product for researchers to test.
Developer’s Manual

About

This manual is meant for developers to understand how to create application programming interfaces (API), set up their Neo4j environment, and extend functionality for the Social Interactome project. The APIs allow researchers to access more data from the Neo4j graph database. This manual assumes that the developer has at least some programming experience in functional programming and object-oriented programming. This manual also assumes that the user is an administrator of (or a person with equivalent access to) the system they are running.

The developer’s manual extensively covers the Node.js API and Neo4j tools that will allow a developer to understand the existing code, and how to extend that code as they see fit. By using this manual, the developer can add new API endpoints and modify and delete current API endpoints. To see a list of endpoints available, please refer to the User’s Manual. Apart from the API, this developer’s manual will also introduce Neo4j’s Cypher as the primary graph query language. By using Cypher, a developer can change data stored on the graph database, create or delete relationships between nodes, or import more data to build the graph further.

System Requirements

For the purposes of this project, we were given a 64-bit CentOS (Linux) 6-core machine. Although other configurations for Neo4j and Node.js are possible, the developer’s manual will focus on a Linux environment for all development. The manual assumes that the developer is an administrator. This is required, as many tools need to be installed and run with elevated privileges.

First, install Git on the system. Please follow the steps here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Second, install NPM (Node Package Manager) and Node.js on the system. Please follow the steps here: https://nodejs.org/en/download/package-manager/

Third, install Neo4j. Please follow the steps here: https://neo4j.com/docs/operations-manual/current/installation/linux/tarball/. Since Neo4j is a Java-based graph database, some tools to run the Java Runtime Environment are also required. Please follow all the steps on Neo4j’s website for installation for further information. Once Neo4j has been installed correctly, please open the configuration file and set the following configurations (can be found at: “/etc/neo4j/neo4j.conf”):
Figure 12: Root directory

Figure 12 shows the root directory. Set the import directory to a root-readable directory. This is where CSV files are stored before being imported into Neo4j.

```conf
# This setting constrains all 'LOAD CSV' import files to be under the 'import' directory. Remove or comment it out to
# allow files to be loaded from anywhere in the filesystem; this introduces possible security problems. See the
# 'LOAD CSV' section of the manual for details.

dbms.directories.import=root:/dataFor CSTeam
```

Figure 13: Changes in config file

Please make the following changes to the neo4j.conf file shown in Figure 13:

- Set the default listen address to `localhost`, which is 127.0.0.1. If localhost is mapped to another address, please type that address here.
- Set the advertised address to the address of the machine running the neo4j instance.
- Make sure Bolt is enabled and is running on a non-used port on localhost.
- Make sure the HTTP connector is enabled and is running on a non-used port on localhost.
- (Optional) Enable the HTTPS connector
For development purposes, please enable logging as well. This can be turned off in a production environment. Our settings are shown in Figure 14.

```
# Logging configuration

# To enable HTTP logging, uncomment this line
dbms.logs.http.enabled=true

# Number of HTTP logs to keep.
dbms.logs.http.rotation.keep_number=5

# Size of each HTTP log that is kept.
dbms.logs.http.rotation.size=20m

# To enable GC Logging, uncomment this line
dbms.logs.gc.enabled=true
```

Figure 14: Logging configuration

Please enable the shell for Neo4j. This will allow shell access to neo4j’s console which is required for the API to work as well as allows extensibility of the API. Our settings are shown in Figure 15.

```
# Enable a remote shell server which Neo4j Shell clients can log in to.
dbms.shell.enabled=true

# The network interface IP the shell will listen on (use 0.0.0.0 for all interfaces).
dbms.shell.host=127.0.0.1

# The port the shell will listen on, default is 1337.
dbms.shell.port=1337
```

Figure 15: Shell enablement

For all other options, please leave them as is. This project’s configuration file is available on the project’s github and can be used to revert or replace the developer’s version of the configuration for faster development. To find the configuration file as well as other files mentioned in the User and Developer’s manuals, please download the ‘GraphQueryPortal-master.zip’ file from VTechWorks. The configuration file for this project is in the root directory of ‘GraphQueryResearch-master’ folder.

Since the data carries sensitive information, the project’s developers have chosen to keep all ports on this server closed and instead use a tunnel over a secure shell to access the server and graph database on localhost. From the project developer’s local machine, we installed PuTTY to fulfill this task. Furthermore, the instructions given to the reader will assume that the reader is also on the Virginia Tech network. If not, the developer must VPN into Virginia Tech using the VPN client application: Pulse. Directions to install and use that can be found here: https://vt4help.service-now.com/sp?id=kb_article&sys_id=d5496fca0f8b4200d3254b9ce1050ee5.
For PuTTY, please use the following setup:

![PuTTY Configuration](image)

Figure 16: Putty configuration

Note that L5001, L5003, L5005, and L5007 refer to localhost ports available on the developer’s personal machine. 127.0.0.1 is localhost which belongs to the server (host), ‘friendica.cc.vt.edu’. Port 7474 belongs to Neo4j; port 7687 belongs to the Neo4j server connector, Bolt; port 7473 is the HTTPS connection for Neo4j; and port 3001 is the Node.js service. These ports can be changed depending on the neo4j.conf file for Neo4j’s service. Figure 16 shows the putty configuration used for this project.

Fourth, install MongoDB. Please follow the steps here: [https://docs.mongodb.com/manual/tutorial/install-mongodb-on-linux/](https://docs.mongodb.com/manual/tutorial/install-mongodb-on-linux/).

Once all these tools have been installed and are functional, please clone the code from the repository, which can be found here: [https://github.com/adayal/GraphQueryResearch](https://github.com/adayal/GraphQueryResearch).

Node.js API

Node.js is a JavaScript runtime event-driven, non-blocking I/O, lightweight server that allows developers to create and use packages using the Node Package Manager (NPM) to create open-source libraries for further development. As its name suggests, Node.js is built on the JavaScript language and requires experience in JavaScript semantics. In particular, it is important to adhere to the non-blocking principles by employing the use of callbacks. Thanks to the ECMAScript6 (ES6), traditional object-oriented developers can also now develop on JavaScript using a unique
class-design. In this project, we employ ‘babel’ to ‘transpile’ the ES6 to ES5 for backwards compatibility on older Node runtime environments. ES6 allows us to use many new features such as ‘promises’, lambda arrow functions, classes, local variables, etc. The current code base employs all these new ES6 functions, and it is important to understand their structure before implementation. To help in that process, comments have been placed in the code to further explain the internal steps of functions.

This project use Node.js’ ‘Express’ package to create an API using custom routes or user-accessible endpoints. In our API, we assumed that each endpoint is stateless. This means that for every subsequent API call, we do not hold an ‘object’ or a current ‘state’ for the requests. In order to enforce that, our development uses a class design strictly using ‘static’ methods.

This project also uses ‘Mongo Database’ (mongo), which is a lightweight NoSQL (Not only Structured Query Language) database. For this project, this database is mainly used to store logging information. This can be useful to backtrace errors and show the input and output for both user and developer API requests. The system requirements section includes steps on how to install this database.

In order to maintain high code reuse and code coupling, our project employs a MVC-like pattern for code maintenance. Since this is just an API server, we do not employ ‘views’, but instead only models and controllers. By using this pattern, we can divide up the code in logical sections, which helps in reading code, finding and debugging errors, and extending new features. More information about the model-controller (MC) pattern for this project can be found in later sections of the developer’s manual.

**Neo4j Usage**

In order to understand some of the scripts and commands created for the model classes of the Node.js project, it is important to understand what exactly Neo4j is and how it works. Neo4j is a Java-based graph database that allows users to import/create data and create relationships between related pieces of information. These relationships are defined using an ontology, which is a flexible representation of the information on the graph. Neo4j’s primary query language is called ‘Cypher’, which gives a developer access to create, read, update and delete (CRUD) data in the graph database. Coupled with the Neo4j Cypher user manual, as well as a strongly-backed open-source developer community, Neo4j is the optimal graph database tool for this project.

