Graph Data Model: Storing temporal financial transaction data in a graph database

Author

Cristian Amaru Sinchico Arias

Utrecht University

Princetonplein 5, 3584CC Utrecht

Departement Informatica UU

Project Supervisors

Dr. I.R. Karnstedt-Hulpus

V. Shahrivari Joghan

Second Examiner

Dr. M.W. Chekol

Applied Data Science

Utrecht University

Amsterdam, June 30, 2022

Abstract

Introduction
This research aims to develop and propose a model to store temporal financial transaction data and create a network graph can be created. As such, the data will be stored in a graph database. To develop a model that handles large amounts of data and different structured data, research is done to examine different techniques and methods for storing such a graph data model. This model is then validated on a synthetically generated financial transaction dataset *PaySim*. This research is then divided into two parts: data representation and data storage.

The data representation part of the research focuses on the representation of the financial transaction data and specifically focuses on:

- Giving insights into the relationships of the financial transaction data by showing the transactions as a network.
- Showing how the network looks at a given point of time since financial transaction data is time-dependent.

The data storage part of the research focuses on the way financial transaction data can be stored and mainly focusses on:

- Creating a data model to store the temporal financial transaction data.
- Storing a synthetic financial transaction dataset, *PaySim*, in a graph database using Neo4J.

The main research question is as follows: *"To what extent can a data model be developed to store temporal financial transaction data in a graph database so that the relationships in the data are not lost, and how can this model be optimized so that it performs well."*

Background
Graphs, where relationships (edges) between nodes are directed, are called directed graphs and are of particular interest for financial transaction data. These graphs can be created from graph databases and these databases can be divided into two categories: Resource Description Dataframe (RDF) and property graphs. In general, property graph databases are known for their ease of use and performance. As properties of the data can be stored, the problem of having to generate big queries to capture relationships is tackled.

Methods
First, in order to answer the research question, research is done on the steps data modeling consists of:

1. Identifying entities;
2. Identifying attributes of these entities;
3. Identifying relationships amongst these entities;
4. Mapping the attributes to the entities;
5. Find a balance between normalization and performance; this means finding a balance between reducing data storage and query performance;
6. Finalize and validate the data model.

As this research focuses on storing the relationships between the data generating a network graph, the most convenient way is to store the data in a property graph database to create a labeled property graph. The resulting financial transaction network is a directed network and can be of different structures. One where transactions are generated as nodes, and one where transactions are generated as edges. The first

network will generate two types of nodes: Account nodes and Transaction nodes. These nodes contain node properties and are connected by two different edges: *Transaction from* and *Transaction to*. Properties of the Account nodes consist of *Account number, Balance before transaction,* and *Balance after transaction*. Properties of the Transaction nodes consist of: *Timestep, Type of transaction, Amount, Fraud, and Flagged as fraud*. In this network, the edges will contain no properties. The second network will generate one type of node: Account nodes. Here, the transactions are generated as edges connecting the Account nodes in the network. Properties of the Account nodes consist of *Account number, Balance before transaction,* and *Balance after transaction*. In this network, the edge *Transaction,* contains properties that consist of: *Timestep, Type of transaction, Amount, Fraud, and Flagged as fraud*.

The composition of the model to store and display the financial transaction data is done in Python and Neo4J. The modeling is done in Python and through the package Py2Neo, the data can be sent from Python to Neo4J to store the data into a graph database and display the network as a graph.

Data
Due to the many regulations concerning data privacy, acquiring financial transaction data is not easy. For this reason, the synthetically generated dataset, *PaySim*, is used in this research. *PaySim* is simulated data based on a sample of real transactions which are extracted from logs of a mobile service and consists of the following 11 properties: *Timestep, type of transaction, amount of transaction, source Account number, Balance before transaction of source account, Balance after transaction of source account, destination Account number, Balance before transaction of destination account, Balance after transaction of destination account, Fraud,* and *Flagged as fraud.*

Implementations and conclusions
The research has shown that, in the scope of this research and with the *PaySim* dataset, the model performs well on both network structures. The network where transactions are edges is generated faster, but this is expected as fewer nodes and edges have to be generated. Due to the scope of this research, the model is still in an early stage and can only handle data that is presented in a single data file. The model is validated on the first timestep and tweaked accordingly to increase performance. The data is stored completely and a complete network is generated. The following points are of importance to improving the model.

First, the model should be able to handle a variable set of properties as this model can only handle a predefined set of properties. If data contains more properties than the predefined set, this data gets lost in the process of this model. Additionally, more time could be spent coding more efficiently, as this influences the performance of the model as well. Next, the model can be improved by implementing a way to extract data from multiple data files and store it in the graph database. Finally, the model was tweaked and performance was measured based on the first timestep of the *PaySim* dataset. This should be done on more data but was not possible due to the scope of this research.

Preface

Before you lies my graduation report on the completion of the Applied Data Science Master's program at the University Utrecht. With this report, a long period as a student comes to an end. With pride and satisfaction, I can look back on this period.

I would like to take this opportunity to thank several people. First of all, I would like to thank my Project Supervisor, Ioana Karnstedt-Hulpus, for the guidance I received during this research. I would also like to thank Vahid Shahrivari Joghan for his help during this period and his overall guidance. Finally, I would like to thank my family and friends for standing by me all this time and supporting me through it all. Without them I would not have made it this far.

Cristian Amaru Sinchico Arias

Amsterdam, June 30, 2022

Table of Contents

1 Introduction

1.1 Context

Financial fraud has been a big problem for many organizations within multiple sectors (Raj, 2022). As such, billions of dollars are lost yearly due to financial fraud. For example, the Bank of America paid 16.65 Billion dollars as a result of a financial fraud case in 2014 (U.S. Department of Justice, 2014). However, fraud also occurs on a much smaller scale. Fraudsters and scammers are getting smarter and their methods to commit fraud are getting more refined (ING, n.d.). In this day and age many forms of fraud exist. According to ING, a Dutch internationally operating bank, many forms of financial fraud are occurring on a frequent base, such as phishing, bank-to-home scams, investment fraud, and many more.

Many organizations are trying to develop new methods to discover patterns of this fraud at an earlier stage. Consequently, this research has been set up and conducted. This research is part of a bigger research and focuses on the way financial transaction data is stored and how it can be presented. Hereafter, the data can be analyzed and algorithms can be built for fraud detection.

1.2 Aim of this research

The aim of this research is to develop and propose a model to store financial transaction data and make it presentable. To get a clear overview of the structure of the financial transaction data, a financial transaction network will be composed. This network will then be presented in a graph. To present the data in such a graph, the data has to be stored in a graph database (Robinson, Webber, & Eifrem, 2015). Chapter 3 elaborates on graph databases. This will result in developing a graph data model, which stores the data and can make a network graph. As Data Scientists, we are concerned with the processing and analysis of large amounts of data (van Beek, 2021). This means the model should handle large amounts of data and be generalized so that it can be used to store different structured financial transaction data. To develop a model that performs well on this account, research will be done to examine different techniques and methods for developing such a graph data model. This model will then be validated on a synthetically generated financial transaction dataset, called *PaySim*. More about *PaySim* can be found in chapter 4.

This research can be divided into two parts: data representation and data storage.

Data representation

The first part of this research focuses on the representation of the financial transaction data. This specifically focusses on:

- Giving insights into the relationships of the financial transaction data by showing the transactions as a network.
- Showing how the network looks like at a given point of time, since financial transaction data is time-dependent.

To show how the financial transaction network looks at a given point of time, a Python API will be proposed for extracting the network of transactions at a given point of time.

Data storage

The second part of this research focusses on the way financial transaction data is stored. In addition to making the graph data model, it is also important that the strengths and weaknesses of the model are described. This model will be used to store financial transaction data. This means the model has to be generalized for different structured financial transaction data to be loaded. Based on how well the model is generalized, advice can be given for possible follow-up research. Therefore, this part of the research mainly focuses on:

- Creating a data model to store the temporal financial transaction data
- Storing a synthetic financial transaction dataset, *PaySim*, in a graph database using Neo4j.

1.3 Research question

In order to achieve the objective of this research, several sub-questions alongside the main research question have been formulated. The main goal of this research is to propose a data model to store temporal financial transaction data. The main research question is then as follows:

"To what extent can a data model be developed to store temporal financial transaction data in a graph database so that the relationships in the data are not lost, and how can this model be optimized so that it performs well."

To answer this research question, the following sub-questions have been formulated that will be answered in this research:

- What are general properties that appear in financial transaction data?
- Which types of graph databases exist and how do they differ?
- How does a data model perform well?

1.4 Reading Guide

In this report, the problem analysis is concretely defined first. Subsequently, the method and methodology of the research are described and the data that is used to validate the model is defined. Hereafter, the results are presented. Finally, advice is given with regard to the results of the research, followed by advice for further research.

2 Background on storing graph structured data

Graphs have been part of Mathematics for many years, but this concept is rather new for data modelers (Frisendal, 2019). Graphs where relationships (edges) between nodes are directed are called directed graphs. These directed graphs are of particular interest for financial transaction data. These graphs can be created from graph databases and these databases can then be divided into two main categories. The first one is a Resource Description Dataframe (RDF-triple store), which is a graph data model that stores semantic facts (OnToText, 2019). The second category is a property graph, which is a graph model where the relationships carry properties and a name (Knight, 2021). In general property graph databases are known for their ease of use and performance, while RDF graph databases are known for interoperability and semantics (Anadiotis, Knowledge graph evolution: Platforms that speak your language, 2020). In the following subchapters, differences between RDF and property graphs will be explained.

