|  |  |  |
| --- | --- | --- |
|  | Project\_2: **Graph Algorithm  Visualization Report** |  |
|  | **PROJECT TEAM 21:**  RUTGER VAN HUYSSTEEN, 23052228  BERNARD OOSTHUIZEN, 21570124  KAYLAN NAIDU, 22778063  ANNA DAVIES, 21161097  HARRY ODENDAAL, 22794484  JACQUES LE ROUX, 23068574 |  |
|  | *Project report submitted in partial fulfilment of the requirements in the Faculty of Science (Natural Science) at Stellenbosch University.* |  |
|  |  |  |
|  | DATE OF SUBMISSION  17 / 05 / 2021  COMPUTER SCIENCE 334 |  |
|  |  |  |

CONTENTS

[CONTENTS 2](#_Toc72175821)

[ACRONYMS AND ABBREVIATIONS 3](#_Toc72175822)

[Defining Junk Data 4](#_Toc72175823)

[Determining Employees 4](#_Toc72175824)

[Methods used to determine the employees 4](#_Toc72175825)

[Reason for using these methods 4](#_Toc72175826)

[ORM vs Traditional queries 5](#_Toc72175827)

[Comparing Databases 5](#_Toc72175828)

[Graph Database 5](#_Toc72175829)

[Relational Database 6](#_Toc72175830)

[Graph Visualization Queries 7](#_Toc72175831)

[Label Propagation 7](#_Toc72175832)

[Centrality 7](#_Toc72175833)

[Shortest Path (Djikstra) 8](#_Toc72175834)

[Chosen Relational Database 8](#_Toc72175835)

[Chosen Graph Database 8](#_Toc72175836)

[Video submission 9](#_Toc72175837)

[Hosted website 9](#_Toc72175838)

[Readme 9](#_Toc72175839)

[REFERENCES 10](#_Toc72175840)

[APPENDICES 11](#_Toc72175841)

[Project Specifications 12](#_Toc72175842)

[Java implementation of Djikstra’s algorithm 14](#_Toc72175843)

# ACRONYMS AND ABBREVIATIONS

|  |  |
| --- | --- |
| ORM | Object Relational Mapping |
| DB | Database |
| SQL | Structured Query Language |
| ACID | Atomicity, Consistency, Isolation, Durability |
| VPS | Virtual Private Server |

# Defining Junk Data

Junk Data are e-mails in the email.csv file that contain left over of past attachments, such as .jpg, .doc and .xls files. Duplicate and blank messages are also considered as junk data.

All e-mails contained within the sent\_mail, all\_documents and discussion\_threads folders contained dupliacte e-mails and were removed. Duplicate e-mails also existed in some other folders, these were determined by checking if two e-mails have identical sender, recipients, subject and body. E-mails that had a blank body were also ommited from the cleaned data.

# Determining Employees

## Methods used to determine the employees

1. Determining whether a sender was an employee was done by looking at the X-From field in the e-mail. This field contained the name associated with the sender in one of two formats (Firstname Lastname; or Lastname, Firstname) that were either in quotation marks or not. After the full name of the sender was identified, it was then converted into the format lastname-initial, then compared to the folder to determine if they were part of the 150 employees in question. If they were not, their e-mail addresses would be converted to no.address@enron.com.
2. Dertermining whether a recipient was an employee was done in a similar manner, however, instead of checking the X-From field, the X-To, X-cc and X-bcc fields were used. These were also in the same format (Firstname Lastname; or Lastname, Firstname; that were either in quotation marks or not), however there could be multiple people in these fields, all each seperated by a comma. After the full name of the sender was identified, it was then converted into the format lastname-initial, then compared to the folder to determine if they were part of the 150 employees in question. If they were not, all their e-mail addresses would be converted into one e-mail address: undisclosed-recipients@enron.com.