**Routes**
To start out, please take a look at the routes.js file (located in ./src/routes/). This file contains all the endpoint routes for our API. Each route follows a general pattern shown in Figure 17.

```javascript
app.route('/:graphNAME/engagement/:engagementTYPE').get(engagementController.fetchEngagementDetails)
```

Figure 17: Single Route example

Breaking this down:
- ‘app.route’ tells the express package that this is a new endpoint
- ‘/:graphNAME/engagement/:engagementTYPE’ is the endpoint itself
  - Using a ‘:’ or other special symbols such as ‘+’ will match a URL endpoint with a regex
  - It is important to remember that there can only be unique regex API endpoints. Having two APIs with the same regex pattern will cause an unexpected error.
- `.get()`
  - This is the HTTP method that will be used on this API. Similarly, the developer can use .post(), .put(), .delete(), etc. for different methods. This allows method overloading for the same HTTP endpoint. This method passes a request and response object to the callback. More information can be found here: https://expressjs.com/en/4x/api.html#app.get
- engagementController
  - This is the controller for our engagements (like, post, share, story, assessments, etc.). Effectively, this controls all methods related to an engagement from Neo4j’s standpoint.
- `.fetchEngagementDetails`
  - This actually calls the method from the controller. This function acts as a callback for the .get() HTTP method.

The routes call methods in the controller only. The controller will process the information and send it to the model or another controller depending on the request. It is important to not call the model directly as information that comes from the routes is not sanitized. To import a controller, use the following format shown in Figure 18.

```javascript
var engagementController = require('./controllers/engagementController');
```

Figure 18: Engagement Controller

Since the response object is sent to the controller’s method, it is important to send a response from the controller. Not sending a response will cause the client to hang, which would be bad as a usability point-of-view. Longer running operations might require the connection to hang as well, so it is crucial to use a non-blocking pattern when implementing new functions.
For a complete view of all current APIs available, please refer to the User Manual documentation.

Models

As mentioned previously, a model class is meant to model create, read, update, delete (CRUD) operations for Neo4j. Each class models a specific part of the graph database by using a series of static functions. In the current codebase, there are 3 models: graph, participant, and engagement. As suggested from their names, the graph model manages all general interactions with the graph (e.g., return all nodes matching keywords); the participant model manages all interactions with the node-types (label) PARTICIPANT, ACCOUNT, DIGITAL_OBJECT and GENDER; the engagement model manages all interactions with the node-type (label) ENGAGEMENT, specifically pertaining to likes, stories, posts, shares, and assessments.

To connect to Neo4j using the dedicated Bolt connector, our codebase uses a package called: ‘neo4j-driver’ (https://www.npmjs.com/package/neo4j-driver). By using this package, our codebase can easily create prepared and raw queries to execute directly on Neo4j’s server. In addition, changes that potentially change the graph can be committed and rolled back as a user sees fit. This allows more flexibility when managing the graph database. The package, by default, uses non-blocking code such as promises, which allow developers to insert custom callbacks when a transaction is taking place, completed, or becomes erroneous. The package also allows developers to ‘pool’ database connections, which allows connection management for multiple server requests.

The code includes in-line and method comments to explain logic and syntax. An overview of all the functions created for the graph model is shown in Figure 19.

```javascript
static fetchGraph(graphName, callback) {
  let session = db.session()
  let resultPromise = session.readTransaction(function (transaction) {
    let query = graphName ? `MATCH (n:(${graphName}+)) WHERE (n)-[PART_OF]->(x) RETURN x` : `MATCH (n) RETURN n`;
    return transaction.run(query)
  });

  resultPromise.then(function(result) {
    session.close()
    callback(null, result)
  }).catch(function(err) {
    session.close()
    callback(err, null)
  });
}
```

Figure 19: This function fetches all nodes given a graph. If no graph name is provided through the query, then the function fetches all nodes across all graphs. Note that the transaction is built and ran in the promise. Once the promise finishes the assigned task, the ‘.then’ part of the promise executes, which results in a callback to send back the results to the controller. This also takes care of any errors since they are caught before being sent back.
Figure 20: doesTypeObjectExist function

```javascript
static doesTypeObjectExist(objectName, callback) {
    let session = db.session();
    let resultPromise = session.readTransaction(function(transaction) {
        let result = transaction.run('MATCH (n) RETURN DISTINCT LABELS (n)'
        return result;
    });
    resultPromise.then(function(result) {
        session.close();
        let resultList = [];
        for (let i = 0; i < result.records.length; i++) {
            resultList.push(result.records[i]._fields[0][0])
        }
        if (resultList.indexOf(objectName) > -1) {
            callback(true, null);
        } else {
            callback(false, resultList);
        }
    }).catch(function(result) {
        session.close();
        console.log(result)
    });
}
```

Figure 20 shows the doesTypeObjectExist function which checks if the label provided by the user API is valid and exists in the graph. This function is dynamic because it checks against the graph database rather than a string. This function is only meant to be used within the model class, hence we do some data processing within the `.then` clause of the promise’s callback. If the label provided did not match a label in the database, we return an error and give back a list of all available labels in the database.

Figure 21 shows the getRegexForEngagementType function. This function grabs the regex string in the correct format for Cypher’s `=~` (SQL equivalent: LIKE) operator. Note that this method should only be used internally within the code.
Figure 21: `getRegexForEngagementType` function

```java
static getRegexForEngagementType(type) {
    if (type == 'share')
        return '.*\\\[share\sauthor=.*\]'
    else if (type == 'assessment')
        return 'I\sam\taking\\\sthe\\\s[bookmark=.*]'
    else if (type == 'stories')
        return 'Please\sread\sthe\sstory,*'
    else if (type == 'likes')
        return '.*\\\[url\\]\slikes\s\\\[url,.*\]'
    else if (type == 'post')
        return true
    else
        return null
}
```

Figure 22: `findAnyNode` function

This function, as its name suggests, finds any node in the graph based on a labelname and a string to match by. This allows a flexible search tool to pull data out of the graph quickly. The function also supports a Boolean NOT, which allows the user to specify if they want the node to NOT contain some information. This query searches all properties of a node and therefore might require a very specific query for certain information. The query then returns a list of neo4j node IDs.
Figure 23: findPropertyValue function

Figure 23 shows the findPropertyValue function. This function finds property values of only DIGITAL_OBJECTs based on an engagement type. This function is mainly used for developers to find DIGITAL_OBJECTs that have not yet been given a relationship to an engagement type.

```
static findPropertyValue(nodeID, propertyName, engagementType, propertyValue, callback) {
    let resultPromise = session.readTransaction((transaction) => {
        if (typeof propertyValue !== 'undefined') {
            transaction.set(nodeID, propertyName, propertyValue);
        }
        return resultPromise.
    });
    resultPromise.then((result) => {
        callback(null, result);
    }).catch((err) => {
        console.log(err);
        callback(err, null);
    });
}
```

Figure 24: createNewPropertyFunction

Figure 24 shows the createNewProperty function. This function creates a new property, given a node ID. This function can also replace a current value for a property. This function should only be used by developers since this function changes data in the graph database.

```
static createNewProperty(nodeID, propertyName, value, callback) {
    let session = db.session();
    let resultPromise = session.readTransaction((transaction) => {
        result = transaction.get(nodeID, propertyName);
        return resultPromise
    });
    resultPromise.then((result) => {
        transaction.set(nodeID, propertyName, value);
        callback(null, result);
    }).catch((err) => {
        console.log(err);
        callback(err, null);
    });
}
```
Figure 25: createNewLabel function

Figure 25 shows the createNewLabel function. This function creates a new label (subject) in the graph database. The label name must be unique and therefore is checked to see if it already exists in the graph database. If the label name already exists, an error is returned. If it doesn’t, we continue the process and add the node to the graph database. Note that this only creates a blank label with no properties.