2.1 RDF Databases

The first method is the RDF-triple store, which is short for Resource Description Dataframe. The RDF is a flexible model which describes semantics of information about resources on the web (Ma, Capretz, & Yan, 2016). It is a building block of the Semantic Web (Anadiotis, Graph databases and RDF: It's a family affair, 2017) and a general framework that represents interconnected data on the web and its statements are used for exchanging metadata. Through exchanging metadata, data can be exchanged based on relationships. The RDF makes use of three entities, hence the name triple store, and generates an expression in the form of subject (entity) - predicate (attribute) – object (value) about semantic data (Stefani & Hoxha, 2018). This structure of subject-predicate-object represents links in an RDF graph. Because of this structure, graphs can be created using RDF.

2.1.1 Pros and cons of RDF

A RDF is used to store data and create graphs. One benefit of this type of database is that sharing metadata is encouraged due to the consistent framework. Another benefit is that software that makes use of metadata can function better due to the uniform syntaxes of RDF to query and describe data. With the uniform syntaxes of RDF and the query capability, information can be exchanged more easily (Loshin, n.d.). Even though RDF seems promising due to these benefits, RDF also has some disadvantages. As the structure of RDF is index-based, it may become inconvenient for larger queries and path analysis (Sağlam, 2018). As RDF is triple store-based and makes use of expressions to capture relationships, capturing even simple relationships can sometimes produce big queries. A way to tackle this is the introduction of property graphs (Knight, 2021).

2.2 Property graph databases

Property graphs are introduced to tackle some of the problems RDF databases face. Property graphs are graphs that consist of nodes, which represent entities, as well as edges, which represent relationships between nodes, and properties, which represent features of the nodes and relationships (Angles, 2018). A general structure of a labeled property graph is shown in Figure 1.

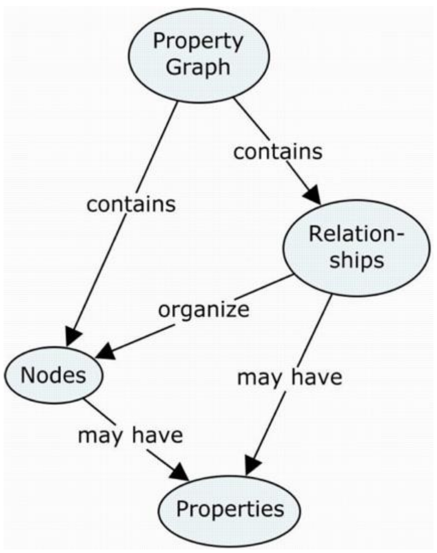


Figure 1: Structure of a labeled property graph (Frisendal, 2019).

Figure 1 shows the structure of a labeled property graph. The property graph contains both nodes and relationships. These nodes and relationships may contain properties as well. Property graphs make use of the query language Cypher. Inspired by SQL, Cypher is Neo4J's query language where patterns and relationships are provided in a visual way (Neo4J, n.d.).

2.2.1 Pros and cons of property graph databases

Property graph databases have some advantages in comparison to RDF-triple stores. One major benefit is that both nodes and edges may contain properties. As such, this type of database is better optimized for path analysis (Maturana, 2015). This tackles the problem of having to generate big queries to capture relationships. Another advantage is that multiple query languages can be used in a property graph database as opposed to RDF, which mainly uses SPARQL. Although this type of database supports multiple query languages, it generally focuses on Cypher (Maturana, 2015). The type of database that is used, differs per situation and depends heavily on the preference of the data modeler.

3 Methods

In order to gain more insight into what data modeling is, and in specific which steps are included in data modeling, research is done through desk research. IBM states: ‘ *Creating a visual representation of an information system to communicate connections between structures and data points is the process called data modeling.’* (IBM Cloud Education, 2020) The whole process knows multiple steps to it and can be divided into the following:

1. Identifying entities
2. Identifying attributes of these entities
3. Identifying relationships amongst these entities
4. Mapping the attributes to the entities
5. Find a balance between normalization and performance; this means finding a balance between reducing data storage and query performance
6. Finalize and validate the data model

Even though many different fundamental types of databases exist nowadays, this research focuses on storing the relationships between the data and presenting this in a convenient way. As such, the most convenient way to store the data would be in a graph database (Robinson, Webber, & Eifrem, 2015). A graph database is optimized for handling highly connected data and provides insights that are not easily found using other techniques (Bechberger, 2019). For this research, a property graph database will be used to store and present the data in a labeled property graph.

3.1 Programs

The composition of the model to store and display the financial transaction data will be done in Python and Neo4J. Neo4J is an open-source graph data platform that can be used for native graph storage (Neo4J, n.d.). The modeling is done in Python and through the package Py2Neo, the data can be sent from Python to Neo4J to store the data into a graph database and display the network as a graph.

3.2 Model structures

A financial transaction network is a directed network and the resulting labeled property graph can be of different structures. In this research, two different structures of labeled property graphs will be developed and compared. Therefore, within the developed model, two different networks will be created. One where the transactions are generated as nodes, and one where transactions are generated as edges. In both cases, properties are given to the transactions. If transactions are generated as nodes, the properties will be node properties and edge properties when transactions are generated as edges. The constructed graphs will thus have different topological properties (Millán, 2017). The following subchapters will dive deeper into these different network structures.

3.2.1 Network structure 1

In the first model, network 1, the account where the transaction is from (source) and the account of the receiving end of the transaction (destination) will be generated as nodes. The properties of these accounts will be node properties. In this model, the transactions will also be generated as nodes. This means that as the transactions are nodes, the properties of the transactions will also be node properties. In this model, two types of nodes are generated: Account nodes and Transaction nodes. These nodes are related by two types of edges: *Transaction from* and *Transaction to*. These edges correspond to the flow of the funds, but do not contain any properties. The edge *Transaction from* connects the source Account node to the Transaction node. The edge *Transaction to* connects the Transaction node to the destination Account node, completing the network. Figure 2 shows a schematic overview of this network structure.

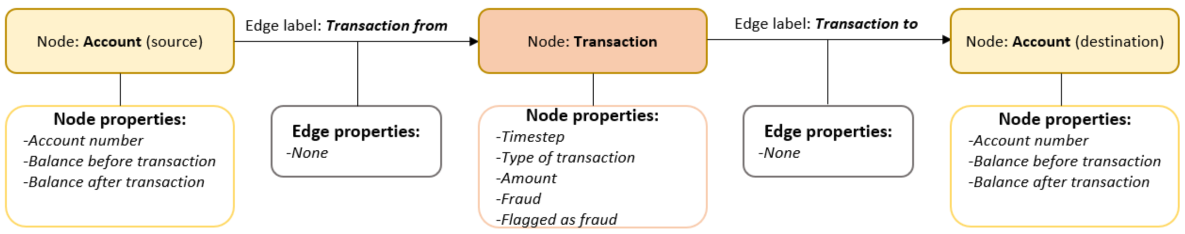

Node: Account (source)
Edge label: Transaction from
Node: Transaction
Edge label: Transaction to
Node: Account (destination)
Node properties: -Account number -Balance before transaction -Balance after transaction
Edge properties: -None
Node properties: -Timestep -Type of transaction -Amount -Fraud -Flagged as fraud
Edge properties: -None
Node properties: -Account number -Balance before transaction -Balance after transaction

Figure 2: Schematic overview of the network structure for one transaction for network 1.

Figure 2 shows the schematic overview of the financial transaction network. It shows the two types of nodes: Account nodes and Transaction nodes, and properties belonging to these nodes. Properties of the Account nodes consist of: *Account number, Balance before transaction,* and *Balance after transaction*. Properties of the Transaction nodes consist of: *Timestep, Type of transaction, Amount, Fraud, and Flagged as fraud*. Figure 2 also shows two edges, with their labels *Transaction from* and *Transaction to,* connecting the nodes in this network. In this model, the edges contain no properties. Modeling the transactions as nodes might be relevant to extract more meaningful information about the graph in terms of topological properties, as the generated graph will contain more nodes and edges.

3.2.2 Network structure 2

The second model, network 2, generates the transactions as edges between the Account nodes. This means that in this model, the properties of the transactions will be edge properties and the properties of accounts are node properties. In this model, only one type of node is generated: Account nodes. These nodes are then related by one type of edge: *Transaction*. This edge corresponds to the flow of the funds and will, in contrast to the previous network structure, contain edge properties. The edge *Transaction* connects the source Account node to the destination Account node. A general schematic overview of this network structure is shown in Figure 3 below.

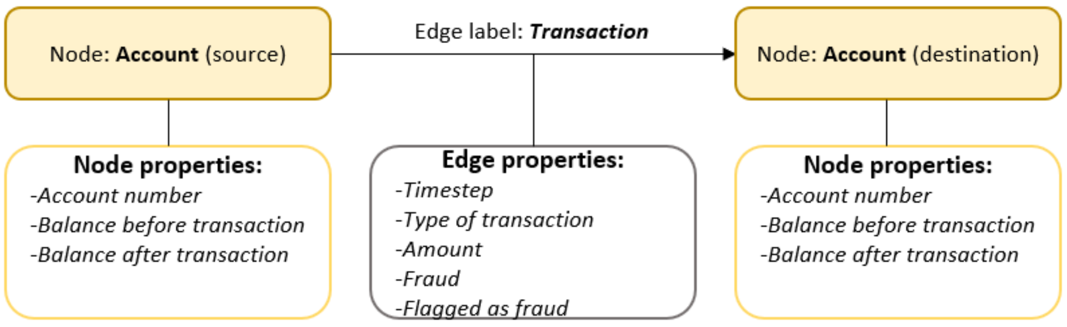

Node: Account (source)
Edge label: Transaction
Node: Account (destination)
Node properties: -Account number -Balance before transaction -Balance after transaction
Edge properties: -Timestep -Type of transaction -Amount -Fraud -Flagged as fraud
Node properties: -Account number -Balance before transaction -Balance after transaction

Figure 3: Schematic overview of the network structure for one transaction for network 2.