## Reason for using these methods

1. This approach for determining whether a sender was an employee was used to compensate for employees that used various e-mail address, as they were all associated with the real name of the employee. It also excludes other enron employees that did not have their own folder. Some valid employees had folders that were named incorrectly, such as Sean Crandall, whose folder was named crandell-s as opposed to crandall-s. These special cases were also account for.
2. The last method used to determine a recipient was used if there was no valid recipient identifed within the fields. All e-mails in the emails.csv file were either sent to, or received by a valid employee, meaning a situation where the sender was no.address@enron.com and the recipient was undisclosed-recipients@enron.com should be impossible, however it did occur. To account for this situation, if there was no recipient specified within the e-mail, the folder name was used to determine the recipient.

# ORM vs Traditional queries

Object-relational mapping (ORM) is a programming technique that allowes for converting data between incompatible type systems, by using object-oriented programming languages. This means that any database queries can be written using the object oriented paradigm of the preferred language. ORM sets the mapping between sets of objects written in the preferred programming language which encapsulates the code needed in order to manipulate the data therefore, complicated SQL queries are not used anymore and objects can be interacted with directly.

The above statements encapsulate the first two benefits of using ORM, as it allows a programmer to use the preferred programming language as there is an ORM for most large languages (such as python and javascript) and allows the programmer to easily interact with the relational database without being proficient in SQL. The third benefit is that an ORM provides a level of abstraction, in turn allowing for a programmer not to be proficient with different databases(postgresql, mysql…) since the ORM will handle interacting with the database. This high level of abstraction can also be detrimental and may cause low-level troubleshooting to be more difficult, but this can easily be compensated for by switching over to SQL. When comparing performance of an ORM, it will be much more efficient that the SQL written by a programmer who is non proficient in SQL.

Traditional Queries in comparison to ORM, is firstly beneficial as it ensures the best possible performance if it is written correctly. Secondly, unlike ORM, when using SQL there is no chance of vendor lock-in restriciting the programmer with access to the database, as code created by another developer is not needed. As there is not a high level of abstraction present when using SQL, it allows for easier troubleshooting since a developer can see exactly what is going on which in turn makes low-level troubleshooting much easier.

# Comparing Databases

Relational databases are the favoured query tool in business, as the tabular structure makes it ideal for data that has uncomplicated relationships. It’s use is also highly recommended in accounting and other areas that work with transactional data. This is due to the simplicity of the data and the fact that it can easily fit in the format of the Relational database. Relational databases use tables with rows and columns and require complex connections between the tables. These relationships are connected across the tables and are set up using foreign keys.

Graph databases are ideal when axploring more complex datasets, as it has an added emphasis on relationships. This emphasis on relationships aids data experts in finding unforeseen links between data points, thus making graph databases the obvious choice when working with very relationship heavy studies. Graph databases are most commonly used in social media, fraud detection and also in recommendation engines. Graph databases use nodes and edges, which allows it to run very quickly as it does not require any conncetions. Relationships in a graph database are regarded as data and are depicted by the edges between the nodes.

## Graph Database

TigerGraph (released in 2017) is native parallel graph database which is purpose-built for loading massive amounts of data (terabytes) in hours and analyzing as many as 10 or more hops deep into relationships in real-time. TigerGraph is implemented in C++ and it’s queries are controlled through GSQL, a query language that is very similar to SQL.

Neo4j (released in 2007) is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend for applications. Neo4j is implemented in Java and it’s queries are controlled using the query language Cypher. Neo4j also has a very active and popular community and therefor is regarded as easy to learn. It also has the biggest support network and backing of all of the graph databases.

TigerGraph takes longer to load than Neo4j, but when it comes to pre-processing, it is much faster than Neo4j. This is because TigerGraph uses 19.3 times less storage than Neo4j. This makes TigerGraph 24.8 times faster when it uses the one-hop path query and when it uses the three-hops path query, it is 1808.43 times faster than Neo4j. It is also able to complete a six-hops path query, which is something that Neo4j can not do as it stops working after 2 hours, because it runs out of memory.