Figure 26: runRawQueryFunction

Figure 26 shows the runRawQueryFunction. This function runs a raw query from the user API. Note that this function should only be reserved for developers, as it can potentially change and/or delete parts of the graph. Please use this function with caution.
Figure 27: createNewRelationship function

Figure 27 shows the createNewRelationship function. This function creates a new relationship (predicate) in the graph. Given the two label names, a relationship name, and any additional query parameters to narrow down results, a developer can create relationships between nodes in a graph using this function. The function is responsible for checking if the labels exist in the graph and throwing an error if they do not. Note that the function will permanently change the graph by creating new relationships and therefore should only be used by a developer.

The graph model is complemented by the graph controller, which controls access to these functions from a request in the routes.js API. For a full overview of the code, including inline comments, please view the ‘graphModel.js’ file, located in the ./src/models/ folder.

A brief overview of the participant model is shown in Figure 28.
This function fetches a participant’s details given a specific graph name and participant ID. Note that the participant ID is from the SQL database and not from the graph database. Based on the entered parameters, a single record is returned from the graph database and back to the user API requesting the data.

```javascript
static fetchAllParticipants(graphName, callback) {
    let session = db.session();
    let resultPromise = session.readTransaction(function(transaction) {
        return transaction.run( MATCH (PROFILE)-[:PART_OF]->(network) where network.name = '${graphName}'
            return result
    }));
    resultPromise.then(function(result) {
        session.close();
        callback(null, result.records);
    }).catch(function(error) {
        session.close();
        console.log(error);
        callback(error, null)
    });
}
```

Figure 29: fetchAllParticipants function

Figure 29 shows the fetchAllParticipants function. The ‘fetch all participants’ function, as its name suggests, fetches all the participants in a certain graph. The function then returns a list of records for each participant returned.

The participant model is complemented by the participant controller, which controls access to these functions from a request in the routes.js API. For a full overview of the code, including inline comments, please view the ‘participantModel.js’ file, located in the ./src/models/ folder.

Figure 30 shows a brief overview of the engagement model class.

```javascript
var db = require('neo4j-driver').v1;
var db = db.addDriver('bolt://localhost', neo4j.auth.basic('neo4j', "password"));

export default class Engagement {
    constructor() {
    }

    static fetchEngagementDetails(graphName, engagementType, callback) {
        let session = db.session();
        let resultPromise = session.readTransaction(function(transaction) {
            return transaction.run(
                MATCH (givenProfile:PROFILE)-[:PART_OF]->(network) where network.name = '${graphName}'
                RETURN givenProfile as givenProfile
            );
        }));
        resultPromise.then(function(result) {
            session.close();
            callback(null, result.records);
        }).catch(function(error) {
            session.close();
            console.log(error);
            callback(error, null)
        });
    }
}
```

Figure 30: engagementModel class
From Figure 30, you can see that we require the neo4j-driver module. Using this we acquire a connection to the database called db. Our constructor for the engagement class is empty, but the class has one method called fetch engagement details. The method takes three parameters, which match up to the parameters in the routes.js. We create a session, draft up our query from the parameters, construct the string, and call transaction.run with the string. We check if the query worked and pass the result back. The engagement model is complemented by the engagement controller, which controls access to these functions. For a full overview of the code, including inline comments, please view the ‘engagementModel.js’ file, located in the ./src/models/ folder.

```javascript
use strict;

// This file handles Engagement 'Digital Object' data from the graph database
// To use the database, import the config file and then invoke the
// getDatabaseConnection function

var neo4j = require('neo4j-driver').v1;
var config = require('./config.js');
var db = neo4j.driver(config.url, neo4j.auth.basic(config.username, config.password));

export default class Comparative {
  static compareTwoGraphs(graphNAME1, graphNAME2, labelName, engagementTYPE, callback) {
    let session = db.session();
    let str1 = 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ', '') + ' return n.name as Network, givenProfile.id as Person, count(n) as engagements UNION ' + 'Match (n: [IS_A]->{eType: ' + engagementTYPE + '}) Match (n)-[:IS_A]->{eType: ' + engagementTYPE + '}'.replace(' ',='')

console.log(str1);
let resultPromise = session.readTransaction(function(transaction) {
  let result = transaction.run(str1);
  return result;
});
resultPromise.then(function(result) {
  session.close();
  console.log(result.records);
  callback(null, result.records);
}).catch(function(result) {
  session.close();
  console.log(result.error);
  callback(errorMessage.neo4jError + result.error, null)
});
}
}
```

Figure 31: Comparative Model class

Figure 31 shows the ComparativeModel class. The comparative model class is similar to the engagement model class with the difference between the two being the parameters that get
passed into the function, and the string containing the query. This particular query allows us to specify two graphs, an engagement Type used between participants, and a label (e.g., MALE). The query returns a table-view of all the engagements between the two persons. The query returns in a table the network name of the two people whose engagements you are seeing, Person 1, Person 2, and the number of engagements of that engagement type between those two people. For a full overview of the code, please view the ‘comparativeModel.js’ file, located in the ./src/models/ folder.

**Controllers**
The controller is meant to ‘control’ access to the model, and as such, does much of the intermediate data processing before sending it to the model. Since some of the functions were created using raw text from the API, without being prepared, it is crucial for the controller to sanitize any request from the API (note that some statements could not be prepared due to the limitations of this Node.js package). The controller is also responsible for parsing data coming directly from a database or Neo4j. The response for a request in Neo4j usually results in a long complicated JSON array containing many objects nested within, so it is important to parse this and return relevant information to the user.

The controllers are stored in the ‘src/controllers’ folder of the codebase. Each controller corresponds to a model, with the exception of the log controller class. Each controller exports methods instead of using a class design, which adheres to the stateless principles of the API. Any and all logic for the application takes place here.

Figure 32 shows a brief overview of the graph controller.
This controller function refers to the function in Figure 19, ‘fetchGraph’ in the graphModel class. The first part of this function logs the interaction in an object. This log is accessible in the log controller (discussed later). The next part fetches the graph from the model and uses a callback to parse the information sent from the model. If an error occurs, the logger writes an error while the graph sends back an ‘error’ response. If successful, the logger writes in the log object while the response sends back a JavaScript Object Notation (JSON) stringified version of the output from Neo4j. This response is sent directly to the client of the API that made the request in the first place, thus fulfilling the request.
Figure 33: findNode function

Figure 33’s function refers to Figure 22’s function ‘findAnyNode’ in the graphModel class. The first part logs the query in the log object. If no label name is provided in the query, the function returns an error immediately. If one is provided, the function passes the data onto the model’s findAnyNode function. Once we get back the data from the function, we parse through it and store it in an array called ‘nodes’. This array only has the IDs for all the nodes queried for from Neo4j’s database. Since this data can be thousands of lines long, we write the results into a CSV.
file and store it temporarily on the disk. Note that because of multiple requests of the same kind, it is possible to have a file with the same name. For that reason, a random alpha 10-character string filename is created to tackle that issue. Once the file is created, we send the file back to the client. Once the file is successfully sent, we go ahead and delete the file from the disk. If the CSV writer fails, we still send a list of IDs back to the client but only in text form. If the query fails with an error, for some reason, an error message will be sent instead of transmitting any data.

```javascript
exports.findPropertyValue = function(req, res) {
  let log = {
    queryType: req.query.type,
    developerID: req.query.developerID,
    didModifyGraph: req.query.didModifyGraph,
    request: req.query,
    timestamp: new Date().getTime()
  };

  Graph.findPropertyValue(req.query.label, req.query.propertyName, req.query.etype, req.query.propertyValue, function(err, result) {
    if (err) {
      logger.writeErrorLog(log, err)
      res.send(err)
    } else {
      let records = result.records;
      let fields = ['name']
      let nodes = [];
      for (let i = 0; i < records.length; i++) {
        let temp = {
          name: records[i].name, 
          nodes.push(temp)
        }
      }

      const json2csvParser = new json2csvParser(fields)
      const csv = json2csvParser.parse(nodes)
      let randomFilename = 
        fs.createReadStream(path.join(_dirname, `./random${Math.floor(Math.random() * 10)}_${Math.random()}.csv`) )

      let cypher = `match (n:User) 
                    return n`;
      if (err) {
        logger.writeErrorLog(log, err)
        res.send(csv)
      } else {
        let absPath = path.join(_dirname, `./random${Math.random()}.csv`) 
        fs.unlink(absPath, function(err) {
          if (err) {
            console.log(err)
            res.send(csv)
            logger.postMessage(log)
          }
        })
      }
    }
)}
```

