POLITECNICO DI TORINO & TÉLÉCOM PARIS

Master degree course in Data Science and Artificial Intelligence

Master Degree Thesis

Use of Graph Databases in Financial Crime Compliance

Supervisors
prof. Paolo Garza
prof. David Bounie

Candidate
Marco Proietto
matricola: 295671

Internship Tutors
Grégory GARBAROVITSCH
Emmanuel BOYERO

Accademic Year 2023-2024

Summary

The following thesis explores the growing importance of graph databases, particularly for financial crime compliance, as traditional relational databases struggle to handle complex and interconnected datasets.

Graph databases such as Neo4j offer significant advantages in detecting fraud and ensuring regulatory compliance by analyzing relationships between data points in real time, enabling institutions to better combat evolving financial crimes.

Acknowledgements

Ai miei genitori.
A mia madre, la donna più forte che io conosca, che ogni giorno mi ricorda cosa sono l'amore e il sacrificio, e a mio padre, che nonostante non può dimostrarlo, so che è immensamente fiero di me.
Ai miei nonni e alla mia famiglia, che anche a migliaia di chilometri di distanza riesco a sentire sempre vicini a me.
Grazie, e scusatemi se sono sempre così lontano.

A mio fratello, la mia guida da sempre, senza il quale non avrei neanche iniziato questo percorso.
Grazie per lasciarmi seguire le tue orme.

Ai compagni che ho avuto accanto in questo percorso, per le sessioni di studio iniziate in casa e finite su Skype.
Grazie per questa opportunità immensa.

Ai miei amici, ormai sparsi in ogni angolo d'Italia, d'Europa e del Mondo.
Grazie per supportarmi e sopportarmi.

A me, che nonostante tutto, alla fine ce l'ho fatta.
Grazie.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In today's rapidly evolving technological landscape, the ability to manage and analyze complex data structures is becoming increasingly essential, particularly within the realm of financial crime compliance where the exponential growth of digital financial transactions has highlighted the need for advanced security mechanisms to protect sensitive data and prevent fraudulent activities.

Traditional relational databases, while fundamental, have significant limitations in managing complex, interconnected financial datasets. The use of **Graph Databases**, with their native ability to represent and traverse relationships between data points, offers a perfect alternative as resolution for these issues.

Graph databases such as **Neo4j** have revolutionized the way financial institutions address the challenges of detecting fraudulent activity, ensuring regulatory compliance, and analyzing transactional patterns in a continuous and scalable manner.

The motivation behind this study stems from the increasing complexity of financial networks and the need for innovative tools to effectively manage these complexities. In the context of financial crime compliance, data on customers, transactions, and legal entities are often deeply intertwined, requiring systems that can manage not only individual entities but also their relationships.

The conventional relational database model struggles to perform well under these conditions because of the need for multiple joins, which increase query complexity and slow performance. Graph databases, on the other hand, excel in environments where relationships and connections between data points are priorities.

The effectiveness of graph databases in detecting fraud has been demonstrated in several industries. According to a ***MarketsandMarkets***[1] report, the global market for graph databases is set to grow from $2.9 billion in 2021 to $7.3 billion by 2028, driven in large part by the growing demand for advanced fraud detection solutions in the financial sector.

Financial institutions are investing heavily in graph technology to combat fraud, and a recent survey by the ***Association of Certified Fraud Examiners (ACFE)***[2] found that more than 80% of financial fraud cases involve complex relationships that can be more easily detected using graph analysis.

Graph databases, designed to model and analyze relationships between entities, offer unique advantages over traditional databases when it comes to identifying fraudulent activity. In a graph database, entities such as individuals or accounts are represented as nodes, while connections or relationships between them are represented as edges.

This data model allows real-time analysis of relationships and patterns, facilitating the detection of complex fraud schemes such as money laundering, transaction layering, and identity theft, which often involve multiple entities interacting in a network.

Furthermore financial crimes are not static, they evolve as fraudsters devise new methods to exploit vulnerabilities in financial systems. The ability to dynamically analyze and predict suspicious patterns in real time is critical to staying ahead of the curve.

Graph databases, with their sophisticated algorithms for *community* detection, *pathfinding*, and *centrality* analysis, enable financial institutions to do just that.

For example, using graph-based machine learning techniques, organizations can predict potential fraudulent relationships or identify key characters in fraud networks. As this thesis shows, graph databases not only provide an innovative and efficient approach to data storage and querying, but also enable advanced analytical capabilities that are essential for modern financial compliance.

This thesis focuses on the application of graph databases in the field of financial crime compliance, using Neo4j as a tool to analyze several case studies.

It delves into the limitations of traditional databases, the advantages offered by graph structures, and how these databases can be leveraged to improve regulatory compliance and fraud detection efforts.

The importance of this research lies in its potential to improve the tools and methods available to financial institutions, regulators and investigators, giving them a more powerful means to combat financial crime in an increasingly connected world.

Chapter 2

Use Of The Graph Databases in Financial Security

In this chapter i will provide an overview of graphs and their relevance in the context of data structure, including the different types of graphs and their characteristics.

I will talk about the challenges posed by relational databases in handling intricate financial data, leading to the exploration of graph databases as a viable alternative, underlying the advantages and improvements brought by their adoption, also adding a brief introduction on possible uses in the financial field.

2.1 What is a Graph

A **Graph** is a data structure that can be used to represent data introducing also information regarding how the data are linked to each other.
It is a diagram composed by 2 sets of elements:

- **Vertices** (Nodes): the set that represents the elements of our Graph (customers, companies, ...).
- **Edges** (Arcs): the set composed by pairs (v_i, v_j) representing a link between the 2 nodes v_i and v_j.

The flexibility and versatility of graphs make them an essential tool for representing and analyzing complex relationships and dependencies within

datasets. Additionally, graph theory, the mathematical study of graphs, provides a framework for understanding and solving problems related to connectivity, paths, and network analysis, making them a fundamental concept in the field of data management and analysis.

They are especially valuable in areas like social network analysis, recommendation systems, and computer networks, or in any use case in which the knowledge of the connections among the data plays a key role in understanding the data behavior, such as in Financial environment.

2.1.1 Types of Graphs

Several types of graphs are available in the context of data structure depending on which information they add on top of the basic Graph structure. We can highlights 2 types of Graphs that respectively add information about the **direction** of the arcs or the **weight** of an arc *[Figure 2.1]*.

Regarding the directions of the arcs, we can separate:

- **Directed Graph**: the Edges have directions, thus if Node v_i is linked with node v_j this does not imply that Node v_j is linked with node v_i. *(Edges are graphically represented with Arrows)*
- **Undirected Graph**: the Edges have no directions, a link between two nodes can be interpreted in both ways. *(Edges are graphically represented with Lines)*

Regarding the weights of the arcs we can separate:

- **Weighted Graph**: each Edge between nodes has a number, called Weight, $w_i \in \mathbb{N}$. This value can represent features like Costs, Lengths, Time... *(w_i can also be a negative number)*
- **Unweighted Graph**: the Edges have no numbers on them. It can also be seen as a Weighted Graph where all w_i have a value of 1.

2.2 Limitations of Relational Databases

A **relational database** is a database which arranges data into tables consisting of rows and columns, creating a structure where the data points are interconnected.

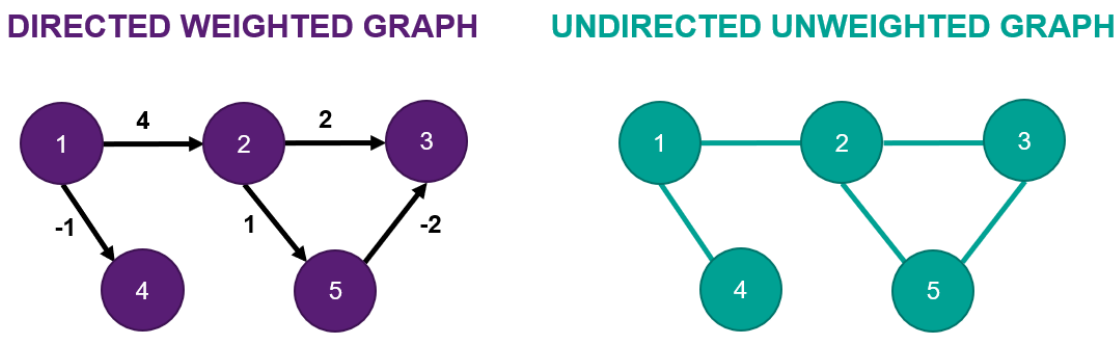

Figure 2.1: Types of graphs

Data is typically structured across multiple tables, which can be joined together via a *primary key* or a *foreign key*. These unique identifiers demonstrate the different relationships which exist between tables, and these relationships are usually illustrated through different types of data models.

