Project 2 Report

Introduction

For this project, it was required that a web-based graph visualisation platform be created. The project made use of the Enron email dataset, a relational database and a graph database to build the platform. Additionally, a frontend was developed for users to create accounts and interface with the platform. This report discusses the approach taken towards the project, as well as the methodologies used in implementing the requirements in the project specification.

Data Cleaning

Method

For the data cleaning we considered the Enron dataset contained in the .csv file provided to us in the project specification. The data was cleaned mainly by using the Python package "Pandas" and with the use of regular expressions. The method for cleaning the data is as follows:

- The .csv file containing all emails is read into memory with the use of Pandas.
- A new .csv file named “parsed.csv” is created with the relevant fields required for our databases. These fields are the “first_name”, “last_name”, “email_address”, “date”, “message_id”, “subject”, “folder”, “to”, “cc”, “bcc”, “body” and “reference_text.”
- For each row in the emails.csv file the relevant data gets extracted from the pure text, is inserted into the fields mentioned above and is written to “parsed.csv”. Some relevant cleaning is also done per row such as checking for invalid emails and/or undisclosed emails.
- After all emails in the emails.csv are broken up into the important fields and said important fields are placed into “parsed.csv”. the emails.csv is closed and “parsed.csv” is read into memory. This is done so that a second cleaning pass is done over all the remaining emails. In the second cleaning pass only empty emails and duplicate emails are removed. The remaining and therefore clean emails are written to “parsed_second_pass.csv.”

Junk emails

For the emails that we considered junk emails, a few findings in the paper provided to us in the project specification were used. The research paper found that the folders “discussion_threads,” “all_documents” and “sent_mail” only contained duplicate emails and were emails created by the computer. Therefore, we ignored all emails present in these folders.

We also removed emails whose bodies were duplicates of others and emails whose bodies were empty. All emails longer than 21844 characters were ignored. The reason for doing this is because many emails contained large webpages in html format which we considered to be junk. Emails of this length are also very unlikely to be typed out by a human and are therefore considered junk. Other steps to identify junk emails include removing all emails received from outside Enron. In other words, all emails not containing “@enron.com” were ignored. The reason for this decision is because we were only interested in emails concerning the employees of Enron. We also considered emails from outside Enron to be much more likely to contain spam or junk than emails that originate from within Enron.

Determining employees

To find employees of Enron we found that all employee emails were of the form <first_name>.<lastname>@enron.com. We therefore considered all unique emails that contained “@enron.com” to be employees and determined their first and last names by splitting their email addresses at the dot and throwing away the “@enron.com” string. We chose to go with this method of determining which emails are employee emails since it proved to be inaccurate to try and determine who were employees by looking at whether the email was sent from a certain folder. Not all employees had the same email folders. Thus, we considered all such email addresses to be employee email addresses, even if these employees were employed in less important jobs.

Databases

Relational Database

Database choice

The relational database chosen for use in the project was MySQL. In choosing a relational database amongst the allowed options, the following was considered:

- Open source over closed source, and with a large community (higher chance of finding solutions to problems that may be encountered).
- Operating system portability.
- Ease of setup and use.
- Documentation that is easy for a beginner given time constraints of project.

MariaDB fails the first requirement as the community is small relative to MySQL and PostgreSQL, and PostgreSQL fails the last requirement as its documentation is exhaustive and tough on the beginner. It is these shortcomings that resulted in choosing MySQL for use in the project.

Setup

The database and the database admin user must be created in the environment where the database will be kept. In the repository in the DB_scripts directory there is a README file that shows all the necessary steps to setup the database.

Populating the database

We wrote a Python 3 script that uses a MySQL Connection to create the necessary tables and then reads in the cleaned CSV file from the data cleaning scripts. It iterates through the messages and splits the information to populate the employee, message, recipient, and reference tables. After the script is finished the database is ready for use by the web application, where SQLAlchemy is used for object–relational mapping.

Import Script

The import script reads the cleaned CSV file (parsed_second_pass.csv) and extracts the first name, last name, and email address columns and then removes the duplicates. The dataset is then imported into the database's employee table. The script then iterates through the full dataset of messages and adds each message to the message table. For each message, the script analyses the recipients and adds each recipient to the recipient table. If the message was forwarded or a reply to another message the reference info is added to the reference table.