Figure 34: `findPropertyValue` function

Figure 34’s function refers to Figure 23’s function ‘findPropertyValue’ in the graphModel class. As always, the function first logs the type of query in an object. Then the function passes all the information to the model’s ‘findPropertyValue’ function. Once the model completes the transaction, the function processes the output from Neo4j and extracts the node IDs and stores them in an array. The function then writes the output to a uniquely-named CSV file and sends it once the file is ready. Once sent, the temporary file is then removed from the filesystem.
Figure 35 shows the createNewLabel function. This function refers to Figure 25’s function ‘createNewLabel’ in the graphModel class. As always, the function first creates a log object for the type of transaction. Note that this transaction is marked ‘developerAPI: true’ and ‘didModifyGraph: true’. These fields can be queried by the log API in order to find all changes over time. The function then sends the information to the model’s function and waits for a response in the callback. Since the result is just marking a successful transaction, the function returns that to the client. If an error occurs, it is logged and an error is sent to the client instead.

Figure 36 shows the createNewProperty function. This function receives the request and response objects. It first creates a log object for the transaction. Then, it loops through the node IDs provided in the request and creates a new property for each node. If an error occurs, the error is logged and an error is sent to the client. If all processes are successful, the function returns to the client.

```javascript
exports.createNewLabel = function(req, res) {
    let log = {
        queryType: req.query.labelName,
        developerAPI: true,
        didModifyGraph: true,
        request: req.query,
        timestamp: new Date().getTime()
    }
    Graph.createNewLabel(req.query.labelName, function(err, result) {
        log.cypher = result
        if (err) {
            logger.writeErrorLog(log, err)
            res.send(err)
        } else {
            res.send(result)
            logger.writeLog(log)
        }
    })
}
```

```javascript
exports.createNewProperty = function(req, res) {
    let log = {
        queryType: req.query.labelName,
        developerAPI: true,
        didModifyGraph: true,
        request: req.body,
        timestamp: new Date().getTime()
    }
    let nodeIDs = req.body.nodeIDs
    let propertyName = req.body.propertyName
    let propertyValue = req.body.propertyValue
    for (let i = 0; i < nodeIDs.length; i++) {
        Graph.createNewProperty(nodeIDs[i], propertyName, propertyValue, function(err, result) {
            log.cypher = result
            if (err) {
                logger.writeErrorLog(log, err)
                res.send(err)
                return
            }
            if (i == (nodeIDs.length - 1)) {
                logger.writeLog(log)
                res.send(result)
            }
        })
    }
}
```
Figure 36 shows the createNewProperty Function. This function refers to the function ‘createNewProperty’ in the graphModel class. As always, the function first marks the type of query in a log object. Once the data is extracted from the query, it is sent to the createNewProperty function. Note that this function runs in a loop and will only send a response once the last node ID is modified. If an error occurs before that, a message is sent and the function returns in order to stop the processing of further requests, thus breaking the loop.

```javascript
exports.createNewRelationship = function(req, res) {
    let log = {
        logType: queryType,
        developerAPI: true,
        didModifyGraph: true,
        request: {body: req.body, files: req.files},
        timestamp: new Date().getTime()
    }
    if (req.files.csvImport) {
        let csvFilePath = req.files.csvImport.file
        let lineNum = 0
        let queries = []
        let label1 = ""
        let label2 = ""
        let relationshipName = ""
        csv().fromFile(csvFilePath).on('json', (obj) => {
            // IMPORT DATA FROM CSV
            // FIRST LINE HAVE THE HEADERS
            // SECOND LINE ONWARDS HAVE DATA
            // THIS INNER FUNCTION WILL BE CALLED FOR N LINES
            if (lineNum == 0) {
                //console.log(obj)
                label1 = obj.label1
                label2 = obj.label2
                relationshipName = obj.relationshipName
            }
            let query = "MATCH (n: " + label1 + ") , (m: " + label2 + ") WHERE ID(n) = " + obj.match1
            query += obj.match2 != "" ? " AND ID(m) = " + obj.match2 : "
            query += "RETURN [\" + relationshipName + \"] + r(n)
            queries.push(query)
            //DO NOT RUN ASYNC QUERY HERE.
            lineNum++
        }).on('done', (error) => {
            
        }
    
```
Figure 37 shows the createNewRelationship function. This function refers to the function createNewRelationship’ in the graphModel class. This function can take a CSV or just post parameters to run. To import a CSV, please make sure the corresponding key for the POST request is ‘csvImport’. The CSV file follows a peculiar format, as described in the user manual. It is imperative to follow the format or else an error will be thrown. The general format for the CSV headers are: ‘label1, label2, match1, match2, relationshipName’. The CSV must have headers, and the headers much be named as mentioned above. Failure to define the headers will result in an error. Depending on the size of the CSV file, this function may hang and not send a response back immediately. This does not mean the server is hanging, but instead that the background job has not returned a response. In that case, it would be best to check the logger to see if the job is complete in case the request timed out. If a CSV is not used, please make sure you have the following body keys when doing a 1:1 relationship match: ‘labelName1’, ‘labelName2’, ‘relationshipName’, ‘option: {‘match1’, ‘match2’}’. If no option is given, any node under labelName1 will be matched to labelName2 using the relationshipName as the relationship. In case the user wants to specify a node ID, simply use the JSON object ‘option’ with the corresponding keys: ‘match1’ and ‘match2’ to specify the single nodes to create a relationship from.
Note:
When creating the network nodes ‘e1r1’, ‘e1r2’, etc. it is very important to make sure the relationship name for all nodes connecting to the network nodes is ‘PART_OF’. Failure to do so will result in inconclusive results from the comparative API. Please follow the Cypher import documentation located at the bottom of the manual to correctly create and import Cypher nodes and their respective relationships.

```javascript
import Participant from "./models/participantModel"

exports.fetchAllParticipants = function(req, res) {
  Participant.fetchAllParticipants(req.params.graphNAME, function(err, participantArray) {
    res.send(JSON.stringify(participantArray));
  });
}
exports.getByIdParticipantId = function(req, res) {
  Participant.getByIdParticipantDetails(req.params.graphNAME, req.params.id, function(err, participant) {
    if (err) {
      res.send(err)
    } else {
      res.send(participant)
    }
  });
}
exports.create = function(req, res) {
  Participant.create(req.params.id, function(err, participant) {
    if (err) {
      res.send(err)
    } else {
      res.send(participant)
    }
  });
}
```

Figure 38: participantController class

Figure 38 shows the participantController class. The controller classes do post-processing of the data obtained from the model class. For the participant class, there are three functions related to the three different endpoints that we have. You can see that we don’t need to do post-processing of these queries, so we just pass the results right on through.
Figure 39: engagementController class

Figure 39 shows a snippet of the engagementController class. This class does post-processing for the data obtained from the engagementModel class. The JSON data returned from the engagementModel class is not fully complete. In that data, an engagement from person a to person b is not counted the same as an engagement from person b to person a. We want the engagement counts to be bidirectional whereas the table makes a distinction where the engagement came from and where it is going to. Therefore, we do a few simple loops to tell where this happens, and we add the two counts together to get an accurate number. We then send the newly updated array back to the client in the response.
Figure 40: comparativeController class