Figure 3 shows the schematic overview of the financial transaction network for network 2. It shows two nodes of the same type: source Account and destination Account, and properties belonging to these nodes. Properties of the Account nodes consist of: *Account number, Balance before transaction,* and *Balance after transaction*. Figure 3 also shows the edge, with label *Transaction*, connecting the nodes of this network. In this model, the edge contains properties. These edge properties consist of: *Timestep,* *Type of transaction, Amount, Fraud, and Flagged as fraud*. As this model generates fewer nodes and edges, compared to model 1, this model will be computationally more efficient. However, this graph structure will have influence on the topological properties.

4 Data

This chapter elaborates on financial transaction data. It looks at the structure of financial transaction data and how such data is constructed. In order to develop the model for storing financial transaction data as generalized as possible, this research looks at which entities are often used in this kind of data, which relationships can be found in the data, and which properties and features it contains.

4.1 Description of financial transaction data

Financial transaction data is data that consists of captured information from financial transactions (TIBCO, 2021). The European System of Accounts states that financial transaction consists of financial assets and liabilities (European Comission, Eurostat, 2014). Since financial transaction data can contain a time element, this data can be presented as a network of transactions. This data is then found in either of two ways when it is stored in CSV files:

- A CSV containing both nodes and edges (relationships).
- Multiple CSV files, usually two, where the nodes and edges are divided over the two CSV files.

To find which entities, relationships, properties, and features often exist in this kind of data, different datasets containing financial transaction data are examined. In the EU a lot of regulations around data privacy are present. Since May 2018 the Global Data Protection Regulation, GDPR, is effective and with this regulation, laws regarding the protection of a customer's personal information became stricter (Hoofnagle, van der Sloot, & Zuiderveen Borgesius, 2019). Because of this, it is rather difficult to acquire financial transaction data for this research.

The properties that can be stored in the graph data model are based on properties that exist in two different financial transaction datasets that are public. The first dataset, *IEEE-CIS Fraud Detection*, is a dataset containing financial transactions that was made public by Vesta Corporation on Kaggle in order to detect fraud in customer transactions. The second is the *PaySim* dataset, which is a synthetically generated dataset, see 4.2.

4.2 PaySim dataset

The dataset which is used in this research to validate the model on is a synthetically generated dataset. This dataset was generated by a simulator called *PaySim* (Lopez-Rojas, Elmir, & Axelsson, 2016). *PaySim* can generate synthetic datasets that are similar to datasets of mobile money payments. It simulates data based on a sample of real transactions which are extracted from logs of a mobile service.

The *PaySim* dataset used in this research is a simulation of mobile money transactions for 1 month's time. This dataset consists of one CSV file of 6.362.620 transactions and 11 different variables:

1. **step**: Timestep that represents one hour. As this dataset is simulated for a month, a total of 744 steps are present in the dataset.
2. **type**: The type of transaction. In total five different types of transactions are included; CASH-IN, CASH-OUT, DEBIT, PAYMENT, and TRANSFER.
3. **amount**: The amount of the transaction.
4. **nameOrig**: The name of the account where the transaction comes from (source account).
5. **oldbalanceOrg**: Balance of the source account before the transaction.
6. **newbalanceOrig**: Balance of the source account after the transaction.

7. **nameDest**: Name of the account receiving the transaction (destination account).
8. **oldbalanceDest**: Balance of the destination account before the transaction.
9. **newbalanceDest**: Balance of the destination account after the transaction.
10. **isFraud**: This variable states whether the simulated transaction was fraudulent or not.
11. **isFlaggedFraud**: In this dataset transfers of big amounts are controlled. As such, transfers of an amount of 200.000 or greater are flagged as an illegal attempt and thus flagged as fraud.

Some incompleteness in this dataset exists, one example being the balance of accounts. In some cases, the new balance does not match the old balance after the transaction has been done. This is both the case for the source accounts and the destination accounts. However, as this dataset is only used to validate the model, this does not influence the results of this research.

As each timestep represents one hour in the data, each timestep consists of different amounts of transactions. This can be seen in Table 1, which shows the number of transactions within the first 5 timesteps.

Table 1: Amount of transactions within each timestep for the first 5 steps

	Amount of transactions
Timestep 1	2708
Timestep 2	1014
Timestep 3	552
Timestep 4	565
Timestep 5	665

An example of how one payment of the *PaySim* dataset, when transactions are generated as nodes, looks like is shown in Figure 4.

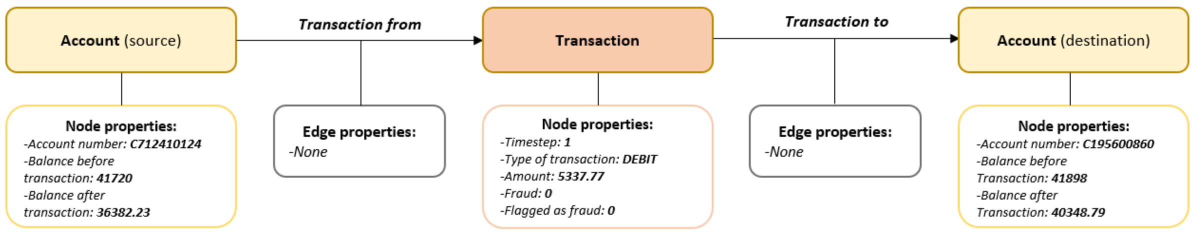


Figure 4: Schematic overview of the network structure for one transaction of the *PaySim* dataset for network 1.

An example of how one payment of the *PaySim* dataset looks like when transactions are generated as edges is shown in Figure 5.

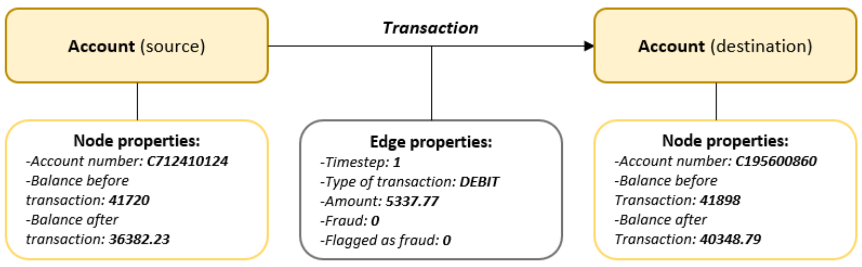

Account (source)
Transaction
Account (destination)
Node properties:
-Account number: C712410124
-Balance before transaction: 41720
-Balance after transaction: 36382.23
Edge properties:
-Timestep: 1
-Type of transaction: DEBIT
-Amount: 5337.77
-Fraud: 0
-Flagged as fraud: 0
Node properties:
-Account number: C195600860
-Balance before Transaction: 41898
-Balance after Transaction: 40348.79

Figure 4: Schematic overview of the network structure for one transaction of the *PaySim* dataset for network 2.

4.3 Properties of the data model

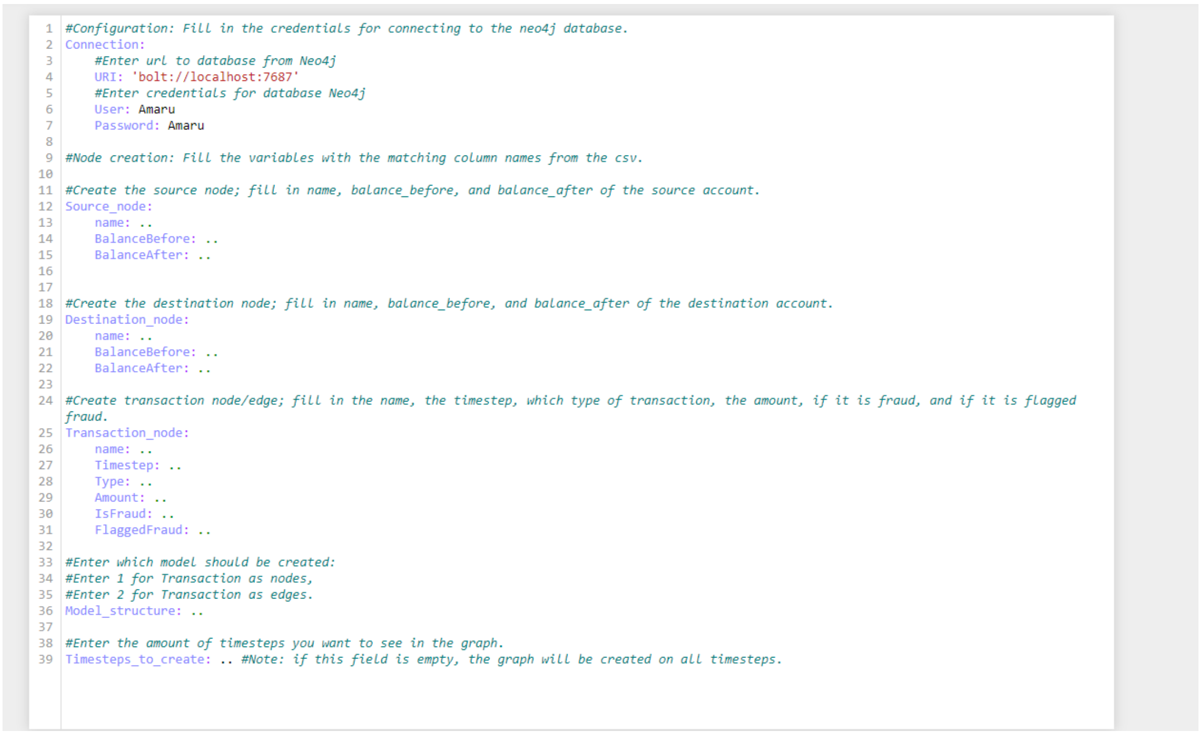
The properties of the data that can be stored in the graph data model are based on the properties that are included in the two datasets: *IEEE-CIS Fraud* Detection and *PaySim*. As such, the properties that can be stored in the graph data model are all the properties the *PaySim* dataset contains. These properties are also included in the *IEE-CIS Fraud Detection* dataset and thus are likely to be common properties of financial transaction data. However, if data is stored in the graph database containing more properties than those included in the *PaySim* dataset, this data will be lost.