## Relational Database

A relational database is a type of database that stores and provides access to data points that are related to one another. Relational databases are based on the relational model ,using a structure which allowes us to identify and access data in relation to other pieces of data in the database. This allowes for a straightforward way of representing data in tables, where each row in the table is a record with a unique ID called ‘key’ and the columns of the table hold the attributes of the data, where each record has a value for each attribute. Therefor allowing to easily establish the relationship among data points.

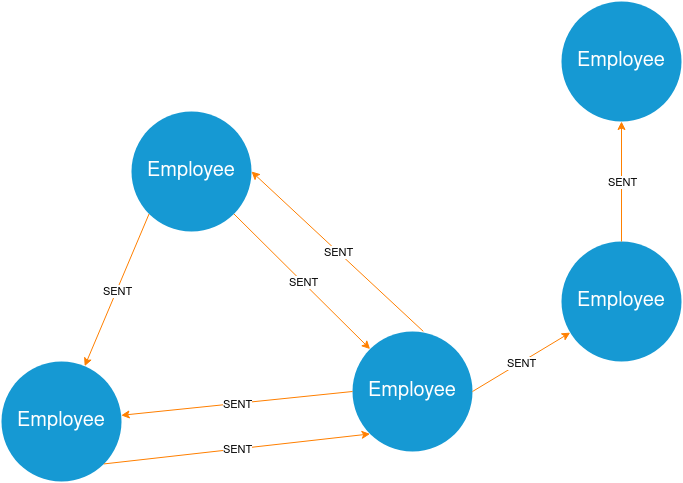
Two of the most widley used options for relational databases are MySQL and PostgreSQL, where PostgreSQL is an object-relational database while MySQL is a purely relational database.

The database speed differences between the latest versions of these databases is not large, but is however dependant on how the database is being used. PostgreSQL is know to be faster while handling massive data sets, complicated queries, and read-write operations. Where MySQL is know to be faster with read-only commands.

PostgreSQL is SQL compliant, whereas MySQL has diverted from the SQL standard. PostgreSQL is much more reliable as it is ACID compliant (Atomicity, Consistency, Isolation, and Durability) which is a transactional standards ensuring data integrity and keeping users from seeing wrong or stale data. In comparison to MySQL’s standard table handler which is not ACID compliant as it does not support consistency, isolation, or durability, however it does support atomicity. Nevertheless using a different database engine, MySQL could be made ACID compliant.

# Graph Visualization Queries

The following diagram shows an example of the Employee nodes and the relationships used in the algorithms.



## Label Propagation

This algorithm runs on all the Employee nodes that have sent at least one email to another employee. It forms communities of Employee nodes based on the amount of emails sent to each other. The nodes are displayed on the front-end by giving each community of Employee nodes its own colour.

|  |
| --- |
| **CALL** gds.labelPropagation.stream({                         nodeProjection: 'Employee',                         relationshipProjection: 'SENT',                         relationshipProperties: 'amount'                        }) |

## Centrality

The algorithm runs on Employee nodes that have sent emails to other employees. It calculates the importance or influence of each node in the graph. The amount of emails sent is used as the relationship weight property.

|  |
| --- |
| **CALL** gds.pageRank.stream({                        nodeProjection: 'Employee',                         relationshipProjection: 'SENT',                         relationshipProperties: 'amount',                         relationshipWeightProperty: 'amount'})                         YIELD nodeId, score                         **RETURN** nodeId **AS** **id**, score |

## Shortest Path (Djikstra)

A user defined procedure was implemented using Java. The integration tests worked on a small part of the database, but it gave a Null Pointer Exception when the procedure was used from the .jar file. The Java implementation can be seen in the appendix. We therefore used the following Cypher query to implement the Djikstra algorithm. It finds the shortest path between two Employee nodes. It weighs each link by the amount of emails sent. This means that it will find the path with the least emails sent.