Figure 40 shows the comparative controller. This controller just checks for errors and passes on the result returned from the model. There is no data processing that occurs from this stage, and the results are directly passed back to the client.
Figure 41 shows the logController. The functions allow us to either get all the logs, get database, write to the log, write to the error log, find the developer logs, find participant logs, and find log logs by querying the Mongo database used to store the previously used API queries.
Running the Node Server

Currently this node server project is hosted here: https://github.com/adayal/GraphQueryResearch. Please access this link and download a copy of the project. The project should live on the same server that is running the Neo4j instance. To begin the installation process, open a bash prompt and type ‘npm install’. This will install all the libraries in a folder called ‘node_modules’ that have been specified in the package.json file. The project will contain a ‘config.js’ file in the root directory that will contain all the configuration for this project. Please change the username and password of Neo4j from the default. Please also change other configuration options as the developer sees fit. Once the node_modules are successfully installed, please run ‘npm start’ as a super user (‘sudo npm start’). The super user grants privileges to the Mongo instance to access files that might otherwise not be shared with the current user.

The developer may notice that there are many files that are converted to a new folder with a mirror copy of the source folder called ‘lib’. This process is done by the node module known as ‘babel’ (https://babeljs.io/docs/usage/cli/). Since the source code for this project is written in JavaScript ECMAScript 6 (ES6), in order to make it compatible for older npm systems, the code is transpiled to ECMAScript 5 (ES5). This folder is constantly updated every time ‘npm start’ is executed, and no manual changes should ever be done in the lib folder. All code changes must be done in the ‘src/’ folder in order to take effect.

The developer may also notice that an instance for the Mongo database is created. The Mongo database listens to port 27017 on the local machine. For this project’s logging tool, mongo must be running in the background. If the developer wishes to run Mongo on a different URL or port, please change the ‘mongourl’ key in the config.js file in the Node.js project.

Once the code is done transpiling, the default port the service uses is 3000. This can be changed in the config.js file. The port number was randomly chosen for this project and can be changed for any reason.

Once the message ‘API server started on: 3000’ shows up, the APIs are now ready to be used. Since the project depends on Neo4j for most of the API interactions, Neo4j must also be running. The project connects to Neo4j via the credential provided in the config.js file. If a change to a configuration change is necessary, the developer must restart the node service.
Accessing the Node Server

In order to access the node server, first open up a PuTTY connection that will forward a local port over localhost to the remote server’s open port, which in this case is 3000. Once this connection is open, please use an HTTP client (Postman, curl, etc.) to send an HTTP request. A list of APIs can be found in the user manual. For the purposes of this developer’s manual, the preferred tool is: Postman. If the server is running and the port forwarding is active, please send a request to the server via one of the provided APIs. Depending on the request type, there might be an immediate response or the response may take some time. Please wait for a few seconds before the response arrives back. The response will have the information requested in a JavaScript Object Notation (JSON) result. If an error occurs, it will be logged in Mongo and an error will be shown in the response. Please refer to the user manual for more specific instructions on how to retrieve error logs. Figure 42 gives an example of a successful request and a response from Neo4j.

Figure 42: HTTP request for participant

Figure 42 shows the HTTP request for getting a certain participant’s details. Note this participant has no identifiable information listed.

Figure 43 shows an example where an error is returned from a faulty request.
Figure 43: Sample Error Message

Figure 43 shows a sample error message. Requesting details about participant ‘abc’ returns an error: ‘Your request returned no results’. This error is produced in Node.js when Neo4j retrieves no results.

**Running the Neo4j Service**

Once the developer follows the installation instructions and configures the tool using the configuration file, the service is ready to run. The service is required to be running whenever interacting with the graph database to any degree. To start the program, simply type ‘neo4j start’ in your bash prompt. This command should be run as a root user, and should run in the background. Once the service starts and a message is printed, please open up a PuTTY telnet client and create a local port forward over localhost to the remote server where the Neo4j instance is installed. When the connection is active, use the provided localhost port to connect to Neo4j’s browser instance. Note that the settings must be similar or match the settings provided in the screenshots in the installation section.

The developer can then open up a browser and connect to the Neo4j service via a forwarded port over a telnet session. This will lead to a window that looks like Figure 44.
The developer will then be asked to enter in credentials for Bolt. In order to connect to the graph database, it imperative that the Bolt connection is working. The user can then issue commands over this interface and see a visualization of the data. The only concern here is that every user with access will have the ability to write and delete information from the database. Unfortunately our version of Neo4j does not have the option to set user privileges. The business version of Neo4j has the option to set user privileges but in the case for this proof-of-concept application, the team decided not to buy the business tier. For that reason, please restrict access to this interface as every command that is run from here will directly affect the data in the graph.

**Cypher Data Import and Relationship Creation**

The developer may notice in the Node.js project source folder there is a file labeled as ‘procedure.txt’ in the root directory. This file lists out the import statements required for importing all the data into Cypher. As mentioned earlier, Neo4j’s configuration file specifies the folder to read files from, and it is important that the files that are used for the import live here. Commands for the import must be run through the Neo4j web interface.

To prepare the data for import, please export the SQL data into comma separated values (CSV) files. For this project, we only imported the following tables: ‘user’, ‘profile’, ‘item’. The CSV files require headers in a certain format in order to be successfully imported.

For the ‘profile’ CSV file, please have the following CSV headers:
id, uid, profile_name, is_default, hide_friends, name, pdesc, dob, address, locality, region, postal_code, country, hometown, gender, martial, with, howlong, sexual, politic, religion, pub_keywords, prv_keywords, likes, dislikes, about, summary, music, book, tv, film, interest, romance, work, education, contact, homepage, photo, thumb, publish, net_publish

For the ‘selectedItems’ CSV file, please have the following CSV headers:

id, guid, uri, uid, type, wall, gravity, parent, created, edited, commented, received, changed, wall_owner, title, body, verb, file, deleted

For the ‘user’ CSV file, please have the following CSV headers:

uid, guid, username, password, nickname, email, openid, timezone, language, register_date, login_date, default_location, allow_location, theme, pubkey, prvkey, spubkey, sprvkey, verified, blocked, blockwall, hidewall, blocktags, unkmail, cntunkmail, notify_flags, page_flags, prvnets, pwdrest, maxreq, expire, account_removed, account_expired, account_expires_on, expire_notification, service_class, def_git, allow_cid, allow_gid, deny_cid, deny_git, openidserver

Note
These headers are required to be in this order since the query that exports data from SQL exports the following columns in this order.

The procedure.txt file specifies the filenames to import. Please change these filenames and their location to how the developer sees fit. Once the CSV files are ready, please copy and paste each of the procedures into the Neo4j command line web browser and run them. The CSV file also marks how to create relationships with other pieces of data; please run those procedures as well in the command line. Once these data have been imported and the relationships have been created, the developer then needs to manually create some nodes using the interface (or API) and link additional data for some of the queries to work (in particular, the comparative studies queries).

To do this, let us look at creating the following nodes in Neo4j: NETWORK, GENDER, MALE, FEMALE, OTHER, ENGAGEMENT, LIKES, SHARES, ASSESSMENTS, STORIES, POST, and their respective relationships. To understand how to create these nodes in Neo4j, we must first introduce some common Cypher keywords:

CREATE

The ‘CREATE’ keyword can be used to create either a node or relationship.
**MATCH**

The ‘MATCH’ keyword is used to find nodes in the graph that match a certain characteristic. In addition, this keyword is most commonly used with the ‘WHERE’ clause, which can be used to specify additional attributes to look for.