5 Implementation

The generated graph data models are modeled in Python via the Py2Neo package, as explained in chapter 3. With this package, the stored data in the graph database can be sent to Neo4J to create a graph. Both models are validated on the *PaySim* dataset and this chapter elaborates on the implementations.

5.1 Configuration file

As the data models have to be generalized, a configuration file has been composed as a YAML-filetype, where parameters of the data can be given. An overview of what this configuration file looks like is shown in Figure 4. At the start of the configuration file, the credentials to connect to the graph database in Neo4j have to be entered, see Figure 4.



```
 1  #Configuration: Fill in the credentials for connecting to the neo4j database.
 2  Connection:
 3      #Enter url to database from Neo4j
 4      URI: 'bolt://localhost:7687'
 5      #Enter credentials for database Neo4j
 6      User: Amaru
 7      Password: Amaru
 8
 9  #Node creation: Fill the variables with the matching column names from the csv.
10
11  #Create the source node; fill in name, balance_before, and balance_after of the source account.
12  Source_node:
13      name: ..
14      BalanceBefore: ..
15      BalanceAfter: ..
16
17
18  #Create the destination node; fill in name, balance_before, and balance_after of the destination account.
19  Destination_node:
20      name: ..
21      BalanceBefore: ..
22      BalanceAfter: ..
23
24  #Create transaction node/edge; fill in the name, the timestep, which type of transaction, the amount, if it is fraud, and if it is flagged
    fraud.
25  Transaction_node:
26      name: ..
27      Timestep: ..
28      Type: ..
29      Amount: ..
30      IsFraud: ..
31      FlaggedFraud: ..
32
33  #Enter which model should be created:
34  #Enter 1 for Transaction as nodes,
35  #Enter 2 for Transaction as edges.
36  Model_structure: ..
37
38  #Enter the amount of timesteps you want to see in the graph.
39  Timesteps_to_create: .. #Note: if this field is empty, the graph will be created on all timesteps.
```

Figure 4: Configuration file for the graph data model of type YAML.

At the start of the configuration file, the credentials to connect to the graph database in Neo4j have to be entered. These contain the URI, the username, and the password of the graph database.

In the next section, properties can be given to the source node. For this, the name of the source account, the balance before the transaction, and the balance after the transaction have to be entered. These variables will have to be of the same name as their respective column headers in the CSV file. The same information is required for the destination account in the following section, lines 19 – 22.

In the next section, lines 25 – 31, properties of the transaction have to be filled in. These will either be node properties or edge properties, depending on the structure of the network, see chapter 3. These transaction properties consist of the name of the transaction, the timestep, the type, the amount, whether the transaction was fraudulent, and whether the transaction was flagged as fraudulent.

Hereafter, the variable Model_structure has to be defined. This variable can contain either the value 1 so that the transactions will be generated as nodes with their respective node properties, or value 2 so that the transactions will be generated as edges with their respective edge properties.

The final variable, *Timesteps_to_create*, ensures that the network will be generated for the number of timesteps that are entered. If this field is left empty, the network will be generated for all timesteps.

5.2 Graph Data Model

In this section, the full Python script of the graph data model will be reviewed per section. The whole code is added in the Appendix. In the first part of the graph data model, a YAML file reader is defined so that the configuration file can be read, see Figure 5.

```python
def yaml_loader(filepath):
    """Loads a yaml file"""
    with open(filepath, "r") as file_descriptor:
        data = yaml.safe_load(file_descriptor)
    return data

filepath = "Configuration.yaml"
config_data = yaml_loader(filepath)
```

Figure 5: Function to read the configuration file in.

This YAML file reader reads the Configuration.yaml file that is in the same working directory as the Python script of the graph data model and reads the information in as *config_data.* Next, the connection is made between the Python script and Neo4J with the following section of the code:

```python
#Make the connection to neo4j
connection = config_data['Connection']
graph = Graph(connection['URI'], user=connection['User'], password=connection['Password'])
```

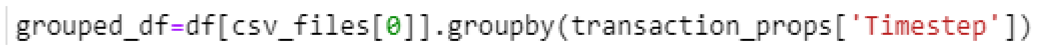
Figure 6: Making the connection to the graph database system, Neo4J.

A function for automated CSV reading is implemented in the code as well, see Figure 6.

```python
def csv_files():

    #get names of only csv files in the directory
    csv_files = []
    for file in os.listdir(os.getcwd()):
        if file.endswith('.csv'):
            csv_files.append(file)

    return csv_files


def create_df(csv_files):

    #path to the csv files
    #data_path = os.getcwd()+'\\'+dataset_dir+'\\'
    data_path = os.getcwd()+'\\'

    #loop through the files and create the dataframe
    df = {}
    for file in csv_files:
        try:
            df[file] = pd.read_csv(data_path+file)
        except UnicodeDecodeError:
            df[file] = pd.read_csv(data_path+file, encoding="ISO-8859-1")

    return df
```

Figure 7: Function for automated CSV reading.

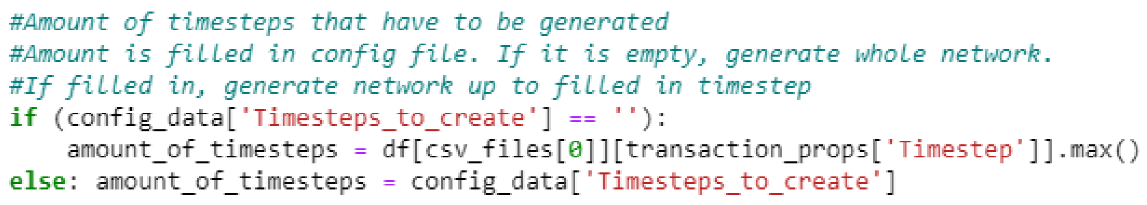
By implementing this automated function for CSV reading, multiple CSV files can be handled that are in the same working directory as the Python script. Next, the financial transaction data is grouped by their timestep so that the network can be generated for a certain amount of timesteps, see Figure 8.



```python
grouped_df=df[csv_files[0]].groupby(transaction_props['Timestep'])
```

Figure 8: The financial transaction data is grouped by timesteps.