|  |
| --- |
| MATCH (e1:Employee {emailaddress: ''}), (e2:Employee {emailaddress: ''})  **CALL** gds.beta.shortestPath.dijkstra.stream({                      nodeQuery: 'MATCH (e:Employee) RETURN id(e) as id',                      relationshipQuery: 'MATCH (e3:Employee)-[r:BI]-(e4:Employee) WHERE r.amount > 4 RETURN id(e3) as source, id(e4) as target, r.amount as weight',                      sourceNode: **id**(e1),                      targetNode: **id**(e2)})  YIELD **index**, sourceNode, targetNode, totalCost, nodeIds, costs  **RETURN** gds.util.asNode(sourceNode).emailaddress **AS** sourceNodeName, gds.util.asNode(targetNode).emailaddress **AS** targetNodeName, [nodeId **IN** nodeIds | nodeId] **AS** node\_Ids, costs |

# Chosen Relational Database

The team chose to use PostgreSQL. The reasoning behind this decision was purely based on the team’s general familiarity with it over MySQL, which meant that there would be less time spent on learning a new syntax, implementation and operations.

# Chosen Graph Database

The team chose to use Neo4j instead of TigerGraph. The reasoning behind this was because it had better documentation than that of TigerGraph and therefore seemed easier to implement and learn, as no one in the team has had any experience with any of these graph databases before.

# video submission

Follow the link below:

<https://youtu.be/qpKdCLB3O-c>

# Hosted website

The website is hosted at <http://cs334team21.xyz/>. The React App, Flask API and PostgreSQL is hosted together on a VPS of DigitalOcean. The Neo4j graph database is hosted on a separate VPS of DigitalOcean.

# Readme

Place the emails.csv file inside the datacleaning folder.

Run datacleaning.py , this will create all the other necessary .csv files.

Run docker in backend by using docker-compose up, so the PostgreSQL is up.

Run loadtodb.py

Install the requirements: pip install requirements

Run app.py

Run npm start in the frontend to run the web app locally

# REFERENCES

<https://www.talentopia.com/back-end/orm-vs-plain-sql-which-should-you-choose-and-when>

<https://developer.okta.com/blog/2019/07/19/mysql-vs-postgres#:~:text=Postgres%20Advantages%20over%20MySQL,more%20closely%20to%20SQL%20standards>.

<https://www.izenda.com/mysql-vs-postgresql/#:~:text=ACID%20stands%20for%20atomicity%2C%20consistency,in%20case%20of%20a%20failure.&text=However%2C%20when%20using%20InnoDB%20and,%2C%20MySQL%20is%20ACID%2Dcompliant>

[Implementing Dijkstra's Algorithm in Python](https://www.pythonpool.com/dijkstras-algorithm-python/)

<http://cs334team21.xyz/>

<https://youtu.be/qpKdCLB3O-c>

# APPENDICES

|  |  |  |
| --- | --- | --- |
| Appendix A | Project Specifications | 10 |
| Appendix B | Java implementation of Djikstra’s algorithm | 12 |

**APPENDIX A**

## Project Specifications

**PROJECT DESCRIPTION**

The project entailed, that a web-based visualization platform be created and that the Enron Email Dataset be used. The Enron Email dataset needed to be “cleaned up” following specific specifications, followed by creating a Relational database using the Enron Database Schema provided (page 2, [link](https://foreverdata.org/1009HOLD/Enron_Dataset_Report.pdf)).

The Web Application is required to enable users to register an account and be able to consequently login with the account after registration, to use the Web-App. All routes, except the login and registration route, must be protected, thus only granting access only if a user has a valid authentication token. If no authentication token is present, then the user must be redirected to the login page. The user must also be able to logout if they choose so.

Users must be able to select different graph algorithms to visualize the data. Users must be able to search for a specific Employee in the Enron Dataset and the information returned must follow adhere to the specifications under Employee Query.