**DELETE**

The ‘DELETE’ keyword is used to delete nodes from the graph database. Note that nodes that are connected to the graph must first be ‘DETACH’ed and then ‘DELETE’ed.

**MERGE**

The ‘MERGE’ keyword couples the CREATE and MATCH keywords and creates relationships and nodes based on matching on pre-existing data. This manual does not use the MERGE keyword but it is important to understand in case a future developer finds use for it.

For more information on Neo4j keywords, please look at the following website: https://neo4j.com/docs/developer-manual/current/cypher/keyword-glossary/

To create the NETWORK node, please model your query with the following example query:

```cypher
CREATE (n:NETWORK{name: 'e1r1'})
```

This will create 1 node with the label name ‘NETWORK’ with the property name that will be set to ‘e1r1’. Please make sure your label is called ‘NETWORK’ (with all capital letters) so that it is compatible with the Node.js code.

To create the GENDER node, please use the following query:

```cypher
CREATE (n:GENDER)
```

This will create 1 node with the label name ‘GENDER’. Please make sure your label is called ‘GENDER’ (with all capital letters) so that it is compatible with the Node.js code.

To create the MALE node, please use the following query:

```cypher
CREATE (n:MALE)
```
This will create 1 node with the label name ‘MALE’. Please make sure your label is called ‘MALE’ (with all capital letters) so that it is compatible with the Node.js code.

To create the FEMALE node, please use the following query:

`CREATE (n:FEMALE)`

This will create 1 node with the label name ‘FEMALE’. Please make sure your label is called ‘FEMALE’ (with all capital letters) so that it is compatible with the Node.js code.

To create the OTHER node, please use the following query:

`CREATE (n:OTHER)`

This will create 1 node with the label name ‘OTHER’. Please make sure your label is called ‘OTHER’ (with all capital letters) so that it is compatible with the Node.js code.

To create the ENGAGEMENT node, please use the following query:

`CREATE (n:ENGAGEMENT)`

This will create 1 node with the label name ‘ENGAGEMENT’. Please make sure your label is called ‘ENGAGEMENT’ (with all capital letters) so that it is compatible with the Node.js code.

To create the LIKES node, please use the following query:

`CREATE (n:LIKES)`

This will create 1 node with the label name ‘LIKES’. Please make sure your label is called ‘LIKES’ (with all capital letters) so that it is compatible with the Node.js code.

To create the SHARES node, please use the following query:

`CREATE (n:SHARES)`

This will create 1 node with the label name ‘SHARES’. Please make sure your label is called ‘SHARES’ (with all capital letters) so that it is compatible with the Node.js code.

To create the STORIES node, please use the following query:
CREATE (n:STORIES)

This will create 1 node with the label name ‘STORIES’. Please make sure your label is called ‘STORIES’ (with all capital letters) so that it is compatible with the Node.js code.

To create the ASSESSMENTS node, please use the following query:

CREATE (n:ASSESSMENTS)

This will create 1 node with the label name ‘ASSESSMENTS’. Please make sure your label is called ‘ASSESSMENTS’ (with all capital letters) so that it is compatible with the Node.js code.

To create the POSTS node, please use the following query:

CREATE (n:POSTS)

This will create 1 node with the label name ‘POSTS’. Please make sure your label is called ‘POSTS’ (with all capital letters) so that it is compatible with the Node.js code.

All of these queries can also be run through the /addLabel endpoint from the API. Please refer to the user manual on how to run these queries through the API.

At this point, all your nodes should be created. Now we need to link the nodes. It is extremely important that the relationship names are exactly the same as described in this developer manual. Failure to do so will require the developer to delete and re-create the relationships until it matches the code.

To create the relationships between the gender-related nodes, please use the following queries:

MATCH (n:MALE), (m:GENDER) CREATE (n)-[:IS_A]->(m)
MATCH (n:FEMALE), (m:GENDER) CREATE (n)-[:IS_A]->(m)
MATCH (n:OTHER), (m:GENDER) CREATE (n)-[:IS_A]->(m)

Please note that all these relationships are denoted with the relationship name: ‘IS_A’.

To find all the PROFILE and DIGITAL_OBJECTs and create a relationship to a single NETWORK node, please use the following queries:

MATCH (n:DIGITAL_OBJECT), (m:NETWORK) CREATE (n)-[:PART_OF]->(m)
MATCH (n:PROFILE), (m:NETWORK) CREATE (n)-[:PART_OF]->(m)
To create the relationships between the engagement-related nodes, please use the following queries:

\[\text{MATCH (n:LIKES), (m:ENGAGEMENT) CREATE (n)-[:IS_A]->(m)}\]
\[\text{MATCH (n:SHARES), (m:ENGAGEMENT) CREATE (n)-[:IS_A]->(m)}\]
\[\text{MATCH (n:STORIES), (m:ENGAGEMENT) CREATE (n)-[:IS_A]->(m)}\]
\[\text{MATCH (n:POSTS), (m:ENGAGEMENT) CREATE (n)-[:IS_A]->(m)}\]
\[\text{MATCH (n:ASSESSMENTS), (m:ENGAGEMENT) CREATE (n)-[:IS_A]->(m)}\]

Please note that all these relationships are denoted with the relationship name: ‘IS_A’. To create the relationships between the DIGITAL_OBJECT types and their respective engagement nodes, please use the API endpoints ‘/findByPropertyValue’ and ‘/addRelationship’ as outlined in the user manual.

All of these queries can also be run through the /addRelationship endpoint from the API. Please refer to the user manual on how to run these queries through the API.

To match all the PROFILEs based on gender, please use the following queries:

\[\text{MATCH (n:PROFILE\{gender:"Female"\}), (m:FEMALE) CREATE (n)-[:IS_A]->(m)}\]
\[\text{MATCH (n:PROFILE\{gender:"Male"\}), (m:MALE) CREATE (n)-[:IS_A]->(m)}\]
\[\text{MATCH (n:PROFILE\{gender:"Transgender"\}), (m:OTHER) CREATE (n)-[:IS_A]->(m)}\]

Unlike the other queries, these queries must be run through the command line only. Searching for the nodes in the graph database use the /find endpoint in the API will result in a mismatch of data due to the current implementation limitations. For example, trying to match for males using the keyword ‘male’ will also tag females as males since ‘male’ is in the word ‘female’.

Once all these relationships are created, the data can be accessed via the APIs. If an error occurs with a Neo4j procedure, please review these installation steps as well as the Neo4j documentation.
User’s Manual

The following API endpoints can be used to create, read, update and delete data from Neo4j. The user should create an HTTP request from either their browser, cURL, HTTP client, or another tool. To run a query, the user must specify the hostname followed by the port and then the endpoint (e.g. http://localhost:5007/e1r1/participant). The user should also specify in their specific HTTP client whether or not the request is a GET or a POST request. If parameters are required, please specify them with the given key (as noted below).