Relational databases[3] are widely used in several types of companies due to their simplicity because simple SQL queries are sufficient for handling the data as there is no need for complex query processing or structuring. SQL also simplifies the retrieval of data from multiple tables and enables straightforward transformations like filtering and aggregation. Additionally, the use of indices in relational databases allows for swift information retrieval without having to search through each row in the chosen table.

The simplicity of relational databases is also the weak point[8] of these kind of databases, showing the limitations of this data storage systems.

As a relational database becomes wider and spread across multiple servers, its structure may become more complex and challenging to manage, particularly when dealing with large volumes of data. As the database grows in size the limitations lead to adverse effects such as latency and availability issues that impact overall performance.

Storing data exclusively in tabular form limits relational databases in representing intricate relationships between objects. This represents a challenge for applications that need multiple tables to store all the necessary data for their application logic.

Another limitation of relational databases comes from the simplicity of the SQL query language, which shows the difficulty in efficiently representing and querying complex, interconnected data. Relational databases are not as well-suited for managing highly connected data structures, such as social networks

or complex hierarchical relationships. Additionally, navigating and querying relationships between data in a relational database can be more cumbersome and less performant compared to the specialized querying capabilities of other types of database systems.

2.3 Improvements by using Graph Databases

A **Graph Database** is any storage system that uses graph structures with nodes and edges, to represent and store data. Graph databases offer a very useful feature in their native processing capabilities, specifically through a property known as **index-free adjacency**. This property ensures that each node is directly linked to its neighboring node. In a database engine utilizing index-free adjacency, each node maintains direct references to its adjacent nodes, effectively serving as an index of nearby nodes. This approach is more cost-effective than using global indexes and is well-suited for local graph queries, as it requires just one index lookup for the starting node, followed by traversing relationships through direct physical pointers. In contrast, in relational databases, achieving a similar outcome would likely involve joining multiple tables through foreign keys and potentially additional index lookups. Furthermore the query computation time in graph databases results to be faster due to the fact that queries are limited to a specific section of the graph. As a result, the execution time for each query is only related to the size of the traversed portion of the graph needed to fulfill that query, rather than the size of the entire graph. One of the most useful advantages of the index-free adjacency is the fast, constant-time relationship traversal in the graph database, despite the relational databases in which an index search is always required leading to $O(log(n))$ operational times which depend on the size of the data. The larger the data size, the longer the computational time. *[Figure 2.2]*

Thus the use of graph databases and their advantage of constant relationship traversal time will lead to an improvement in terms of scalability and performances of the database.

In the financial field, where complex and interconnected data relationships are prevalent, graph databases emerge as the optimal choice for managing and analyzing data. Financial transactions, customer relationships, and market data are inherently interconnected, making it essential to represent and

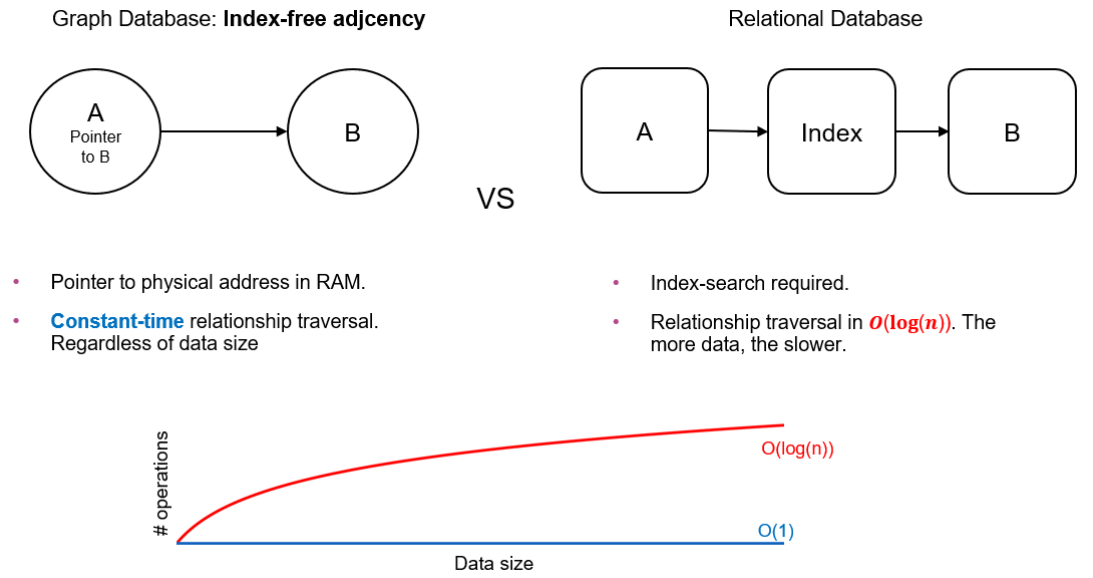

Figure 2.2: Graph Databases vs. Relational Databases

query these intricate relationships effectively.

Graph databases excel in capturing and navigating such complex, interconnected data structures, enabling efficient analysis of transaction patterns, fraud detection, risk assessment, and customer behavior analysis.

The ability of graph databases to handle interconnected data with ease, coupled with their native processing capabilities and index-free adjacency, allows for swift and targeted queries that are well-suited for the dynamic and interrelated nature of financial data.

Additionally, as financial data grows in scale and complexity, the ability of providing localized queries, resulting in proportional execution times based on the traversed portion of the graph, further solidifies their suitability for the financial domain.

2.3.1 What can we do with Graphs in the financial environment?

The uses of the Graphs to analyze and manage financial data can be multiple, here there are some example of their possible applications:

- **Knowledge Graphs**: to understand what is important in the graph.
  - Graph visualizations for analysts to **explore**.

  - **Analyze** a customer's complete environment: capital, family and business links, etc.
  - **Investigating** a mixed fraud network.

- **Graph Algorithms**: to understand what is atypical in the behavior of nodes.
  - Identify **anomalies** and clusters by mapping business rules to algorithms or using existing ones.
  - **Centrality**: determining the key individuals in a network.
  - **Similarity**: find individuals similar to a known fraudster.
  - **Path Finding**: establish links between entities.

- **Graph Native Machine Learning**: to predict what is going to happen in terms of nodes and relationships.
  - **Enhance models** with indicators derived from graphs (e.g. new beneficiary fraud risk model).
  - **Node classification**: training a model to recognize a potential terrorist financer based on its properties and relationships.
  - **Relationship prediction**: model that estimates the probability of a link between unconnected individuals.

Chapter 3

Neo4j

This chapter will put the focus on explaining the tool used to perform all the operations and analysis that will further be described in the chapter 5, "Algorithms Applications".

I will quickly describe the basic notions of Neo4j, the data structure used by the program analyzing its components, its query language which powerful and easy to understand, finishing with the data science library which has been used to compute scores for the analysis further described.

3.1 What is Neo4j

Neo4j is a prominent graph database tool that has revolutionized the way data is stored and queried in the field of computer science. Initially developed by Neo4j, Inc., this graph database has gained widespread recognition for its innovative approach to managing and analyzing complex relationships within data.

The roots of Neo4j can be traced back to its inception in 2007, when it emerged as one of the pioneering graph databases in the market. Since then, Neo4j has evolved into a robust and versatile tool, offering a range of features and functionalities that cater to diverse application scenarios.

One of the key features of Neo4j is its native graph storage and processing, which enables the representation of data in the form of nodes and relationships. This approach aligns with the inherent nature of connected data, allowing for efficient traversal and querying of complex relationships. Moreover, Neo4j's query language **Cypher**, provides a powerful and intuitive means to interact with the database, facilitating the seamless retrieval and manipulation of interconnected data.

In addition to its core capabilities, Neo4j boasts a rich ecosystem of tools and libraries that support various aspects of graph data management, analysis, and visualization. These include integrations with popular programming languages, comprehensive graph algorithms, visualization tools and **Graph Data Science *(GDS)* libraries** that includes algorithms to compute scores and show nodes and relationships behaviors depending on data topology and distribution as well as on pure data such as transactions, client information, etc.

3.2 Property Graph

Neo4j relies on the use of a particular type of graph called **Property Graph**, which blends together elements proper of the Graph data structure with information proper of Databases.