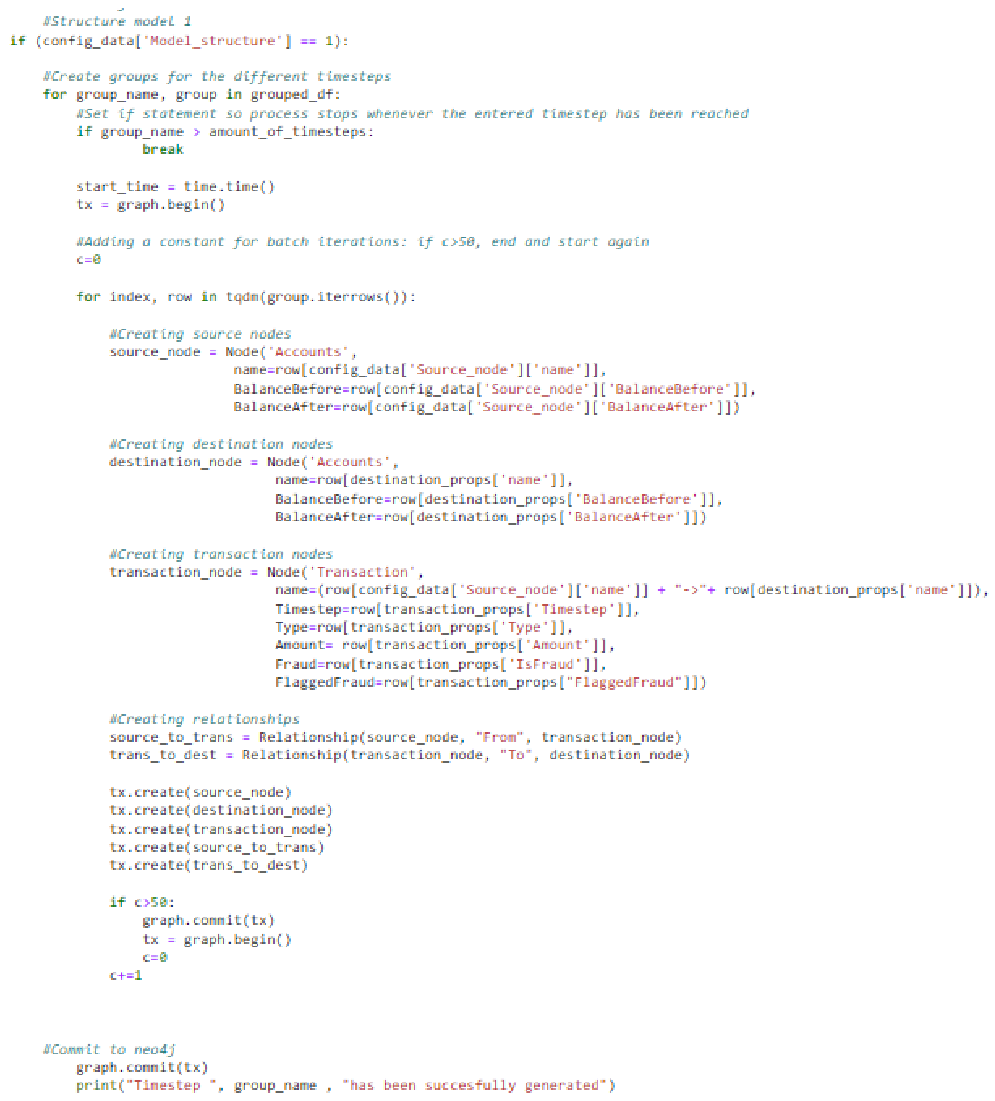
Following this, the next part is created to read in the number of timesteps the network has to be generated for. If no amount has been entered in the configuration file, the network will be generated for all timesteps, see Figure 9.



```python
#Amount of timesteps that have to be generated
#Amount is filled in config file. If it is empty, generate whole network.
#If filled in, generate network up to filled in timestep
if (config_data['Timesteps_to_create'] == ''):
    amount_of_timesteps = df[csv_files[0]][transaction_props['Timestep']].max()
else: amount_of_timesteps = config_data['Timesteps_to_create']
```

Figure 9: Reading in the number of timesteps for the network from the configuration file.

The following part of the code is developed to generate the network graph of model structure 1, where transactions are generated as nodes. See Figure 10.

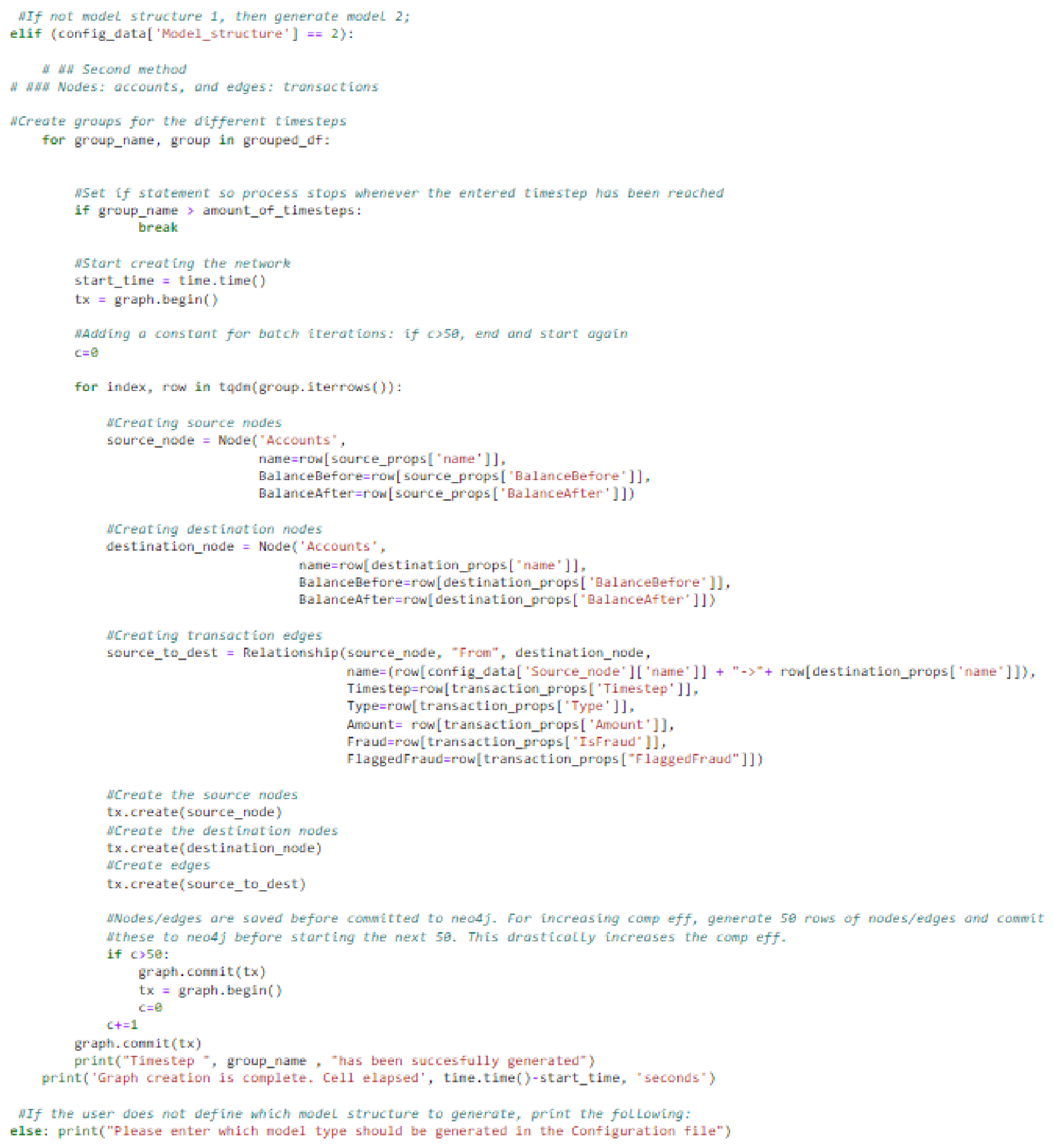


```
    #Structure model 1
if (config_data['Model_structure'] == 1):

    #Create groups for the different timesteps
    for group_name, group in grouped_df:
        #Set if statement so process stops whenever the entered timestep has been reached
        if group_name > amount_of_timesteps:
                break

        start_time = time.time()
        tx = graph.begin()

        #Adding a constant for batch iterations: if c>50, end and start again
        c=0

        for index, row in tqdm(group.iterrows()):

            #Creating source nodes
            source_node = Node('Accounts',
                               name=row[config_data['Source_node']['name']],
                               BalanceBefore=row[config_data['Source_node']['BalanceBefore']],
                               BalanceAfter=row[config_data['Source_node']['BalanceAfter']])

            #Creating destination nodes
            destination_node = Node('Accounts',
                                    name=row[destination_props['name']],
                                    BalanceBefore=row[destination_props['BalanceBefore']],
                                    BalanceAfter=row[destination_props['BalanceAfter']])

            #Creating transaction nodes
            transaction_node = Node('Transaction',
                                    name=(row[config_data['Source_node']['name']] + "->"+ row[destination_props['name']]),
                                    Timestep=row[transaction_props['Timestep']],
                                    Type=row[transaction_props['Type']],
                                    Amount= row[transaction_props['Amount']],
                                    Fraud=row[transaction_props['IsFraud']],
                                    FlaggedFraud=row[transaction_props["FlaggedFraud"]])

            #Creating relationships
            source_to_trans = Relationship(source_node, "From", transaction_node)
            trans_to_dest = Relationship(transaction_node, "To", destination_node)

            tx.create(source_node)
            tx.create(destination_node)
            tx.create(transaction_node)
            tx.create(source_to_trans)
            tx.create(trans_to_dest)

            if c>50:
                graph.commit(tx)
                tx = graph.begin()
                c=0
            c+=1


    #Commit to neo4j
        graph.commit(tx)
        print("Timestep ", group_name , "has been succesfully generated")
```

Figure 10: Code for the graph data model of structure 1 with transactions as nodes.

The first for loop checks whether the timesteps exceed the entered amount of timesteps the network has to contain. If this is the case, the code stops, and the network is finished. Next, a constant is added so that the data can be generated in batches, see 5.3.2. Hereafter, the data is stored as source-, destination-, and transaction nodes with their respective properties. Following this, the relationships between the nodes, the edges, are generated. The nodes and edges are then created and committed to Neo4J when more than 50 transactions have been processed, see 5.3.2.