`:graphName/participant`

- **HTTP Method:** GET
- **URL Parameters:**
  - ‘graphName’ → name of graph
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** Yes
- **Success Response:**
  - Array of Objects
    - nodeID
    - Label
    - Properties of Profile
- **Error Response:** Neo4j Error occurred: [custom error]
- **Description:** Get all the participants from a certain graph
- **API Access:** User
- **Example Request:**
  - localhost:5007/e1r1/participant

`:graphName/participant/:id`

- **HTTP Method:** GET
- **URL Parameters:**
  - ‘graphName’ → name of graph
  - ‘id’ → Database id (from friendica) for user account to look up
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** Yes
- **Success Response:**
  - nodeID
  - Label
● Properties of Profile

• **Error Response:** Neo4j Error occurred: [custom error]
• **Description:** Get data for a specific participant
• **API Access:** User
• **Example Request:**
  ○ localhost:5007/e1r1/participant/1060

/graph/view

• **HTTP Method:** GET
• **URL Parameters:** None
• **Body Parameters:**
  ○ graphName → name of graph
• **File Upload?** No
• **Parameters Required?** No
• **Success Response:**
• **Error Response:**
• **Description:** Return all nodes in neo4j. If graph name specified, return nodes only in specific graph
• **API Access:** User
• **Example Request:**
  ○ localhost:5007/graph/view
  ○ localhost:5007/graph/view?graphName=e1r1

/graph/findByPropertyValue

• **HTTP Method:** GET
• **URL Parameters:** None
• **Body Parameters:**
  ○ engagementType → Type of engagement to search for within a Digital Object
• **File Upload?** No
• **Parameters Required?** Yes
• **Success Response:**
• **Error Response:**
• **Description:**
• **API Access:** User
• **Example Request:**
  ○ localhost:5007/graph/findByPropertyValue?engagementType=likes

• **Notes**
○ Specify these in engagementType field in request

/graph/describe

- **HTTP Method:** Get
- **URL Parameters:** None
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** No
- **Success Response:**
- **Error Response:**
- **Description:** Describe the different nodes, properties and interconnectedness of the graph
- **API Access:** User
- **Example Request:**
  - localhost:5007/graph/describe

/graph/addProperty

- **HTTP Method:** POST
- **URL Parameters:** None
- **Body Parameters:**
  - nodeIDs: array of IDs for nodes ([id, id, id])
  - propertyName: property key
  - propertyValue: property to change or add
- **File Upload?** No
- **Parameters Required?**
- **Success Response:**
- **Error Response:**
- **Description:** Add new property or change existing property for a node given its id. This function can be called for multiple nodes using an array of node IDs.
- **API Access:** Developer
- **Example Request:**
  - localhost:5007/graph/addProperty
  - `{nodeID: [1, 2], propertyName: 'hello', propertyValue: 'world'}`

/graph/addLabel

- **HTTP Method:** GET
- **URL Parameters:** None
Body Parameters:
- **labelName** → name of new label

File Upload? No

Parameters Required? Yes

Success Response:

Error Response:

Description: Add a new label to the graph.

API Access: Developer

Example Request:
- localhost:5007/addLabel?labelName=helloWorld

/graph/addRelationship

HTTP Method: POST

URL Parameters: None

Body Parameters:
- **labelName1** → Name of label to connect from
- **labelName2** → Name of label to connect to
- **relationshipName** → Name of the relationship from labelName1 to labelName2
- **options** → Object {...}
  - **match1**: specific node id of a node with the label as labelName1
  - **match2**: specific node id of a node with the label as labelName2

File Upload? Yes
- **Type of File**: CSV
- **Body Parameter**: cvsImport
- **Headers Required?** Yes
- **Headers**: ‘label1’, ‘label2’, ‘match1’, ‘match2’, ‘relationshipName’
- **2nd Row**: labelName1 (required), labelName2 (required), Node ID (required), Node ID (not required), relationshipName (required)

Parameters Required? Yes

Success Response:

Error Response:

Description: Add a relationship between multiple nodes to a single node or label. This function accepts a CSV file upload for multiple node IDs. This function might timeout before a request is finished. Please check the logs to see if any errors occurred.

API Access: Developer

Example Request:
- localhost:5007/graph/addRelationship
- POST Params:
  - {labelName1=MALE&labelName2=GENDER&relationshipName=IS_A}
Notes
- When uploading a CSV, please use the following body parameter for the file: ‘csvImport’
- Please include the CSV headers in your file
- This cannot be rolled back. Every change with this API is permanent

/graph/find

- HTTP Method: POST
- URL Parameters: None
- Body Parameters:
  - `labelName` → name of label to look for
  - `propertyName` → specific property to look for
  - `contains` → string to look for
  - `not` → nullify the query (equivalent to a NOT in SQL operator)
- File Upload? No
- Parameters Required? Yes
- Success Response:
- Error Response:
- Description: Find a node that contains a string (substring) and return the ID of the node
- API Access: User
- Example Request:
  - Example task: Find all ‘female’ participants
  - *DISCLAIMER: YOU CANNOT DO THIS FOR FINDING ALL MALES. THIS IS A SUBSTRING MATCH FUNCTION. “Females” contains the substring “male”.

/graph/fetchNode

- HTTP Method: GET
- URL Parameters: None
- Body Parameters:
  - `nodeID` → id of node to fetch (in Neo4j)
- File Upload? No
- Parameters Required? Yes
- Success Response:
- Error Response:
• **Description:** Fetch details of a specific node based on the Node ID
• **API Access:** User
• **Example Request:**
  - localhost:5007/graph/fetchNode?nodeID=100

/::graphName/engagement/:engagementType

• **HTTP Method:** GET
• **URL Parameters:**
  - ‘graphName’ → Name of graph (e.g. ‘e1r1’)
  - engagementType → type of engagement
• **Body Parameters:** None
• **File Upload?** No
• **Parameters Required?** Yes
• **Success Response:**
• **ErrorResponse:**
• **Description:** Get all users whose engagement matches a specific engagement type
• **API Access:** User
• **Example Request:**
  - localhost:5007/e1r1/engagement/likes

/compare/::graphName1/:graphName2/:labelName/:engagementType

• **HTTP Method:** GET
• **URL Parameters:**
  - ‘graphName1’ → Name of graph (e.g. ‘e1r1’) to compare from
  - ‘graphName2’ → Name of graph (e.g. ‘e1r1’) to compare to
  - labelName → Categorical Variable (e.g. MALE, FEMALE, etc.)
  - engagementType → type of engagement
• **Body Parameters:** None
• **File Upload?** No
• **Parameters Required?** Yes
• **Success Response:**
• **ErrorResponse:**
• **Description:** Get all users whose engagement matches a specific engagement type
• **API Access:** User
• **Example Request:**
  - localhost:5007/compare/e1r1/e1r2/MALE/likes
/view/log/all

- **HTTP Method:** GET
- **URL Parameters:** None
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** No
- **Success Response:**
- **Error Response:**
- **Description:** Get all the logs
- **API Access:** User
- **Example Request:**
  - localhost:5007/view/log/all

/view/log/developer

- **HTTP Method:** GET
- **URL Parameters:** None
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** No
- **Success Response:**
- **Error Response:**
- **Description:** Get all the logs any transaction done by a developer
- **API Access:** User
- **Example Request:**
  - localhost:5007/view/log/developer

/view/log/graph

- **HTTP Method:** GET
- **URL Parameters:** None
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** No
- **Success Response:**
- **Error Response:**
- **Description:** Get all the logs that are marked for graph-related transactions
- **API Access:** User
- **Example Request:**
○ localhost:5007/view/log/graph

/view/log/participant

- **HTTP Method:** GET
- **URL Parameters:** None
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** No
- **Success Response:**
- **Error Response:**
- **Description:** Get all the logs that are marked for participant-related transactions
- **API Access:** User
- **Example Request:**
  ○ localhost:5007/view/log/participant

/view/log/log

- **HTTP Method:** GET
- **URL Parameters:** None
- **Body Parameters:** None
- **File Upload?** No
- **Parameters Required?** No
- **Success Response:**
- **Error Response:**
- **Description:** Get all the logs that are logged in the background (e.g. internal errors)
- **API Access:** User
- **Example Request:**
  ○ localhost:5007/view/log/log

**Commonly Encountered Errors**

The following error messages may occur when executing an API from above. If the error is an internal error, then please contact the developer. For other errors, the most common issue may be that a parameter is misnamed or missing entirely. Please make sure to follow the user manual when using an API endpoint. If more information is required, please use the log endpoints (as described in the user manual) to find the specific request and the error that occurred. Developers who encounter Neo4j internal errors can refer to the Neo4j log, which is stored in /var/log/neo4j.
Error Message: “Neo4j Error:...”
Error Meaning: Neo4j encountered an internal error. Please look at the logs on mongo to see more information.
Example:
  - Neo4j Error: Neo4jError: The client is unauthorized due to authentication failure.
    - Developer: Change your Neo4j password in the config.js file

Error Message: “Error: Missing parameter:”
Error Meaning: API endpoint found but is missing a parameter. Please check the corresponding API’s documentation to see which parameters are required. Note that parameter names are case sensitive.