In this type of graphs we can identify 5 main elements.[4]

3.2.1 Nodes

Nodes *[Figure 3.1]* are the elements of the graph often used to represent entities, the elements of the databases which are interconnected to each other through **relationships**.

Nodes can also be described by **properties** and can also be labeled with zero or more **labels**.

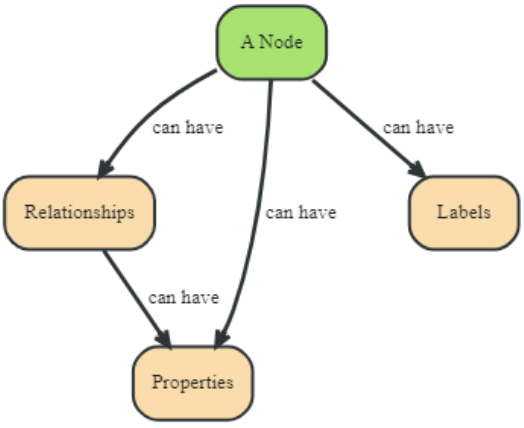

Figure 3.1: Property Graph Nodes

3.2.2 Relationships

In property graphs **Relationships** *[Figure 3.2]* are used to describe connections between Nodes.

They connect a *Source* node to a *End* node (a node can have relationships to itself as well) and since relationships are always directed, they can be seen as outgoing or incoming in relation to a node, which is helpful when navigating the graph.

Just like nodes, relationships can have **properties**.

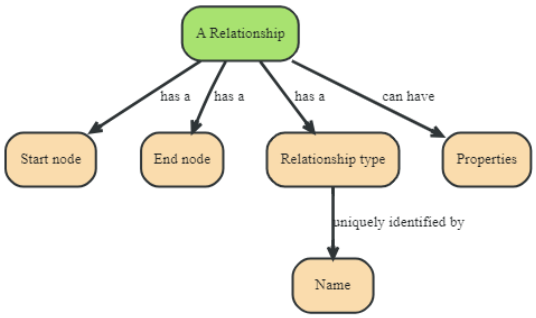

Figure 3.2: Property Graph Relationships

3.2.3 Properties

Properties *[Figure 3.3]* are key-value pairs to describe Nodes or Relationships.

Property values may be either primitive or arrays consisting of several elements of a single primitive type.

Properties of a node can be seen as the information stored in a tuple of a relational database (e.g. the information of a customer).

3.2.4 Labels

A **Label** *[Figure 3.4]* is a designated element in a graph that organizes nodes into groups, which means that nodes sharing the same label are part of the same group.

Knowing which node belongs to which group, we can enhance queries performances by adding constraints and enabling them to operate on specific groups rather than the entire graph.

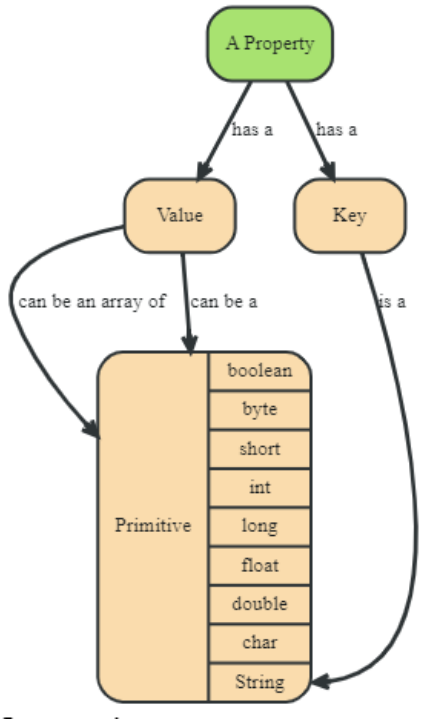

Figure 3.3: Property Graph Properties

Nodes can be assigned multiple labels or none at all, providing the option to include labels in the graph.

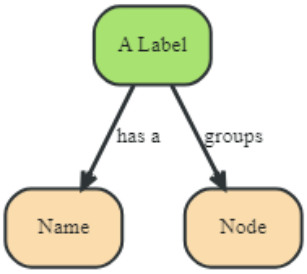

Figure 3.4: Property Graph Labels

3.2.5 Types

In property graphs **Types** can be seen as the equivalent of the Labels but related to Relationships.

Differently from the previous ones they are *compulsory* and *exclusive*, which means that each relationship must have one and only one Type in order to be crated and stored into the graph

3.2.6 Elements Combination

Combining together the elements of the graph we can identify **Paths** which connect a node defined as START to another node defined as END through one or more relationships.

Since relationships by definition are always **directed**, we can specify during the process if we want to considerate only the nodes that follow the direction of the relationships as belonging to the path, or we can indicate the presence of a relationship as simply a connection between 2 nodes, keeping in the path also the relationships that go in the opposite direction from the one followed up to now.

Traversing a graph involves moving through its nodes, adhering to certain rules as you follow the connections between them. Typically, you only need to visit a portion of the graph, as you are already aware of the locations of the nodes and relationships that are of interest to you.

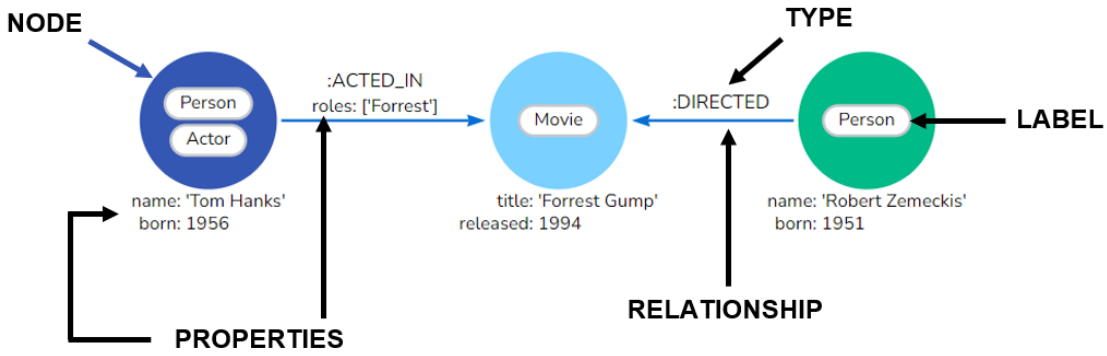

Figure 3.5: Property Graph Summary

3.3 Cypher Language

Cypher is the query language used by Neo4j on the property graphs.

It is a declarative language designed and developed with the same declarative capabilities of SQL but used onto a graph database structure, providing functionalities for both querying and modifying data.

Similar to SQL, Cypher uses a query method called *Linear*[5]. This method is based on a process in which the user can write the query in which the operations are executed one after the other in a step-by-step manner, usually from the start to the end. Each section of the query is processed in the sequence it is written, and the result of one operation becomes the input for the following one.

A Cypher query accepts a property graph as its input and generates a table as its output. These tables can be seen as offering associations for parameters that reveal certain patterns in a graph, with further processing applied to them.

Cypher compared to SQL offers queries which are often much more concise and natural than their SQL equivalents when it comes to relationships between entities, leading to a more **compact** and easy-to-read schema. *[Figure 3.6]*

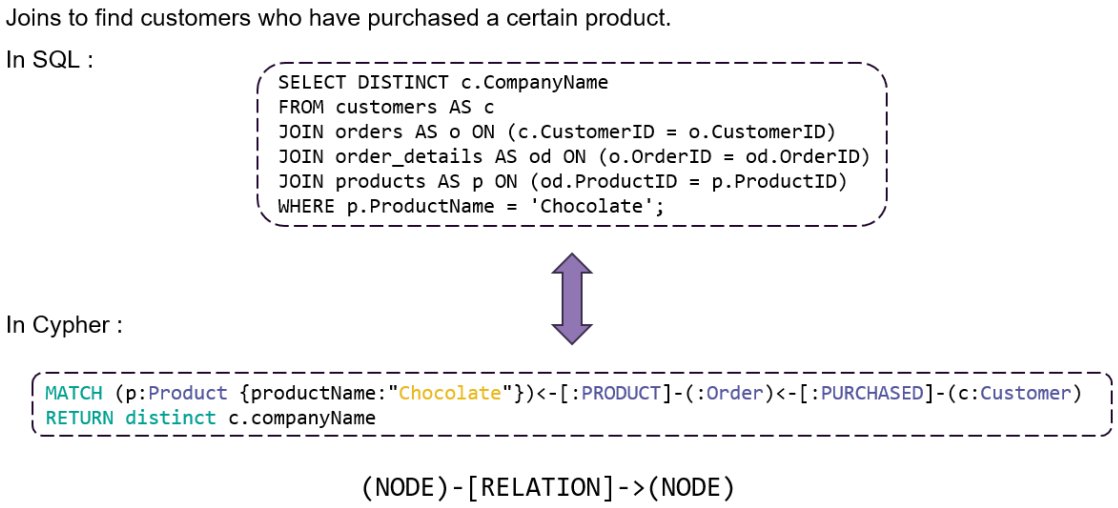

Figure 3.6: Differences between Cypher and SQL
(Example: finding customers who have purchased a certain product)

Since Cypher is a **Declarative Language** in which we define what to do and not how it has been done, there should be a **pipeline** to manage the query planning[6].