If the network graph has to be generated with transactions as edges, model structure 2 is generated. The following code then applies, see Figure 11.



```
 #If not model structure 1, then generate model 2;
elif (config_data['Model_structure'] == 2):

    # ## Second method
# ### Nodes: accounts, and edges: transactions

#Create groups for the different timesteps
    for group_name, group in grouped_df:


        #Set if statement so process stops whenever the entered timestep has been reached
        if group_name > amount_of_timesteps:
                break

        #Start creating the network
        start_time = time.time()
        tx = graph.begin()

        #Adding a constant for batch iterations: if c>50, end and start again
        c=0

        for index, row in tqdm(group.iterrows()):

            #Creating source nodes
            source_node = Node('Accounts',
                               name=row[source_props['name']],
                               BalanceBefore=row[source_props['BalanceBefore']],
                               BalanceAfter=row[source_props['BalanceAfter']])

            #Creating destination nodes
            destination_node = Node('Accounts',
                                    name=row[destination_props['name']],
                                    BalanceBefore=row[destination_props['BalanceBefore']],
                                    BalanceAfter=row[destination_props['BalanceAfter']])

            #Creating transaction edges
            source_to_dest = Relationship(source_node, "From", destination_node,
                                          name=(row[config_data['Source_node']['name']] + "->"+ row[destination_props['name']]),
                                          Timestep=row[transaction_props['Timestep']],
                                          Type=row[transaction_props['Type']],
                                          Amount= row[transaction_props['Amount']],
                                          Fraud=row[transaction_props['IsFraud']],
                                          FlaggedFraud=row[transaction_props["FlaggedFraud"]])

            #Create the source nodes
            tx.create(source_node)
            #Create the destination nodes
            tx.create(destination_node)
            #Create edges
            tx.create(source_to_dest)

            #Nodes/edges are saved before committed to neo4j. For increasing comp eff, generate 50 rows of nodes/edges and commit
            #these to neo4j before starting the next 50. This drastically increases the comp eff.
            if c>50:
                graph.commit(tx)
                tx = graph.begin()
                c=0
            c+=1
        graph.commit(tx)
        print("Timestep ", group_name , "has been succesfully generated")
    print('Graph creation is complete. Cell elapsed', time.time()-start_time, 'seconds')

 #If the user does not define which model structure to generate, print the following:
else: print("Please enter which model type should be generated in the Configuration file")
```

Figure 11: Code for generating the network with transactions as edges.

This code is nearly the same as the code for the network of model structure 1. The difference lies between the creation of the relationships, the edges. In this part, the transaction is generated as an edge (*source_to_dest*) and thus contains edge properties. If no structure of network is entered in the configuration file, the output of the code is a statement stating: "*Please enter which model type should be generated in the Configuration file*".

5.3 Model Performance

As described in chapter 3, the structure of the model is of high importance as to how well the model performs. To validate the model and measure its performance the *PaySim* dataset is stored using the developed models. The performance is then measured by storing the first 100 elements of this dataset and presenting them in a graph. This is also done for the first timestep of this dataset, containing 2708 elements.

5.3.1 Performance measures

The structure of the generated model will be of high importance as to how well a model performs. For this research three main measures of performance are looked at to determine how well the models are modeled:

- Generalization: How well the model can be applied to different datasets.
- Completeness of data storage: How well the data is stored in its completion.
- Computational efficiency: How well the model performs in terms of speed.

5.3.2 Model network structure 1

The results of the average time it took to store the first 100 elements of the *PaySim* dataset, and the first timestep, containing 2708 elements, are shown in Table 2. These results are based on averaging the time it took to store the data and generate the network graph five different times.

Table 2: Results of storing the data and generating the network graph for the first 100 elements and first timestep.

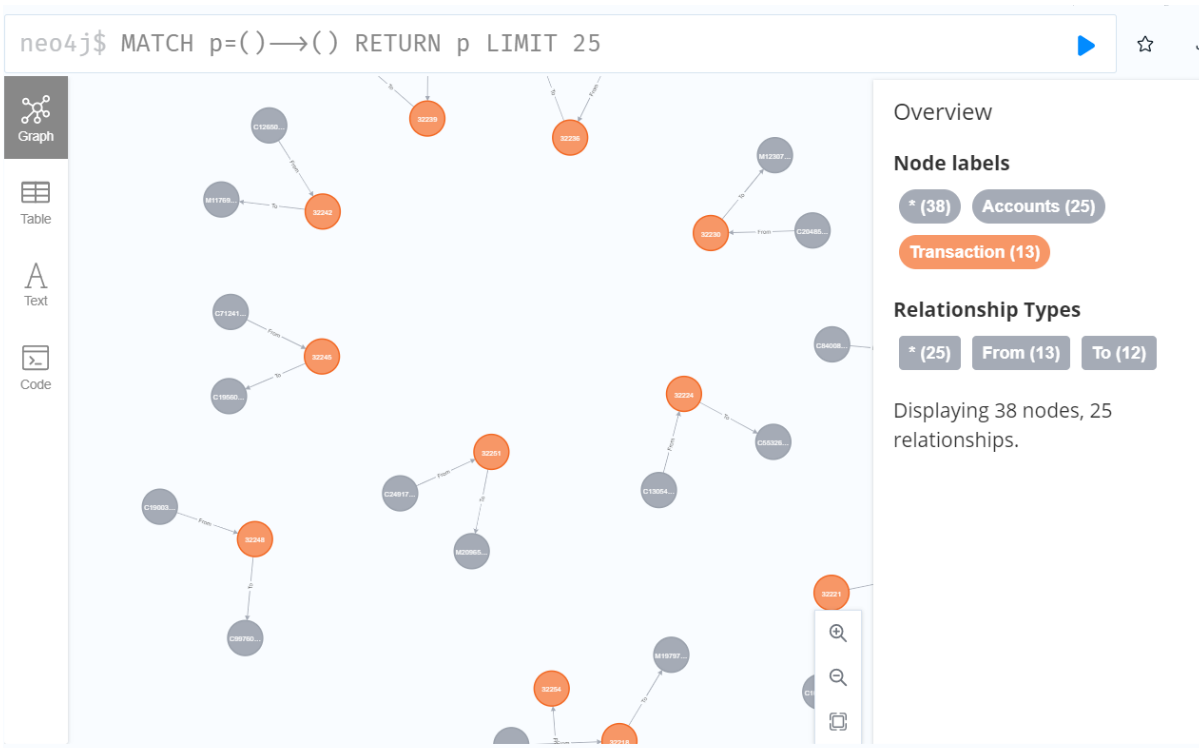
	Average time
First 100 elements	1.945 seconds
First timestep (2708 elements)	546.884 seconds

As the data is stored in nodes and relationships before the graph is created, the amount of iterations per second of the model drops immensely as time progresses. Upon the start of running the code, the number of iterations per second is on average 130 It/s. After around 200 iterations, this amount of iterations per second drops down to 3 It/s. By implementing the method to load the data and create the graph in batches, a high amount of iterations per second can be maintained as every time a batch is generated, the iterations per second are the same as at the start of the process, averaging 130 It/s. This model stores and loads the data in batches of 50. The results of the average time it took to store the first time step are shown in Table 3. This result is based on averaging the time it took to store the data and generate the network graph five different times.

Table 3: Results of storing the data and generating the graph by implementing batch sizes.

	Average time
First timestep	36.177 seconds

Figure 12 shows what the graph network looks like in Neo4J when the transactions are modeled as edges, see Figure 12.


neo4j$ MATCH p=()⟶() RETURN p LIMIT 25

Overview

Node labels

* (38) Accounts (25) Transaction (13)

Relationship Types

* (25) From (13) To (12)

Displaying 38 nodes, 25 relationships.

Figure 12: Overview of the structure of the financial transaction network graph with transactions as nodes in Neo4J.

5.3.3 Model network structure 2

The results of the average time it took to store the first 100 elements of the *PaySim* dataset, and the first timestep where transactions are edges, are shown in Table 4. These results are based on averaging the time it took to store the data and generate the network graph five different times.

Table 4: Results of storing the data and generating the network graph for the first 100 elements and first timestep.

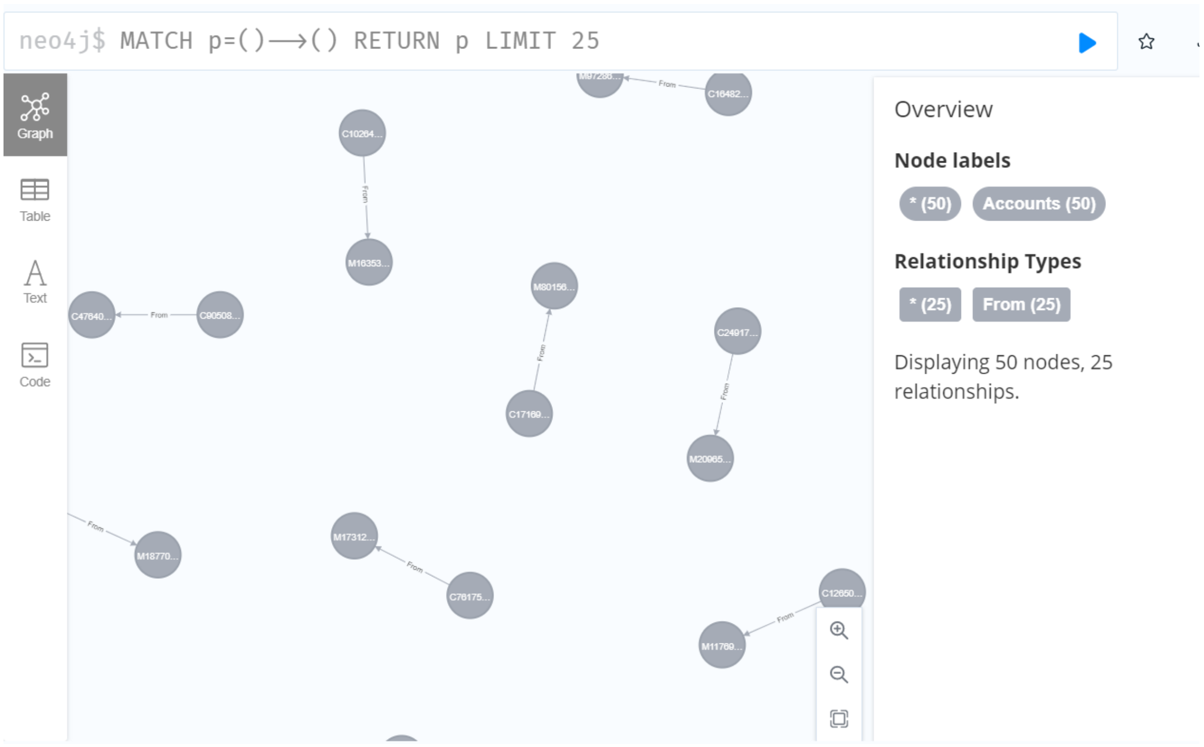
	Average time
First 100 elements	1.327 seconds
First timestep	219.534 seconds

The results of the average time it took to store the first timestep, when the data is stored and the network is generated in batches of 50, are shown in Table 5. This result is based on averaging the time it took to store the data and generate the network graph five different times.