A step by step video tutorial must be made and uploaded to youtube, that showcases every section of the marking rubric.

**IMPLEMENTATION SPECIFICATIONS**

This section describes all the technical aspects of what needs to be implemented and technologies that need to be used as solutions.

The following specifications are implemented for the Web Stack:

Any frontend framework is allowed,

Flask must be used for backend,

ORM must be used for the Relational Datebase operations,

May use either PostgreSQL, MySQL or MariaDB for the Relational Database,

May use either Neo4j or TigerGraph for the Graph Database.

**USERS**

* User information must be stored in a relational database
* Successful login must return an Authentication token (e.g JWT) - The generated

authentication token should be stored in a database so that it can be checked when the user browses the web page to ensure that it is a valid authentication token. The authentication token should be deleted when the user logs out.

* New users MUST be able to register to use the application.
  + Usernames must be unique
  + Passwords must be at least 12 characters long, including an upper-case
  + letter, lower-case letter, number and special character
  + A password confirmation should be included in the register page.

**DATA CLEANING**

* Remove duplicate emails
* Remove emails containing junk data - defining what these emails are is up to the

student/team, but the paper linked in the last bullet point can be used as a

guideline.

* Remove all entries with blank message section
* Convert invalid email addresses to “no.address@enron.com”
* Convert undisclosed email address to “undisclosed-recepients@enron.com”
* Remove junk messages such as messages returned by email system or email

transaction failures.

* If the message was sent to a group, retain the group name. The recipient

information can be determined from the message of each user.

* The data cleaning steps are taken from ([link](https://foreverdata.org/1009HOLD/Enron_Dataset_Report.pdf))(2021).

**EMPLOYEE QUERY**

* User MUST be able to search for an Employee based on first name and/or surname
* Information returned should include the following:
  + Number of emails the user has sent in total or in a time period
  + Top 5 most contacted employees (based on sent emails)

**GRAPH VISUALIZATION**

* The following three graph algorithms must be implemented:
  + Shortest Path between two people (Dijkstra, A\* etc)
  + Label Propagation (Clustering)
  + Centrality Algorithm
* 1 of the 3 algorithms must be done in native Cypher/GSQL - This algorithm must be built from scratch, and all the processing and maths etc must happen in the native querying language. The data cannot just be query then processed in python. The whole algorithm must be executed in the native querying language. - Due to the limitations of Cypher, user-defined functions will be allowed.
* 2 of the 3 algorithms can be done using Networkx or using a pre-existing library in Cypher/GSQL
* A derived social network must be built (page 7 in [link](https://foreverdata.org/1009HOLD/Enron_Dataset_Report.pdf))
* The visualization can be done using a library like d3.js
* The output of all 3 algorithms must be a visual graph displayed for the user.