Error Message: “Error: Please select a label from the following:”
Error Meaning: API endpoint found but cannot find requested label in the graph database.

Error Message: “Error: This label already exists”
Error Meaning: Occurs when adding a pre-existing label to the graph. Please make sure label names are unique.

Error Message: “Error: An internal error has occurred, please check the logs.”
Error Meaning: Node.js encountered an internal error. Please look at the logs on mongo to see more information.

Error Message:
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Error</title>
  </head>
  <body>
    <pre>Cannot GET /e1r1/notARealQuery</pre>
  </body>
</html>
Error Meaning: Could not find endpoint. Please check endpoint again. Verify node is running as well.

Testing
In order to test this project, the client followed the Developer’s manual from beginning to test out all the features. The installation process was straightforward according to the client. But the difficulties set in once the client started to import other graph data into Neo4j. Since our team only had access to one database, we were not sure how portable the code would be for other versions of the Friendica database. Inevitably, the client ran into issues, which the developers helped troubleshoot and fix. The client’s setup was initially faulty, which led to many of our APIs breaking. This prompted us to create graceful fail errors in order to avoid ugly errors from Neo4j and Node.js. Once the client had all the components of the SQL database successfully imported, we started to test each of the individual APIs.

Through the testing of the APIs, we found some dependencies of the graph that we had not mentioned before (e.g., relationships that must be named in all capital letters). This led us to change the user and developer manuals to correct those problems. In addition, the client also tested with faulty API queries to ensure that errors were handled gracefully.

To test the APIs the client first ran our API through an HTTP client and then compared the response from the server to the response of the SQL query against the corresponding MySQL database. Through this rigorous test, the client was able to test all functionality of this project’s API and verify that the results obtained from the API were accurate.

Furthermore, to test whether or not ‘write’ APIs worked, the client first ran the write API through the HTTP client and then checked if a change occurred through the Neo4j web command line and visualizer interface.

The testing process lasted approximately 3 weeks and covered all use-cases and listed scenarios in the requirements. Any issues were reported directly to the developers (us) and were fixed in a timely manner. This process required many 1:1 meetings with the client to ensure that results were being obtained correctly.

In addition to testing, we sat down with the client and explained each line of code in Node.js and each Cypher query so that the client could run more fine-grained tests against our API. This was helpful in our development since the client was able to run tests, fix the code when necessary, or alert us when help was needed.

Since this project is hosted on github, all changes can be viewed through the commit history, comments, and changes. For future testing, please create a pull-request so that all changes can be viewed before being pushed onto the master branch.

**Timeline**
<table>
<thead>
<tr>
<th>Date</th>
<th>Task Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2/2</td>
<td>Defining Scope for Amit’s and David’s tasks</td>
</tr>
<tr>
<td>2/6</td>
<td>Presentation for requirements and goals</td>
</tr>
<tr>
<td>2/15</td>
<td>Research Complete for RDF conversion and graphQL</td>
</tr>
<tr>
<td>2/28</td>
<td><strong>Milestone 1:</strong> Rough RDF scheme</td>
</tr>
<tr>
<td>3/2</td>
<td>Meeting with Prashant about goals and re-evaluation about current solution</td>
</tr>
<tr>
<td>3/13</td>
<td>Progress report on coding efforts</td>
</tr>
<tr>
<td>3/30</td>
<td><strong>Milestone 2:</strong> Successful execution for at least 2 scenarios</td>
</tr>
<tr>
<td>4/3</td>
<td>Hard deadline for code completion</td>
</tr>
<tr>
<td>4/30</td>
<td><strong>Milestone 3:</strong> Early draft for development manual complete</td>
</tr>
<tr>
<td>5/2</td>
<td>Completion of documentation</td>
</tr>
</tbody>
</table>

**Lessons Learned**

As the semester progressed, we learned that the goals we set initially were much harder than we anticipated, which led to an ambitious and often aggressive timeline. During our planning, we had forgotten to incorporate a week break for Spring break, which shifted all our goals back by one week. We had hardly any buffer time between different goals, which made it hard to recover when things went awry from previous goal timelines. It was ambitious to assume that a team of
two would be able to handle a database migration and full stack development over the course of the semester. This would not have been done without long nights from the team. Furthermore, many of the problems stemmed from IRB requirements that the data be handled a certain way. Figuring out firewalls on the host machine took two weeks. Those two weeks were not accounted for in the timeline and forced us to work harder to meet our goals. With the help of John Crawford, an IT specialist in the Social Interactome team, we were able to solve server connectivity issues.

Other issues arose when the data exported from one of the databases (e1r1) did not match subsequent versions of the same database (e1r2). This required specialized debugging to come up with a better export script when moving the data out of the SQL database. There was also a mismatch of data that we had thought was exported from the database but was in fact left out due to weird SQL constraints.

After several interactions with our client, we determined that we needed to change some of our project requirements to incorporate all the ideas our client had. This led to adding more APIs than what we had originally intended. We made the API self-sufficient enough for a developer to only use the API rather than the Neo4j command line to create, modify, read, and delete graph nodes.

Because of the complexity of the project and the difference in developer knowledge of the code, there was a bit of learning curve for the developers and the client. This learning curve increased the workload and required a more in-depth understanding of the different components of the code as well as the Neo4j server instance and how each interacted with another. Since the code was so complex, additional features were added to help aid in future debugging. This led to the unintended consequence of using a Mongo Database instance and logging all queries using Mongo. While this was technically not necessary for this project, it significantly helped subsequent developer efforts by providing logs of every database interaction. Implementing this feature took extra time that was not accounted for in the timeline.

Lastly, the client had asked for a front-end that consumed the APIs as an example of how a system would work when placed on top of the API. The developers created a front-end website that consumes the Node.js API. This led the developers to quickly spin up a website that consumed the API, and present it to the client. While the development of the website was unnecessary for the completion of the project, we thought that by including it in our final presentation, we could show the real-world example of how a developer or user might consume the API. For this matter, the development time was extended past the deadline and was incorporated as a special feature of this project.
Future Work

For future developers who wish to develop the APIs, it is recommended that the developer have prior experience in development of Mongo, Express, Angular (optional), Node (MEAN) stack. Developers who are interested in developing the website must have experience in Linux, Apache, MySQL, and the PHP (LAMP) stack. It is also recommended that the developer should follow an MVC pattern when developing the code in order to maintain highly coupled code. Since we did not have the knowledge of each respective stack or the MVC framework when we started, it became significantly harder and time consuming to make progress and thus added an extra burden during the development stage of the code.

Acknowledgements

The team would like to acknowledge John Crawford, an IT specialist and developer, who helped resolve many of our server issues during this project. John spent many hours with the team to help fix the several issues we faced when accessing data stored on the server.

The team would also like to acknowledge the professor, Dr. Edward A. Fox, whose insight and feedback on the project deliverables led to improvements in both the final product as well as this report.

Lastly, the team would also like to acknowledge our client, Prashant Chandrasekar, for helping with some of the development and system architecture. Prashant’s extensive experience in the Social Interactome project and experience working with researchers in the field helped set many of our goals and shaped our development to match the needs of the researchers who will eventually use this project’s deliverables. Any further information about this project can be obtained by contacting Prashant (peecee@vt.edu).

This research is part of the Social Interactome of Recovery: Social Media as Therapy Development project, which is a National Institute of Health (NIH) funded project. The grant identification number is: 1R01DA039456-01.
References


https://github.com/cayleygraph/cayley

2018, from https://neo4j.com/docs/

Sons, Ltd., 2012.