Initially, the Cypher query is processed by the *Cypher Parser*, which then creates a plan for executing the query using the *Cypher Query Planner*.

This plan determines which operators will be used to complete the query and the final result is returned to the user by the *Cypher Runtime*. The *Cost Estimator* also selects the most affordable plan from the list of potential execution plans.

The full pipeline is summarized in the following *Figure 3.7*

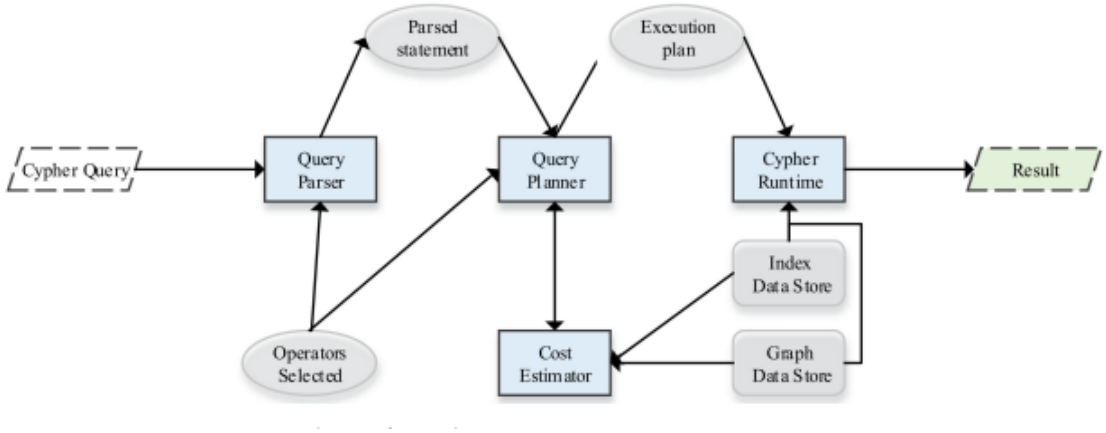

Figure 3.7: Cypher Query Pipeline

3.4 Graph Data Science (GDS)

Neo4j offers to its users a **Graph Data Science** (GDS)[7] library containing algorithms and Cypher procedures to compute scores and perform more accurate analysis on the datasets.

The algorithm and functions provided are used to calculate measurements for graphs, nodes, or connections. They can offer understanding into important elements within the graph, such as centrality and rankings, as well as the inherent structures like communities.

A lot of graph algorithms involve iterative methods that often navigate the graph for calculations using random walks, breadth-first or depth-first searches, or pattern matching. Because of the exponential increase in potential paths as the distance grows, many of these approaches also have a high level of algorithmic complexity.

Each algorithm can be analyzed before being executed to have an output containing the complexity in terms of time and computational resources needed, to have an initial estimation of the complexity of the algorithm.

The GDS library is continuously updated by researchers and programmers,thus we can find algorithms and procedures in 3 different states:

- ***Production-quality***: the algorithm has been evaluated for stability and scalability.
- ***Beta***: the algorithm is being considered for the production-quality tier.
- ***Alpha***: the algorithm is still being updated and developed, it can be changed or deleted at any time

3.4.1 Graph Algorithms

In this section of the library we can find a lot of different algorithms which implements procedures which use the key functionalities of graphs to perform analysis.

Here we can find algorithms that provides information for:

- ***Centrality***: it measures the relative importance of nodes within a graph.
- ***Community Detection***: it identifies groups of nodes with dense connections.
- ***Similarity***: it assesses the likeness between nodes.
- ***Path Finding***: it determines the shortest or optimal routes between nodes
- ***Directed Acyclic Graph Algorithms***: they handle graphs with directed edges and no cycles.
- ***Node Embeddings***: it represent nodes as low-dimensional vectors.
- ***Topological Link Prediction***: it predicts the likelihood of future connections between nodes based on their network structure.

These algorithms will be the key part for the chapter 5 "Algorithms Applications", thus their explanations, uses and mathematical details will be further described in that chapter.

3.4.2 Bloom

Next to algorithms and procedures to compute scores, Neo4j offers a tool named **Bloom** which is an illustrative, codeless search-to-visualization design that offers an intuitive interface for users to visually explore and interact with their graph data, expanding or removing the relationships of the nodes and moving the points into the space.

In this tool each Node and Relationship can be personalized adding colors and sizes depending on the Labels and Types each Node/Relationship belongs to, making the data visualization much easier as shown in the Figure 3.8.

Bloom allows users to create and execute graph queries, view graph patterns, and analyze the connections and relationships within the data.

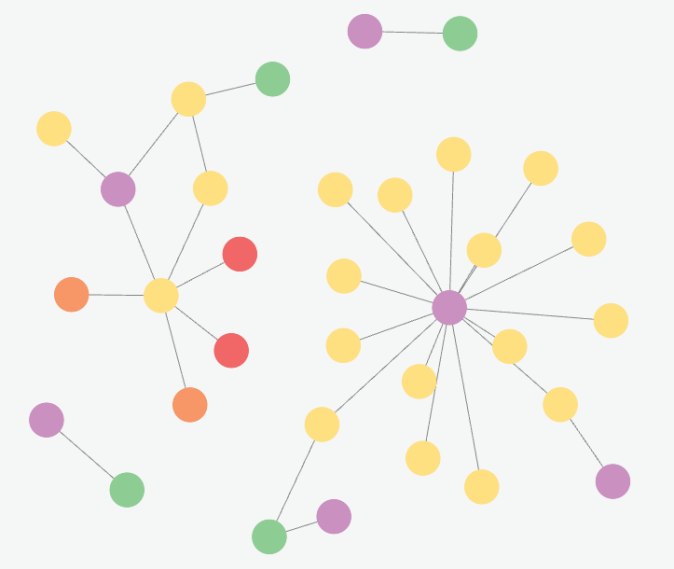

Figure 3.8: Neo4j Bloom Visualization

Bloom and GDS can work side to side to merge together the functionalities of both tools to graphically visualize the scores computed by some algorithm introduced in the section "Graph Algorithms" (3.4.1).

The sizes and the intensities of the color of a node will be proportional to the computed scores, making the analysis on the databases much faster to perform thanks to the extreme ease of finding the most important nodes through graphical visualization, as shown in the Figure 3.9.

Bloom works with the concept of **Scene**. A scene is everything that is showed in the working window, thus each analysis or GDS operation is related only on the current scene instead of the whole database.

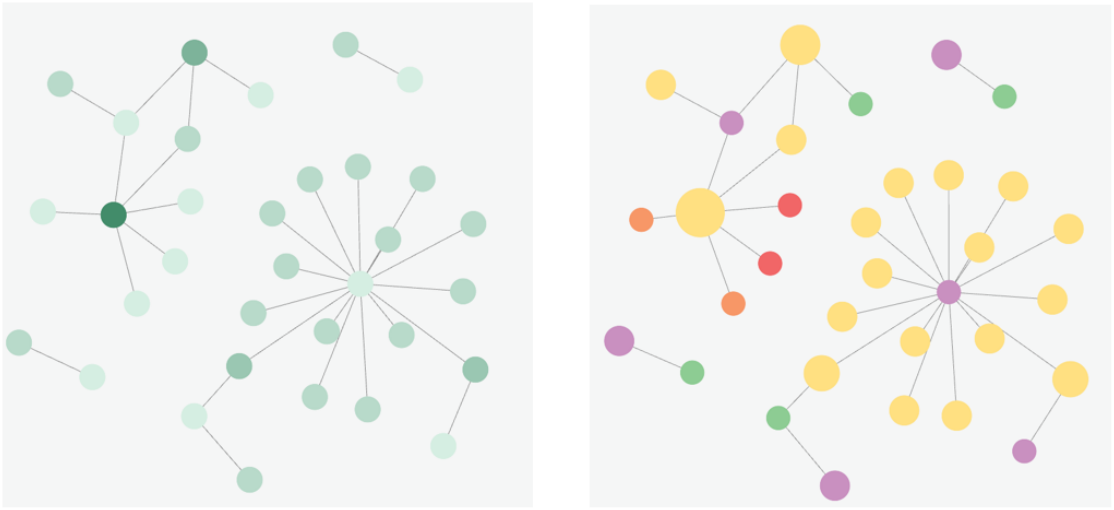

Figure 3.9: Neo4j GDS Bloom Visualization *(Color Shading/Size Scaling)*

3.4.3 Neo4j Browser

Neo4j **Browser** *[Figure 3.10]* is the tool parallel to Bloom. Designed for developers, it enables them to run Cypher queries and display the outcomes in a visual and numeric format.