Table 5: Results of storing the data and generating the graph by implementing batch sizes.

	Average time
First timestep	14.034 seconds

Figure 13 shows what the graph network looks like in Neo4J when the transactions are modeled as edges, see Figure 13.



```
neo4j$ MATCH p=()⟶() RETURN p LIMIT 25
```

Overview

Node labels

* (50) Accounts (50)

Relationship Types

* (25) From (25)

Displaying 50 nodes, 25 relationships.

Figure 13: Overview of the structure of the financial transaction network graph with transactions as edges in Neo4J.

6 Conclusion and discussion

This report elaborates on the research for developing a model for storing temporal financial transaction data. In chapter 2 the formulation of this research question is divided into sub-questions. These sub-questions will be answered in this chapter. Finally, recommendations are made for possible follow-up research.

6.1 Research conclusions

Due to the strict regulations concerning data privacy, financial transaction data is hard to acquire. As such, this research makes use of the *PaySim* dataset, a synthetically generated dataset, see chapter 4. This dataset is used to validate and measure the performance of the developed graph data model. The data is then stored in a graph database so that a labeled property graph can be generated. This type of graph works best for displaying financial transactions due to the possibility to store the data as nodes and edges with properties.

As seen in chapter 5, the structure of the network influences the time it takes to store the data in the graph database and generate the network graph. The developed graph data model can generate two different network structures, one with transactions as nodes, and one with transactions as edges. The properties of the transaction data will thus be different per scenario. These properties will be node properties if the transactions are stored and presented as nodes, and edge properties if the transactions are stored and presented as edges. The computational efficiency is measured by the time it took to store the first 100 transactions and all the transactions of the first timestep of the *PaySim* dataset. This is then measured five times with the result being the average of the five measurements. It takes the model 546.883 seconds to store the transactions of the first timestep and create a graph when the transactions are generated as nodes. However, when transactions are generated as edges, the time it takes for the model to run is 219.534 seconds, which is 59.86% faster than the time it takes to run the model with transactions as nodes. This increase in performance time is due to the number of nodes and edges that have to be created. When transactions are represented as nodes, a single transaction will consist of three nodes and two edges. However, when transactions are represented as edges, a single transaction will consist of two nodes and one edge. Therefore, fewer nodes and edges have to be created to store the data and to create the graph.

The longer the model runs, a decrease in iterations per second is observed. This results in a decrease in performance time of the model. At the start of the run, the model performs with 130 It/s and quickly drops down to 3 It/s, resulting in complete data storage and graph creation for the network with transactions as nodes and the network with transactions as edges in respectively 546.833 seconds and 219.534 seconds. To increase the performance of the model, the data is loaded and stored in batches of 50. This ensures a high amount of iterations are maintained throughout the process, as the number of iterations per second will reset after the batch is generated, averaging around 130 It/s. This reduces the time it takes to store the data and to generate the network graph for the transactions of the first timestep of the *PaySim* dataset. This process is reduced by 93.38% of the original time when transactions are stored and modeled as nodes, resulting in 36.177 seconds to complete. It reduces the time it takes to store the data and to generate the network graph where transactions are edges by 93.61% of the original time, resulting in 14.034 seconds to complete the process. As expected, the model with transactions as edges is faster than the model with transactions as nodes. This difference in performance amounts to 61.21%.

The aim of this research was to develop a generalized model that could store temporal financial transaction data and present it in a graph. One step of this process is to load the data in Python from different files, for example, different or multiple CSV files, and store these in the graph database. Even though multiple CSV files can be read in, the developed model can only store data from one single CSV file in the graph database. So if data needs to be extracted from multiple CSV files this model is not suitable. One way to work around this limitation could be to do data preparation in advance, for example store the data from different CSV files into one single CSV file. Another limitation of this model is that the properties that can be stored are hard coded. This means that only the following properties of the data, see chapter 5, can be stored:

- Timestep of transaction;
- Type of transaction;
- Amount of transaction;
- Name of the source account;
- Balance before transaction of the source account;
- Balance after transaction of the source account;
- Name of the destination account;
- Balance before transaction of the destination account;
- Balance after transaction of the destination account;
- Whether the transaction is fraud or not;
- Whether the transaction is flagged as fraud or not.

Because of the predefined set of properties of the data that can be stored in the graph data model, data is lost if more properties are present in the data.

6.2 Recommendations for future research

Even though the developed graph data model stores data in a graph database and generates a financial network graph in Neo4J, this model is still in a very early stage. Future studies could address the way properties of the data are handled. In the current model, a predefined set of properties can be loaded into the model but as some datasets contain more data, the properties should depend on the dataset itself. This way no data is lost in the process of data storage. Additionally, more time could be spent on the structure of the Python code of the model as this has influence on the performance of the model as well.

Another important limitation of this study is the way data is loaded into the model. As stated before, the current model can load multiple CSV files, but can only extract data from a single CSV file. However, financial transaction data can be provided in more than one data file. One way to overcome this limitation would be to do data preparation. For instance, extensive data preparation can be done before loading the data into the database so that all the data is combined into a single data file.

Finally, the model was tweaked and performance was measured based on the first timestep of the *PaySim* dataset. To gain better insights into the performance of the model, the model should be tweaked and performance should be measured based on more data. However, this was not feasible due to the scope of this research.

References

Anadiotis, G. (2017, May 19). *Graph databases and RDF: It's a family affair*. Retrieved from ZDNet: https://www.zdnet.com/article/graph-databases-and-rdf-its-a-family-affair/

Anadiotis, G. (2020, Januari 30). *Knowledge graph evolution: Platforms that speak your language*. Retrieved from ZDNet: https://www.zdnet.com/article/knowledge-graph-evolution-platforms-that-speak-your-language/

Angles, R. (2018). The Property Graph Database Model. (AMW, Ed.)

Bechberger, D. (2019, June 17). A Skeptics Guide To Graph Databases. Oslo.

European Comission, Eurostat. (2014). European System of Accounts. *ESA2010*. doi:10.2785/16644

Frisendal, T. (2019, March 11). *The Atoms and Molecules of Data Models*. Retrieved from Dataversity: https://www.dataversity.net/the-atoms-and-molecules-of-data-models/

Hoofnagle, C. J., van der Sloot, B., & Zuiderveen Borgesius, F. (2019). The European Union General Data Protection Regulation: What It Is And What It. *Information & Communications Technology Law*, 65-98. doi:10.1080/13600834.2019.1573501

IBM Cloud Education. (2020, August 25). *Data Modeling*. Retrieved from IBM: https://www.ibm.com/cloud/learn/data-modeling

ING. (n.d.). *Soorten oplichting en fraude*. Retrieved June 17, 2022, from ING: https://www.ing.nl/de-ing/veilig-bankieren/soorten-oplichting-fraude/index.html

Knight, M. (2021, April 28). *What is a Property Graph*. Retrieved from Dataversity: https://www.dataversity.net/what-is-a-property-graph/#

Lopez-Rojas, E. A., Elmir, A., & Axelsson, S. (2016). PaySim: A financial mobile money simulator for fraud detection. *Proceedings of the European Modeling and Simulation Symposium*.

Loshin, P. (n.d.). *Resource Description Framework (RDF)*. Retrieved June 20, 2022, from TechTarget: https://www.techtarget.com/searchapparchitecture/definition/Resource-Description-Framework-RDF#:~:text=A%20consistent%20framework%20encourages%20the,to%20exchange%20information%20more%20easily.

Ma, Z., Capretz, M. M., & Yan, L. (2016, December 7). Storing massive Resource Description Framework (RDF) data: a survey. (C. U. Press, Ed.) doi:10.1017/S0269888916000217

Maturana, R. A. (2015, October 29). *What are the differences between a Graph Database and a Triple Store*. Retrieved from GNOSS: https://nextweb.gnoss.com/en/resource/what-are-the-differences-between-a-graph-database/ced22960-845d-410f-9e3c-5616c603993e#:~:text=RDF%20triple%20stores%20focus%20solely%20on%20storing%20rows%20of%20RDF%20triples.&text=Graph%20databases%20are%20node%2C%

Millán, P. P. (2017). *Network analysis of protein interaction data: an introduction.* doi:10.6019/tol.networks_t.2016.00001.1

Neo4J. (n.d.). *Cypher Query Language*. Retrieved from Neo4j: https://neo4j.com/developer/cypher/

Neo4J. (n.d.). *Neo4j Graph Database*. Retrieved from Neo4J: https://neo4j.com/product/neo4j-graph-database/

OnToText. (2019, October 22). *What is an RDF Triplestore?* Retrieved from OnToText: https://www.ontotext.com/knowledgehub/fundamentals/what-is-rdf-triplestore/#:~:text=LinkedIn%20Twitter%20Facebook-,The%20RDF%20triplestore%20is%20a%20type%20of%20graph%20database%20that,with%20materialized%20links%20between%20them.