ORM vs Traditional Queries

The benefits of using an ORM over writing traditional SQL queries are listed below:

- Allows for code to be easily changed without affecting the rest of the application.
- Allows the programmer to work with their preferred language, which in our case is Python. As such, the programmer can leverage the programming language's functionality when working with the database.
- Takes care of optimising SQL code. Writing traditional queries may result in unnecessary code that may not be obvious to the beginner, and may also result in poor performance if incorrectly structured.
- Results in code that is reusable, less complex and independent of the type of database used (i.e., it allows for database abstraction).
- Adds a layer of protection from SQL injection, as the Object Relational Mapper composes the SQL query instead of having the programmer do so. Traditional queries are therefore more prone to this type of attack, especially if written by programmers without considerable experience.
- Does not require one to fully understand SQL and databases, so development time is lessened as the learning of the above mentioned by the programmer is reduced.

Graph Database: Neo4j

Why Neo4j

Our main reason to use Neo4j over TigerGraph was that Neo4j could be used on both Windows 10 and Linux directly without the use of Docker. The reason this mattered is because not all group members had computers running some form of Linux so we needed a graph database that could easily run on both operating systems.

Usage

For the graph database we used Neo4j. Specifically, we used the Neo4j community server version 4.2.5 as a standalone server. We made use of the import script linked in the project specification, with a few minor modifications, to import the data into the graph database. We create two other relationships other than that created by the import script. One of which is the “Mailed” relationship with a property label “num_emails” that denote the number of emails that an employee sent to another employee through any means (the email may have been sent directly, as a carbon copy or as a blind carbon copy). This directed edge is made so that it is easy to count the number of emails that one employee sent to another employee and to eliminate the need for us to look at the intermediate message nodes between two employees since we are not interested in the emails themselves but rather the number of emails and whether emails were sent or not. The other relationship being created is the “Contacted” relationship with the “exchange_emails” property label. The “Contacted” relationship is a bidirectional edge telling us if one employee sent messages to another employee and received messages back from that employee, thus denoting that they were in contact. The “exchange_emails” tell us how many emails both employees sent and received from each other. It is created by looking at the “Mailed” relationship and creating the “Contacted” relationship if two employees both emailed each other.

Graph Visualization

For the visualization of the graphs, we used Neo4j’s Neovis.js to draw the graphs. Because Neovis.js is written specifically for Neo4j it was very easy to draw the graphs without writing a lot of JavaScript. Despite this ease of use, due to the nature of the dataset we ended up with some graphs that took very long to calculate and load. The Label Propagation, Centrality and Social Network visualization displayed a lot of nodes and a lot more edges for each node and so is visually unpleasing to look at. For the Centrality visualization output we remedied the output to have the user limit the lower bound on the centrality so that only the nodes of specified centrality and higher were displayed. For the Label Propagation unfortunately, Neovis.js provided no functionality to group all nodes of the same labels together when drawn and we end up having to draw all nodes and edges. The Social Network as well, with the way we cleaned the data and considered the employees, displayed a lot of nodes and edges.The shortest path could unfortunately only display all the shortest paths in entire network and not between a single source and a single target.

Algorithm Queries

We made use of Neo4j’s Graph Data Science (GDS) library to compute the relevant data for our database. For the Label Propagation and Centrality, the result of the algorithm is written back to the graph database and the relevant Cypher queries and visual changes are used to call and correctly display the data.

An important thing to note is that we use the bidirectional “Contacted” relationship (employee A sent employee B an email and employee B responded to employee A in some way) in our graph algorithms. The reason for this is because we considered users that sent no emails but only received emails and users who only sent email but received none not to be participants of the global Enron community. Also due to the nature of our cleaned dataset some graphs did not draw in a reasonable amount of time when we considered every employee that simply only received or sent emails.