It works as a classic declarative sheet, in which the user writes Cypher queries and gets as output a result based on the whole database, rather than Bloom which relies only on the portion of the Data showed in the Scene.

It is able to perform 3 main functions:

- Creating and executing **Graph Queries** using Cypher.
- Generating exportable, **tabular representations** of query outcomes.
- Visualizing **query results** that include nodes and relationships in the graph. The results can be seen as graph representations, tables or text format.
- Visualizing the detailed information about the **operations pipeline** performed during the execution of the query *[Figure 3.11]*.

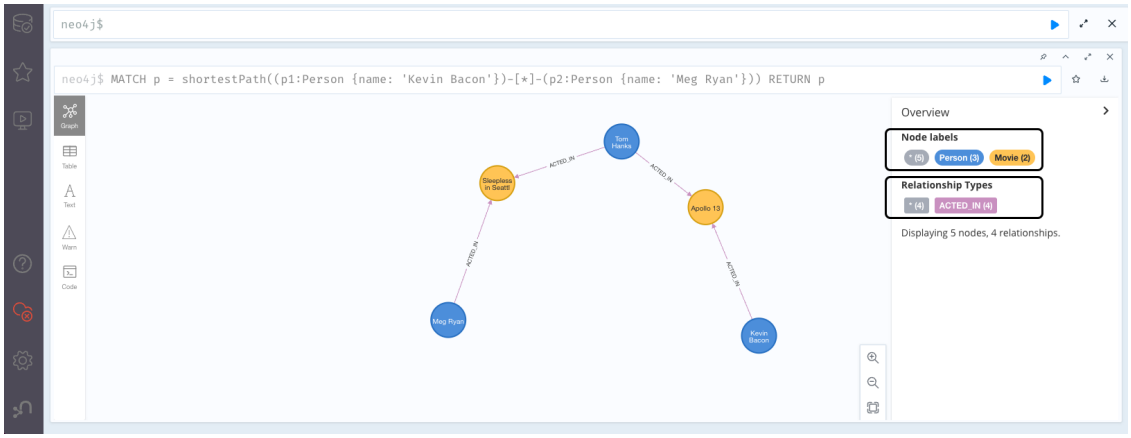

Figure 3.10: Neo4j Browser
(Example of a Shortest Path)

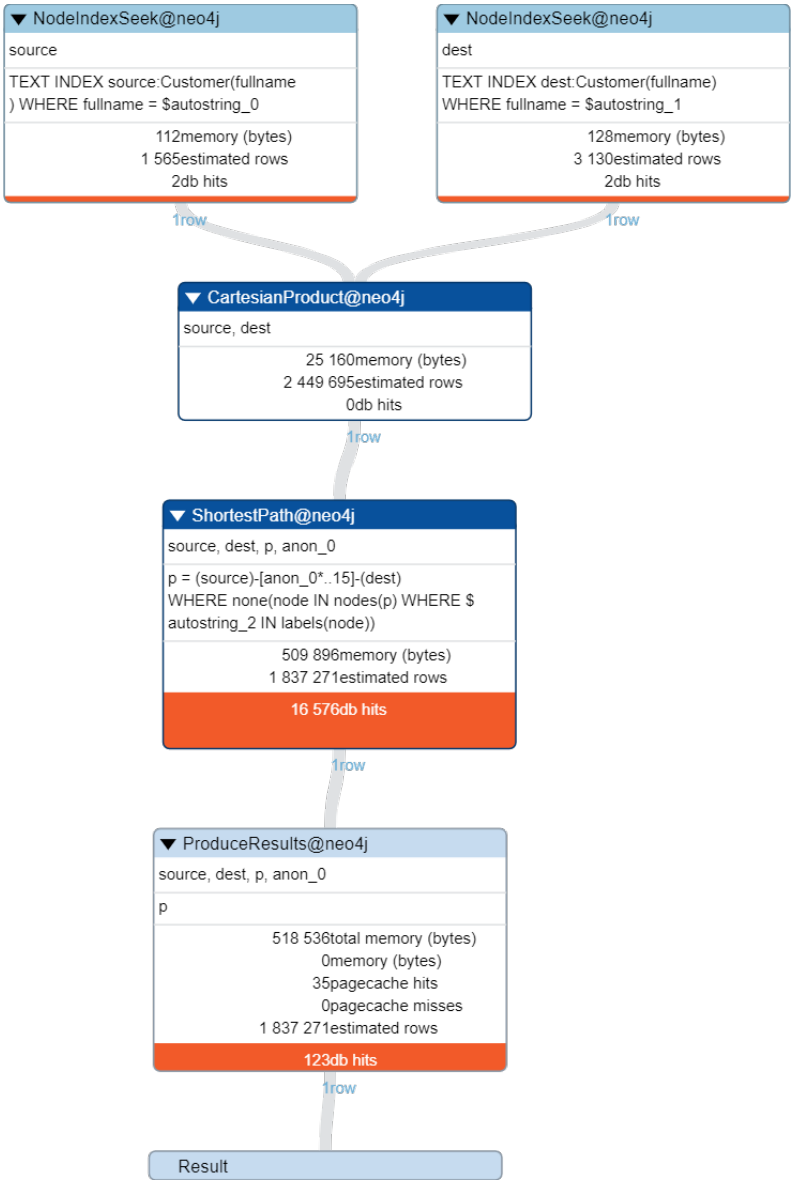

Figure 3.11: Neo4j Browser
(Pipeline of a Shortest Path)

Chapter 4

Used Dataset

This chapter will introduce the basic knowledge to understand the database that will be used to perform all the analysis in the chapter 5 "Algorithms Applications".

It starts with a short introduction of ICIJ, the consortium that provided the starting database used by the company as foundations to build its own analysis database on top of it.

I will talk about the Nodes and Relationships that compose the database, listing Labels, Properties and amount of samples for each type, explaining the behaviors and the role of each element of the Graph.

4.1 ICIJ and its Dataset

The **International Consortium of Investigative Journalists** or **ICIJ**[9] is a worldwide organization of 290 investigative journalists and 140 media outlets across more than 100 countries headquartered in Washington, D.C.

The goal of this consortium is to promote transparency and accountability through collaborative investigative reporting, as well as to provide support and resources to journalists worldwide. Additionally, the organization advocates for press freedom and the protection of investigative journalists.

The ICIJ makes available to everyone the data retrieved from them as downloadable Dataframes and one of the most important dataframe related to financial crimes is the ***ICIJ Offshore Leaks Database***[10].

This database reveals the hidden details of Companies and Trusts established in tax havens, showing the people associated with them, including the names of the real owners of these opaque structures when available.

Altogether, the interactive application reveals more than 750,000 names of people and companies linked to secret offshore entities.

This data comes from leaked documents rather than a standardized corporate registry, so there may be duplicates, even within the same leak. In addition, some companies are listed as shareholders in another company or trust, a setup that often serves to hide the real people behind the offshore entities.

The database does not reveal large amounts of raw documents or personal data. It contains a substantial amount of information about company owners, attorneys, and intermediaries in secret jurisdictions, but does not reveal details such as bank accounts, e-mail exchanges, or financial transactions in the documents.

The ICIJ makes this information public in the interest of the public. Although many activities conducted through offshore entities are legal, extensive reporting by the ICIJ and its media partners over the past eight years has shown that the anonymity provided by the offshore economy facilitates money laundering, tax evasion, fraud, and other illicit activities. Even when it is legal, transparency advocates argue that the use of an alternative and parallel economy undermines democracy by favoring a select few over the majority.

The Offshore Leaks Database holds data on over 810,000 offshore entities connected to investigations like the *Pandora Papers*, *Paradise Papers*, *Bahamas Leaks*, *Panama Papers*, and *Offshore Leaks*, linking individuals and businesses in over 200 countries and territories.