**APPENDIX B**

## Java implementation of Djikstra’s algorithm

|  |
| --- |
| **@Procedure**(value = "example.Djikstra") **@Description**("Get djikstra.")  **public** Stream<ShortestPathNodes> **Djikstra**(@Name("sourceNode") Node sourceNode, @**Name**("targetNode") Node targetNode) {     *// Set distance to each node to infinity, source node is set to 0*     *// Evaluate each neighbouring node and check if current value is higher than value of current node + value of edge connecting neighbouring node. If it is then change to the new value.*     *// Delete current node from unvisited nodes*     *// Pick nearest neighbour and set it as current node*      Map<Node, Long> unvisited = **new** HashMap<Node, Long>();     Iterator<Node> employeeNodes = db.beginTx().findNodes(EMPLOYEE);     Map<Node, Long> visited = **new** HashMap<Node, Long>();     Dictionary<Node, Node> parents = **new** Hashtable<Node, Node>();      employeeNodes.forEachRemaining(n -> {         unvisited.put(n, **new** Long(10000000));         parents.put(n, targetNode);     });      Node currentNode = sourceNode;     Long oldDistance = **new** Long(0);     Long newDistance = **new** Long(1000000);      Long currentNodeDistance = **new** Long(0);     unvisited.put(currentNode, currentNodeDistance);      Node neighbourNode;     Node nearestNeighbourNode = currentNode;     Long nearestNeighbourDistance = **new** Long(10000000);     Long amount = **new** Long(0);     Node parentNode = targetNode;     Long parentNodeDistance = **new** Long(1000);     Relationship rel;     Iterable<Relationship> relationships;     Iterator<Relationship> relIterator;      **while** (unvisited.isEmpty() == **false**) {         relationships = currentNode.getRelationships(MyRelationshipTypes.EMAILS);         **if** (relationships != **null**) {        relIterator = relationships.iterator();                    **if** (relIterator != **null**) {    **while**(relIterator.hasNext()) {         rel = relIterator.next();        neighbourNode = rel.getOtherNode(currentNode);               **if** (neighbourNode != **null**) {                 **if** (unvisited.containsKey(neighbourNode)) {                     oldDistance = unvisited.get(neighbourNode);                     **if**(rel.getProperty("amount", **null**) != **null**){                         amount = (Long)rel.getProperty("amount", **null**);                     }                     **if** (amount != **null**) {                         newDistance = currentNodeDistance + amount;                         **if** (oldDistance > newDistance) {                             *// Change current node distance to shorter distance*                             unvisited.put(neighbourNode, newDistance);                         }                         *// Update parent node if current node distance < current parent distance*                         parentNode = parents.get(neighbourNode);                         **if** (visited.containsKey(parentNode)) {                             parentNodeDistance = visited.get(neighbourNode);                             **if** (currentNodeDistance < parentNodeDistance) {                                 parents.put(neighbourNode, currentNode);                             }                              } **else** {                             parents.put(neighbourNode, currentNode);                         }                     }                                     }             }            }    }         }          **if** (currentNode != **null** && currentNodeDistance != **null**) {             visited.put(currentNode, currentNodeDistance);             unvisited.remove(currentNode);         }*.*              *// Set current node/distance to neighbour with shortest distance (look for shortest distance in unvisited nodes dictionary)*         *// First look if any neighbours are in unvisited and find shortest distance one, else take shortest distance one in whole unvisited list*         Long nextDistance;         **int** i = 0;          **for** (Map.Entry<Node, Long> entry : unvisited.entrySet()) {             nextDistance = entry.getValue();             **if** ( nextDistance != **null**) {                 **if** (i == 0) {                     nearestNeighbourNode = entry.getKey();                     nearestNeighbourDistance = nextDistance;                 } **else** **if** (nextDistance < nearestNeighbourDistance) {                     nearestNeighbourNode = entry.getKey();                     nearestNeighbourDistance = nextDistance;                 }             }                                 i = i + 1;         }         currentNode = nearestNeighbourNode;         currentNodeDistance = nearestNeighbourDistance;     }      *// Get shortest path from source node to target node through parents*     currentNode = targetNode;     List<Node> shortestPath = **new** ArrayList<>();      **while**(currentNode != sourceNode) {         shortestPath.add(currentNode);         currentNode = parents.get(currentNode);     }     shortestPath.add(sourceNode);     Collections.reverse(shortestPath);          **return** Stream.of(**new** ShortestPathNodes(visited, shortestPath)); }  **public** **static** **class** **ShortestPathNodes** {     **public** List<Node> nodes = **new** ArrayList<>();     **public** List<Long> distances = **new** ArrayList<>();      **public** **ShortestPathNodes**(Map<Node, Long> visitedNodes, List<Node> shortestPath) {          **for**(Node n: shortestPath) {             **this**.nodes.add(n);             **this**.distances.add(visitedNodes.get(n));         }     }  } } |

The user defined procedure can be called from within a Cypher query with the following statement.

|  |
| --- |
| **CALL** example.Djikstra(sourceNode, targetNode) |