The algorithm used for label propagation is the following:

```
CALL gds.labelPropagation.write(
{
```

```
  nodeProjection: 'User',
  relationshipProjection: 'Contacted',
  writeProperty: 'label',
  relationshipProperties: 'exchange_emails',
  relationshipWeightProperty: 'exchange_emails'
}
);
```

As mentioned, we use the “Contacted” relationship between users (the employees) with the number of exchange emails as weight to run the Label Propagation algorithm. When the data is then called to be drawn in the browser the following query calls the relevant nodes and edges to be drawn for the label propagation algorithm:

```
MATCH p=(n:User)-[r:Contacted]-(m:User) RETURN p;
```

The algorithm used for centrality is the following Page Rank algorithm. As with the Label Propagation algorithm the “Contacted” relationship is used with “exchange_emails,” the number of emails exchanged, to be the weight of the relationships:

```
CALL gds.pageRank.write(
{
  nodeProjection: 'User',
  writeProperty: 'centrality',
  relationshipProjection: 'Contacted',
  relationshipProperties: 'exchange_emails',
  relationshipWeightProperty: 'exchange_emails'
}
);
```

The following query is used to call the data which must be drawn (in the browser via Java script the minimum centrality of the nodes can be set with a centrality variable):

```
MATCH p=(n:User)-[r:Contacted]-(m:User)
WHERE n.centrality > centrality AND m.centrality > centrality
RETURN p
```

For the Social Network we understood the threshold for a user to be considered part of the social network to be that the sum of the emails sent between two employees to be five or more and that both employees had to send at least one email to each other. The following query is used to request the social network:

```
MATCH p=(n:User)-[r:Contacted]-(m:User)
WHERE r.exchange_emails > 4
RETURN p
```

We understood the shortest path algorithm to be displayed based on the entire employee network structure. As mentioned, we used the “Contacted” relationship between users, with the number of emails exchanged representing the cost. The following query calls the nodes and edges to be drawn to represent the shortest path of the node in the network structure:

```
MATCH path=(n)-[r:Contacted]->(m)

RETURN

path AS shortestPath,reduce(exchange_emails=0,r in
relationships(path)|exchange_emails+r.exchange_emails) AS total
```

ORDER BY total ASC

Relational Database vs Graph Database

Below is a comparison of the relational database and graph database.

Relational Database	Graph Database
To establish relationships across tables a join needs to be used, which can become complex as the number of tables used grows.	Relationships are pre-established during creation.
Only context available is textual and numerical.	Adds more context to the data being presented through visual presentation.
Less resource intensive.	Very resource intensive during hosting and querying data.
Storage space used relatively similar in size to data imported.	Uses more storage space relative to data imported.
Data queried via ORM with connection to MySQL server.	Data queried via the bolt port by using JavaScript in Neovis.

Frontend Web Application

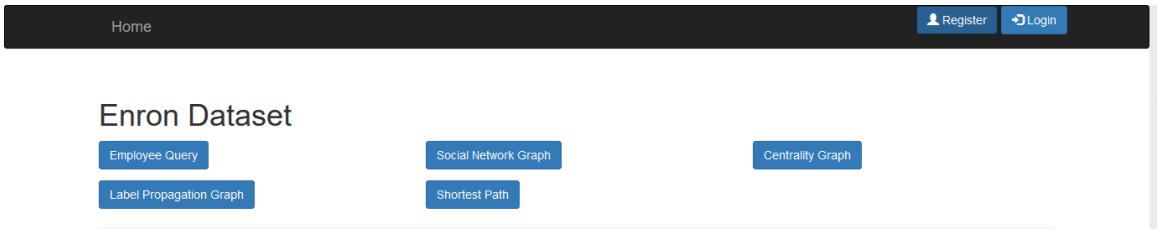
For the frontend framework we decided to continue to use Flask, as Flask was required to be used for the backend, as stipulated in the project specifications. Styling was also done using Bootstrap (Flask_bootstrap).

Implementation

We built an API that interacts with the Enron relational database and used the *requests* module to access form data sent from the website. The API includes functionality for creating users in the database and for querying employees. Html pages inherit their styling from base.html, and in this way the styling is consistent throughout the web application. Jinja was used within html documents.