All these leaks are large-scale journalistic investigations that have exposed the use of offshore tax havens and the financial activities of wealthy individuals, corporations, and public figures in illicit activities such as tax evasion, money laundering, and others.

The leaks have been revealed in different years and are related to different areas of the world and people.

Briefly:

- ***Pandora Papers***: They are a massive financial leak of 11.9 million documents made public in October 2021 that revealed the offshore financial dealings on more than 330 world leaders, politicians, celebrities and business tycoons.

- ***Paradise Papers***: They refer to a leak of 13.4 million confidential electronic documents made public in November 2017 that exposed the offshore financial activities of numerous individuals and entities to avoid taxes and engage in complex financial arrangements.

- ***Bahamas Leaks***: They refer to a financial leak made public in September 2016 that exposed the offshore financial activities of 175,000 companies, trusts, and foundations registered in the Bahamas.

- ***Panama Papers***: They refer to a massive leak of financial documents from Panamanian law firm *Mossack Fonseca*, which revealed the offshore financial activities of numerous individuals and entities. The leak has been made public in April 2016.

- ***Offshore Leaks***: They refer to leaked documents, obtained by German newspaper *Süddeutsche Zeitung*, that uncovered numerous high-profile individuals and corporations around all the world.

The Offshore Leaks Database is available also as downloadable dump[11] directly insertable into **Neo4j**, on which we can perform all the wanted analysis using the relationships between companies, people and leaks.

The Structure of the Graph Offshore Leaks Database is the one showed in the figure 4.1.

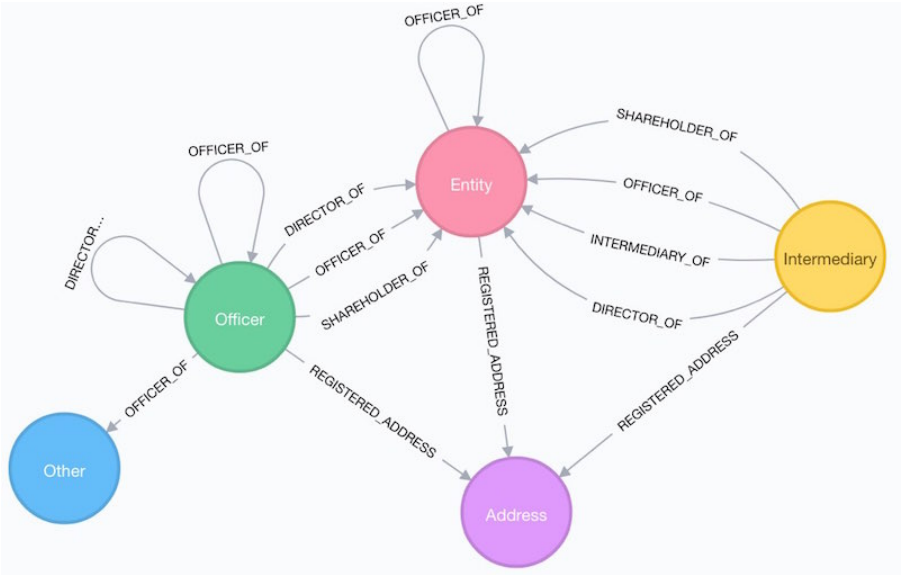

Figure 4.1: Graph Offshore Leaks Database

Officers are connected through various relationships (such as directors, shareholders, beneficiaries) to Entities (known as shell companies).

Intermediaries, such as banks and law firms, oversee the establishment and functioning of these shell companies. Furthermore, all these entities have addresses that are potential subjects for investigation.

Every node contains attributes for countries and their corresponding country codes, allowing for their association with particular geographic regions and a variety of other data.

4.2 Company Database

To pursue its analysis, the company used the ICIJ Offshore Leaks Database as base to build its own database from it.

Natixis decided to slightly modify the ICIJ databease making it lighter by reducing the number of different labels and relationships and adding on top of the new database, the company data of its clients.

In particular some Nodes found in the figure 4.1 have been suppressed like the "*Address*" one, while others like "*Officer*", "*Intermediary*" and "*Other*" have been transformed into relationships that link together a new type of node that generally describes an entity that has been called as "**ICIJ Entity**".

Furthermore the information about the Customer and the people that have links with them have been inserted into new types of nodes.

After all the modifications and removals, the Graph database associated to the company database, used to perform all the analysis described in the next chapter and has a structure showed by the property graph in Figure 4.2.

4.2.1 Nodes

The Natixis Database is composed of 4 types of Nodes, reducing by one the amount of nodes of the original database.

The nodes that compose the company Property Graph are the following ones:

- **ICIJ Entity**: This type of node represents any instance of any company or person that has been ever appeared in any leak described by ICIJ. (There could be several ICIJ Entities related to the same customer or person)

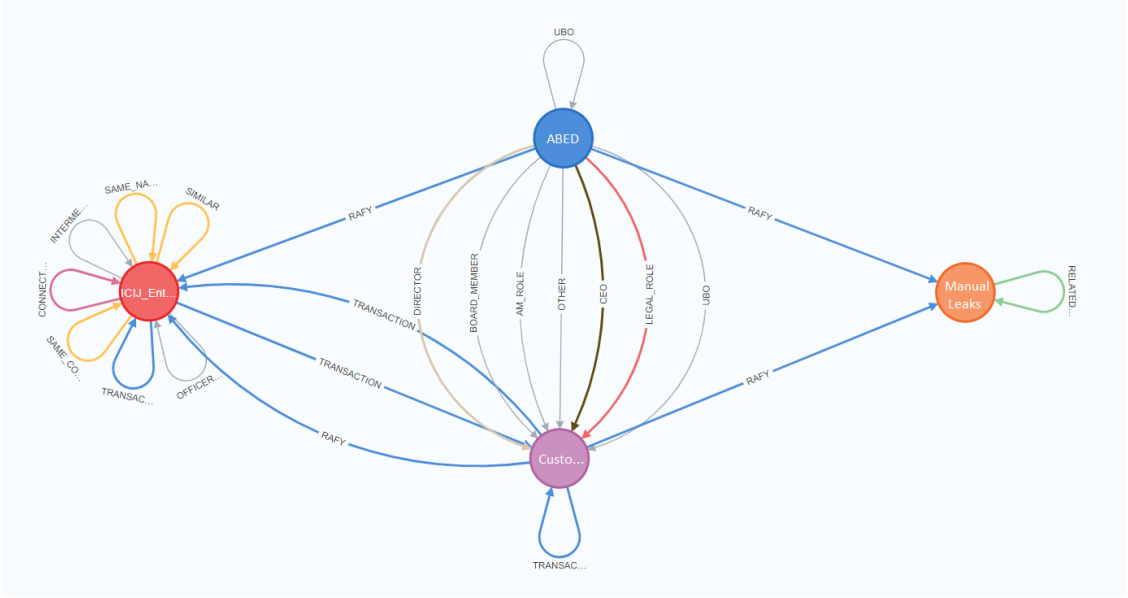

Figure 4.2: Modified Company Database

With the word "*instance*" we can refer at any physical or moral person that is related to a company, but we can also directly refer to the company itself or a subpart that composes it.

These nodes are described by a lot of properties that add information about the instance itself such as the *Name*, the *Address* and the *Countries* the instance is from. It is further described by attributes that add information about the Leak Papers, as ***sourceId*** property which gives the name of the Paper the entity has been derived from, *Valid Until* which gives information about eventual deadlines or the ***Status*** property that gives information about the state of the entity (Active, Inactive, Suspended, Removed, etc).

- **ABED**: It is a french acronym that stands for *Actionnaires Bénéficiaires Effectifs Dirigeants*, which means "Effective Shareholders Beneficiaries Officers".
  In other words these nodes represent any physical or moral person related to a Customer.

  These nodes are described by properties like *FirstName*, *LastName* and *Id* and also by a further attribute called ***Type*** that indicates if the ABED is and Individual or a Legal Entity.

- **Manual Leak**: These nodes represent manually selected Leaks from

press articles.
These nodes have been created because using the full amount of articles for the analysis would have been to costly in terms of resources and computational time.
In particular the Leaks manually inserted concern 2 main topics:

  - The "***Cyprus Confidential***"[12] which is a recent inquiry conducted by ICIJ and other media collaborators, revealing the extensive financial sector that has supported the Putin government, enabling it to exert influence over neighboring countries while also posing a threat to Western interests.
  - The "***ISF Gate***"[13] which refers to the alleged use of opaque Canadian trusts by wealthy French families to conceal assets and reduce their tax burden, particularly to evade the French wealth tax (ISF).