Raj, A. (2022, April 11). *IBM Study shows financial fraud a big problem in the US*. Retrieved from TechHQ: https://techhq.com/2022/04/ibm-study-shows-financial-fraud-a-big-problem-in-the-us/

Robinson, I., Webber, J., & Eifrem, E. (2015). *Graph Databases* (Second edition ed.). (I. Neo Technology, Ed.) United States of America: O'Reilly Media, Inc.

Sağlam, U. (2018, January 1). *Neo4j CEO discusses the pros and cons of RDF*. Retrieved from Medium: https://medium.com/@ugur.saglam/neo4j-ceo-discusses-the-pros-and-cons-of-rdf-b6ff6bb9a740

Stefani, E., & Hoxha, K. (2018, November 24). Implementing Triple-Stores using NoSQL Databases.

TIBCO. (2021, June 18). *What is Transactional Data?* Retrieved from TIBCO: https://www.tibco.com/reference-center/what-is-transactional-data

U.S. Department of Justice. (2014, August 21). *Bank of America to Pay $16.65 Billion in Historic Justice Department*. Retrieved from Office of Public Affairs: https://www.justice.gov/opa/pr/bank-america-pay-1665-billion-historic-justice-department-settlement-financial-fraud-leading

van Beek, D. (2021). *Data Science for Decision Makers & Data Professionals* (1st edition ed.). Passionned Publishers.

Appendix

I Configuration file

```
#Configuration: Fill in the credentials for connecting to the neo4j database.
Connection:
   #Enter url to database from Neo4j
   URI: 'bolt://localhost:7687'
   #Enter credentials for database Neo4j
   User: Amaru
   Password: Amaru

#Node creation: Fill the variables with the matching column names from the csv.

#Create the source node; fill in name, balance_before, and balance_after of the source account.
Source_node:
   name: nameOrig
   BalanceBefore: oldbalanceOrg
   BalanceAfter: newbalanceOrig


#Create the destination node; fill in name, balance_before, and balance_after of the destination account.
Destination_node:
   name: nameDest
   BalanceBefore: oldbalanceDest
   BalanceAfter: newbalanceDest

#Create transaction node/edge; fill in the name, the timestep, which type of transaction, the amount, if it is fraud, and if it
is flagged fraud.
Transaction_node:
   name: nameDest
   Timestep: step
   Type: type
   Amount: amount
   IsFraud: isFraud
   FlaggedFraud: isFlaggedFraud

#Enter which model should be created:
#Enter 1 for Transaction as nodes,
#Enter 2 for Transaction as edges.
Model_structure: 1

#Enter the amount of timesteps you want to see in the graph.
Timesteps_to_create: 1 #Note: if this field is empty, the graph will be created on all timesteps.
```

II Python code of the model

```
#Importing libraries
import pandas as pd
import py2neo
from py2neo import Graph
from py2neo import Node, Relationship
import time
from tqdm import tqdm
import os, shutil
import yaml

# ### Checking for yaml:
def yaml_loader(filepath):
    """Loads a yaml file"""
    with open(filepath, "r") as file_descriptor:
        data = yaml.safe_load(file_descriptor)
    return data

filepath = "Configuration.yaml"
config_data = yaml_loader(filepath)

#Properties of the nodes:
source_props = config_data['Source_node']
destination_props = config_data['Destination_node']
transaction_props = config_data['Transaction_node']

#Make the connection to neo4j
connection = config_data['Connection']
graph = Graph(connection['URI'], user=connection['User'], password=connection['Password'])

# ### Read in the datasets
# Automation for reading csv is done so that multiple csv's can be read in.
# This might be the case if nodes and edges have different csv's.

# Find CSV's in my current working directory:
#isolate only the CSV files
def csv_files():
    #get names of only csv files in the directory
    csv_files = []
    for file in os.listdir(os.getcwd()):
        if file.endswith('.csv'):
            csv_files.append(file)
    return csv_files

def create_df(csv_files):
    #path to the csv files
    data_path = os.getcwd()+'\\'
    #loop through the files and create the dataframe
    df = {}
    for file in csv_files:
      try:
         df[file] = pd.read_csv(data_path+file)
       except UnicodeDecodeError:
         df[file] = pd.read_csv(data_path+file, encoding="ISO-8859-1")
   return df

csv_files = csv_files()
df = create_df(csv_files)

# Group timesteps (when automatic df read in)
grouped_df=df[csv_files[0]].groupby(transaction_props['Timestep'])

#Amount of timesteps that have to be generated
#Amount is filled in config file. If it is empty, generate whole network.
#If filled in, generate network up to filled in timestep
if (config_data['Timesteps_to_create'] == ' '):
   amount_of_timesteps = df[csv_files[0]][transaction_props['Timestep']].max()
else: amount_of_timesteps = config_data['Timesteps_to_create']


 #######Making the models
   #Structure model 1
if (config_data['Model_structure'] == 1):
   #Create groups for the different timesteps
   for group_name, group in grouped_df:
      #Set if statement so process stops whenever the entered timestep has been reached
      if group_name > amount_of_timesteps:
            break

      start_time = time.time()
      tx = graph.begin()
      #Adding a constant for batch iterations: if c>50, end and start again
      c=0
      for index, row in tqdm(group.iterrows()):
         #Creating source nodes
         source_node = Node('Accounts',
                     name=row[config_data['Source_node']['name']],
                     BalanceBefore=row[config_data['Source_node']['BalanceBefore']],
                     BalanceAfter=row[config_data['Source_node']['BalanceAfter']])

         #Creating destination nodes
         destination_node = Node('Accounts',
                        name=row[destination_props['name']],
                        BalanceBefore=row[destination_props['BalanceBefore']],
                        BalanceAfter=row[destination_props['BalanceAfter']])

         #Creating transaction nodes
         transaction_node = Node('Transaction',
                        name=(row[config_data['Source_node']['name']] + "->"+ row[destination_props['name']]),
                        Timestep=row[transaction_props['Timestep']],
                        Type=row[transaction_props['Type']],
                        Amount= row[transaction_props['Amount']],
                    Fraud=row[transaction_props['IsFraud']],
                    FlaggedFraud=row[transaction_props["FlaggedFraud"]])

         #Creating relationships
         source_to_trans = Relationship(source_node, "From", transaction_node)
         trans_to_dest = Relationship(transaction_node, "To", destination_node)

         tx.create(source_node)
         tx.create(destination_node)
         tx.create(transaction_node)
         tx.create(source_to_trans)
         tx.create(trans_to_dest)

#Nodes/edges are saved before committed to neo4j. For increasing comp eff, generate 50 rows of nodes/edges and commit
         #these to neo4j before starting the next 50. This drastically increases the comp eff
         if c>50:
           graph.commit(tx)
           tx = graph.begin()
           c=0
         c+=1

   #Commit to neo4j
     graph.commit(tx)
     print("Timestep ", group_name , "has been succesfully generated")
   print('Graph creation is complete. Cell elapsed', time.time()-start_time, 'seconds')

#If not model structure 1, then generate model 2;
elif (config_data['Model_structure'] == 2):

   # ## Second method
# ### Nodes: accounts, and edges: transactions

#Create groups for the different timesteps
   for group_name, group in grouped_df:

     #Set if statement so process stops whenever the entered timestep has been reached
     if group_name > amount_of_timesteps:
         break

     #Start creating the network
     start_time = time.time()
     tx = graph.begin()

     #Adding a constant for batch iterations: if c>50, end and start again
     c=0
     for index, row in tqdm(group.iterrows()):
       #Creating source nodes
       source_node = Node('Accounts',
                    name=row[source_props['name']],
                    BalanceBefore=row[source_props['BalanceBefore']],
                    BalanceAfter=row[source_props['BalanceAfter']])
       #Creating destination nodes
```

```
        destination_node = Node('Accounts',
                    name=row[destination_props['name']],
                    BalanceBefore=row[destination_props['BalanceBefore']],
                    BalanceAfter=row[destination_props['BalanceAfter']])


        #Creating transaction edges
        source_to_dest = Relationship(source_node, "From", destination_node,
                         name=(row[config_data['Source_node']['name']] + "->"+ row[destination_props['name']]),
                         Timestep=row[transaction_props['Timestep']],
                         Type=row[transaction_props['Type']],
                         Amount= row[transaction_props['Amount']],
                         Fraud=row[transaction_props['IsFraud']],
                         FlaggedFraud=row[transaction_props["FlaggedFraud"]])

        #Create the source nodes
        tx.create(source_node)
        #Create the destination nodes
        tx.create(destination_node)
        #Create edges
        tx.create(source_to_dest)

        #Nodes/edges are saved before committed to neo4j. For increasing comp eff, generate 50 rows of nodes/edges and
        commit
        #these to neo4j before starting the next 50. This drastically increases the comp eff.
        if c>50:
           graph.commit(tx)
           tx = graph.begin()
           c=0
        c+=1
      graph.commit(tx)
      print("Timestep ", group_name , "has been succesfully generated")
   print('Graph creation is complete. Cell elapsed', time.time()-start_time, 'seconds')

  #If the user does not define which model structure to generate, print the following:
else: print("Please enter which model type should be generated in the Configuration file")
```