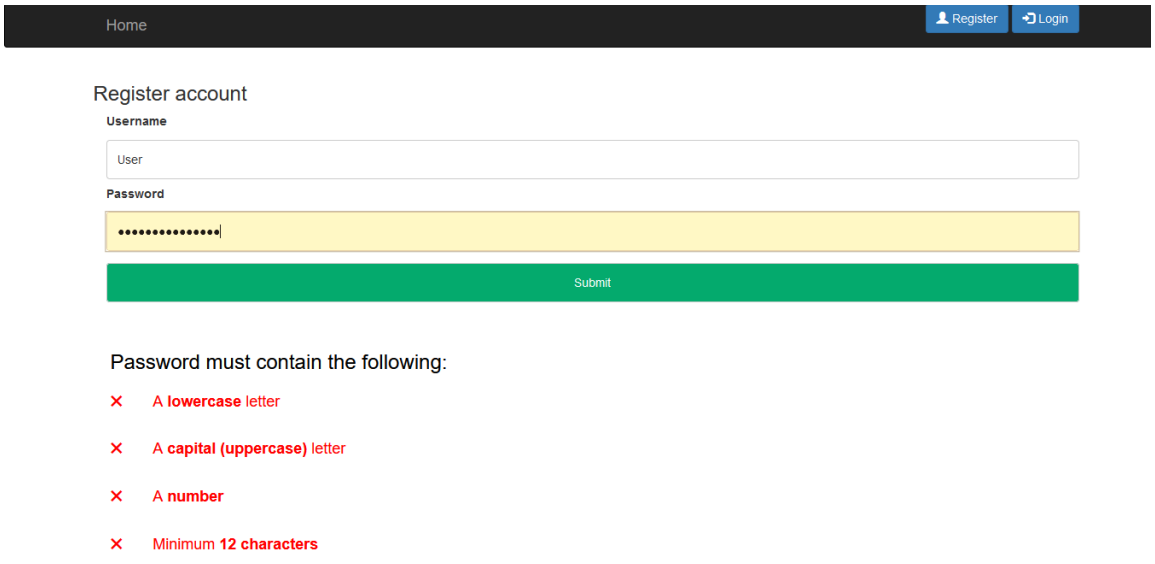
Home Page

Form data was sent back through the use of button actions, where users are redirected to the correct page. There is a Home button in the top left where users are able to return to the home page at any time, and users are navigated to custom error pages if a problem occurs.



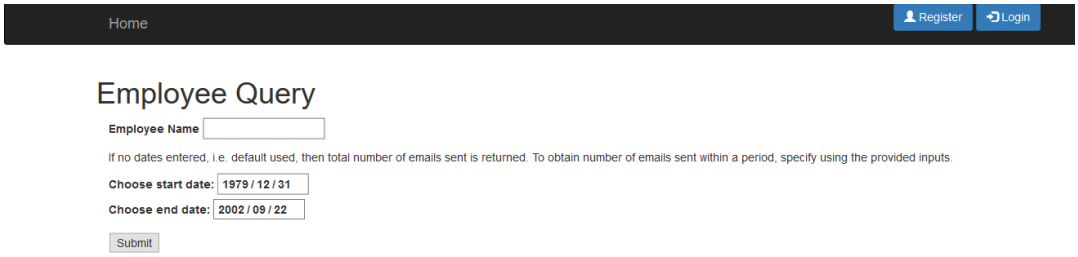
Register/Login Page

Users are able to register an account with the app, where their username and password are stored in the Enron database. Regular expressions were used to ensure passwords are unique, and usernames are checked from the database, ensuring that usernames are unique. When a user tries to enter an existing username, they will be redirected and informed that user is taken. Password conditions turn green as the user meets that condition.



Employee Query

Users are able to make employee queries by sending data back through the API as mentioned earlier, a search query is made and relevant information is sent back to the web app and displayed for the user.



Description of contribution made by group members

- Noah Foroma
  - Worked on developing SQL queries using ORM to interface with the database (for both Enron employees and app users).
  - Development on backend API to interface with frontend.
  - Worked on front-end pages for employee queries.
- Daniël Brand
  - Partial work on the cleaning of the Enron dataset.
  - Setup of the graph database.
  - Drawing of the Centrality, Label Propagation and Social Network graphs.
- Levi Dipnarain
  - Front-end web app for home page/login page/custom error pages
  - Logging/Registering interaction between Front-end and back-end
  - Styling
- George Soule
  - Front-end webpages
  - Styling
- Joubert Visagie
  - Partial work on the cleaning of the Enron dataset.
  - Setup and creation of the MySQL database
  - Created the API
  - Created the Flask app
  - Created the Home screen
  - Partial work on registering a user in the app.
  - Added graphs to display in the app.
- Xintomane Mahange
  - Shortest Path Algorithm

Each member contributed to the report where necessary (corresponding to the part of the project that they worked on). Additionally, wherever necessary, group members assisted each other.

YouTube Link

The demo video can be found here.