  Like ABEDs these nodes are described by attributes such as *FirstName*, *LastName*, *Id* and *Type* but also from an attribute called ***Role*** that describes the type of the physical or moral person the leak refers to. It can have values like "Artist", "Owner", "Investor", "Holding Company" and much more.

- **Customer**: These nodes represent the actual physical companies the ABEDs refers to.

  These nodes are described by 3 properties that describe its naming which are *FullName*, *Id* and *Geographical Location*.

  Then a 4^{th} property called ***Risk*** has been added, indicating the risk of the company with a value that can be Low, Medium or High.

4.2.2 Relationships

The Natixis Database is composed of 16 types of Relationships.

The relationships that compose the company Property Graph are the following ones:

- **UBO (Ultimate Beneficial Owner)**, **AM ROLE**, **BOARD MEMBER**, **CEO**, **DIRECTOR**, **LEGAL ROLE**, **OTHER**: All these types of relationships describe interactions coming from internal resources.
  It is mainly a group of relationships from an ABED to a Customer that explains which is the role of a particular ABED into the company represented by the Customer node.

(For example a relationship of type CEO means that the ABED is the Ceo of that Company).

Some details of these Relationships are further described by the property called "***detailed type***" that adds more information on the type of relationship between moral/physic person and the company.
(Possible values can be "Shareholder", "Executive Director", "President", "General Member", etc).

- **CONNECTED TO**, **INTERMEDIARY OF**, **OFFICER OF**, **SAME COMPANY**, **SAME NAME**, **SIMILAR**: All these Relationships link together only ICIJ Entity nodes.
  Relationships like CONNECTED TO, INTERMEDIARY OF, OFFICER OF, SAME COMPANY and SAME NAME are self-explanatory, describing the type of relationship that holds between the two linked nodes or if 2 entities share the same name; SIMILAR instead expands a bit the information given by the SAME NAME relationship, saying that 2 nodes have a similar Name and Address at the same time.

  All these relationships share a common property called "***Link***" that describes the reason why the nodes are linked together.
  (Possible values might be: "Related Entity", "Nominee Shareholder of", "Nominee Investment Advisor of", etc).

  Then CONNECTED TO, INTERMEDIARY OF, OFFICER OF and SAME COMPANY share between them other 2 common attributes called "***sourceID***" and "***valid until***" that respectively indicate the Leak Paper the relationship has been derived from and eventual deadlines or expiration dates that could be present.

- **RELATED TO**: This relationship connects a Manual Leak with another Manual Leak.

  It has no particular attributes, thus it simply explains a link between 2 Manual Leaks.

- **RAFY**: This particular relationship explains a name-matching link between a Customer and an ICIJ Entity.
  It is an internal relationship that links a Customer with its instances found in ICIJ leaks basing on an algorithm developed by the company to asses how similar two entities are basing on their names and other information.

For privacy reason the mechanism behind the algorithm will not be explained.

It has only one substantial attribute called "***Score***" that indicated the Similarity Score between the 2 nodes involved in the Relationship.
Higher scores indicates an high level of similarity between the entities.
(In our case only scores with a minimum value of 70/100 can be translated into a RAFY relationship).

- **TRANSACTIONS**: This Relationship indicates a money exchange from a Customer to an ICIJ Entity or vice versa.
  It is a Relationship that is currently being implemented by the company, thus so far it does not link too many entities, therefore this type of relationship has not been used in the analysis performed during the project.

  It contains 2 main properties called "***Currency***" and "***Amount***" that add information on the nature of the transaction between the nodes.

4.2.3 Data Distribution

The distribution of Nodes and Relationship in the dataset is not homogeneous. Certain types of Entities are present in much higher number than others.

All the numbers of samples of the different Nodes and Relationships present in the database are showed in the following tables 4.1 and 4.2:

Table 4.1: **List of Nodes of the Company Database**

Node Type	Number of Samples
ICIJ Entities	1.321.614
ABED	79.995
Customer	31.303
Manual Leaks	160

Table 4.2: **List of Relationships of the Company Database**

Relationship Type	**Number of Samples**
OFFICER OF	1.497.324
INTERMEDIARY OF	560.3565
SAME NAME	90.513
UBO	46.750
SIMILAR	44.986
DIRECTOR	27.135
RAFY	24.500
BOARD MEMBER	20.776
ENTITY IN	16.466
SAME COMPANY	15.725
CONNECTED TO	12.540
LEGAL ROLE	9.522
CEO	7.000
OFFICER IN	5.675
OTHER	5.220
INTERMEDIARY IN	937
TRANSACTION	315
AM ROLE	276
RELATED TO	109

Chapter 5

Algorithms Applications

This chapter is the actual explanation of all the analysis and computations performed during the project.

It winds its way through 4 different applications of Graph algorithms, explaining the theoretical fundamentals at the base of any used algorithm and how that algorithm has been applied on the Dataset provided by the company for analysis.

The used algorithms belong to 4 different groups of Graph Algorithms that present notions of Centrality, Community detection, Path Finding and Topological Link Prediction.

5.1 Centrality

In graph analysis, **Centrality**[14] plays a crucial role in identifying the key nodes of a graph, evaluating the importance of different nodes within the graph based on relationships and/or other nodes.

Centrality includes several metrics, each of which offers a unique perspective on the importance of a node and provides valuable analytical information about the graph and its nodes.

The concept of Centrality is a crucial information while performing analysis on graphs because the distribution of data, in particular Nodes and Relationships, in the majority of cases is uneven.

Thus having a way to asses the value of "importance" of a node in a graph that may contain thousands or millions of Nodes is an important way to indicate which node takes up critical position in the data distribution.

Nodes with high centrality scores are often associated with strong leadership, widespread popularity, or a huge reputation within the graph. When a node possesses greater centrality, it indicates its proximity to the network's core, potentially resulting in increased power, influence, and accessibility within the network.[15]

There are several metrics to asses Centrality scores within graphs. Here we will put our focus on 2 of them: the Degree Centrality and the Page Rank score.

5.1.1 Degree Centrality

We start by examining the most basic and well-known centrality metrics, the **Degree Centrality**.

According to Freeman's definition[16], degree centrality consists of counting the number of edges that are connected to a specific node.

The Degree Centrality of a Node ***i*** in a graph ***G:=(N, R)***, where ***N*** is the set of nodes and ***R*** is the set of Relationships is written with the following syntax:

$$C_D(i) = deg(i) \tag{5.1}$$

Since with Neo4j we work only with **directed** graphs we need to split the concept of degree into 2 parts called **Inner Degree** and **Outer Degree**, that relies on the direction of the Relationships.

- The **Inner Degree** consists of the amount of ***Incoming Relationships*** that ends into a node ***i***. The syntax is the following:

$$deg^-(i) \tag{5.2}$$

- The **Outer Degree** consists of the amount of ***Outgoing Relationships*** that starts from a node ***i***. The syntax is the following:

$$deg^+(i) \tag{5.3}$$

Using both of these degree values we can asses 2 different values of Centrality in our graph, the **Inner Centrality** and the **Outer Centrality**.

INNER CENTRALITY APPLICATION

Example 1:

As first example of application of Inner Centrality score on our database we decided to see the amount of relationships of types UBO, AM ROLE, BOARD MEMBER, CEO, DIRECTOR, LEGAL ROLE and OTHER that goes from ABED nodes into a Customer node.

The goal of this analysis relies on the concept that big companies which have a huge number or employees and people related to them should have as well a high number of ABEDs linked to them. Small companies instead should have a smaller amount of links with ABEDs due to its smaller amount of employees.

The goal is thus analyze the Graph Database finding anomalies related to this concept, in other words finding if there is the presence of small companies that are nonetheless linked with a high number of ABEDs.

As we can see from the Figure 5.1 the company (which actual name has been changed) showed is a small Canadian vinyl manufacture company. This company being a small producer of niche products should be related to a small number of ABEDs while during the analysis we saw that it is linked with 97 ABEDs as shown by its **innerDegree** score in the Table 5.1.

This is an abnormal behavior for this type of enterprise, thus this company can be chosen as a candidate to perform further in-depth study of its situation, looking for other abnormal situations or traces of illegal behaviors.

Table 5.1: **Inner Centrality example 1**

Company Name	**innerDegree Score**
Vinyl Company	97

Example 2:

As second example we changed topic and Nodes, focusing only on Manual Leaks and the RELATED TO Relationships between them.

The key concept is finding among the Manual Leaks the one which appears to be more important among the others, thus the one most often referred to by other leaks.

This can be important because if in the future we find Companies or ABEDs appeared into these types of Leaks we can directly indicate which

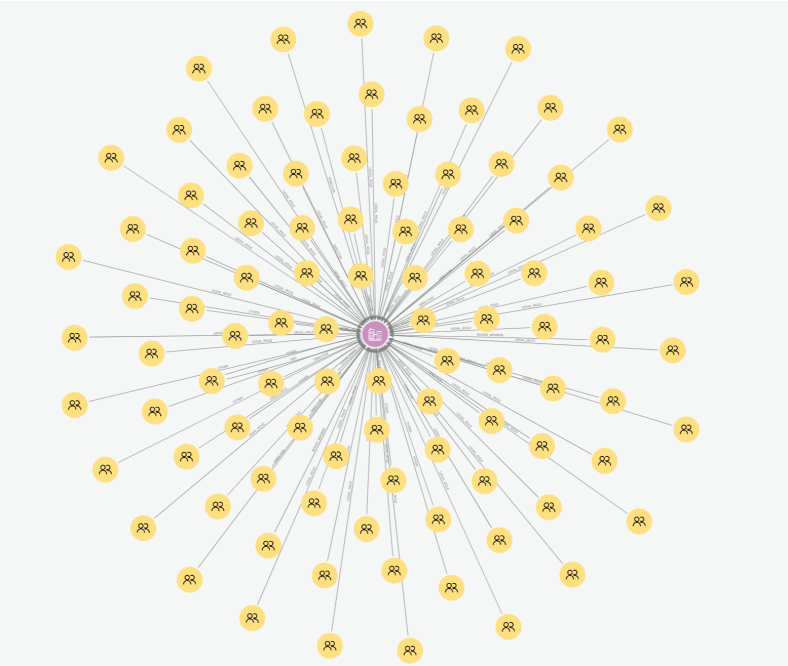

Figure 5.1: Inner Centrality example 1 *(Customer in pink, ABEDS in yellow)*

are the one to put our focus on in the first time, which will be the ones related to the "Most Important Leaks".

The example shows the **most related** Manual Leak and its results are shown in the Figure 5.2 and the score is stored in the Table 5.2.

Table 5.2: **Inner Centrality example 1**

Manual Leak	**innerDegree Score**
"Famille" Bontoux-Halley	11

OUTER CENTRALITY APPLICATION

Regarding the **Outer Centrality** we developed 2 examples in which we put the focus on the RAFY Relationship between Customers/ABEDs and ICIJ Entities.

The main concepts of these analysis are that when a Customer or an ABED is linked with ICIJ Entities through the RAFY relationship, it means

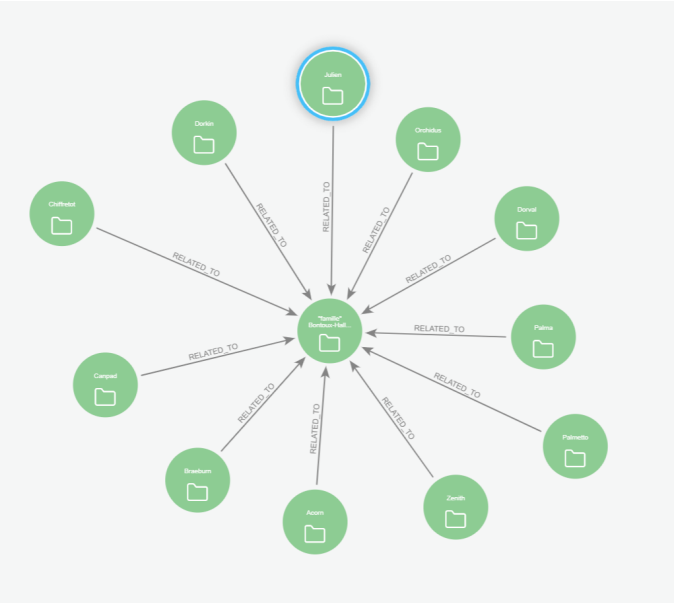

Figure 5.2: Inner Centrality example 2 *(Manual Leaks in green)*

that it has been found in that Leak *(because RAFY is a name-matching relationship with at least 70% of similarity)*.

Using the **outDegree** score, thus the number of outgoing RAFY relationships from Customer/ABED to ICIJ Entities we can find the Customers and the ABEDs that can be addressed as the *most suspicious* ones, because they have appeared the most in papers containing illegal activity records.

The results of the analysis are showed in the 2 Figures 5.3 and 5.4, while the scores are showed in the Table 5.3.
(Also in these examples the names of the most suspicious Company and the most suspicious ABED have been omitted for privacy reasons)

Table 5.3: **Outer Centrality examples**

Customer/ABED	**outerDegree Score**
Most Suspicious Customer	128
Most Suspicious ABED	96

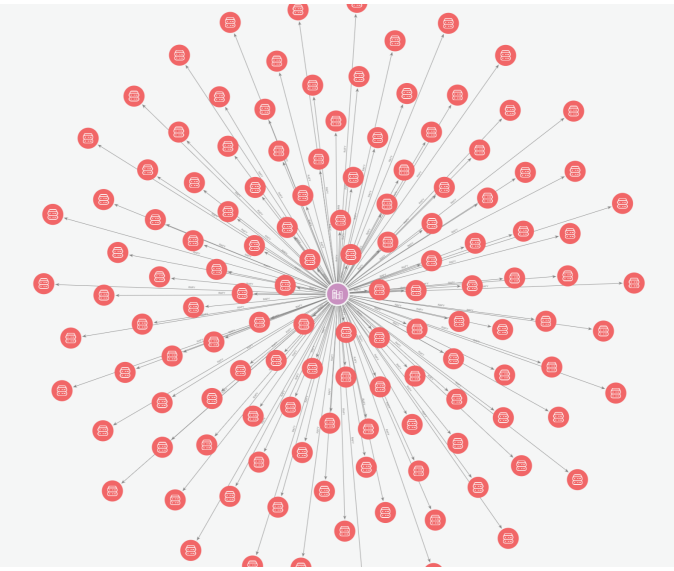

Figure 5.3: Outer Centrality example 1 *(Customer in pink, ICIJ Entities in red)*

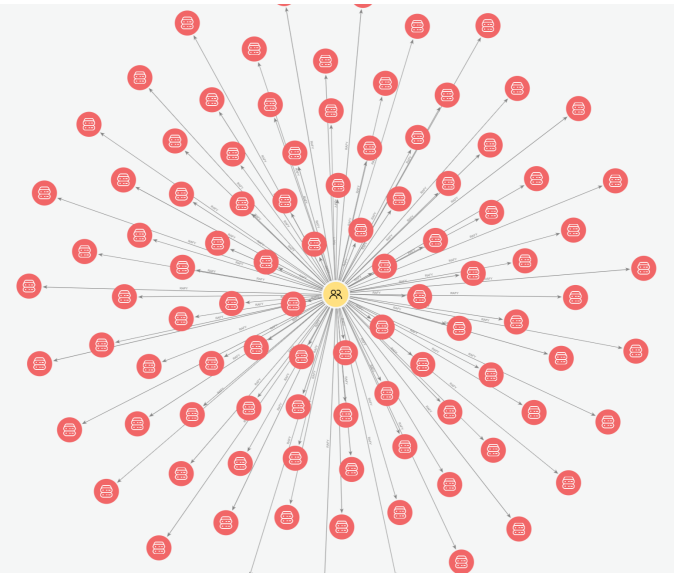

Figure 5.4: Outer Centrality example 2 *(ABED in yellow, ICIJ Entities in red)*

5.1.2 Page Rank

Another metric that is possible to use to asses the Centrality value of a Node in a graph is the **PageRank** algorithm.

PageRank[17] is an algorithm used to measure the importance of nodes within a graph, particularly in the context of web pages and hyperlinks.

The fundamental concept behind PageRank involves the introduction of a measure of node authority that is not based on the content of the node itself, but rather on the structure of the graph.

This authority is analogous to the concept of citations in scholarly literature. Specifically, a nodes's authority is determined by the number of incoming links (similar to citations) and the authority of the linking node. Additionally, selective and relevant citations are considered to have a greater impact on a page's score than uniform citations. As a result, PageRank calculates the score of a page by considering the set of pages that link to it.

In simple words the main idea behind the algorithm may be summarized as "***A node directly connected to an important node will be important as well***".

The algorithm is summarized[18] by the following formula:

$$PR(A) = (1-d) + d\left(\frac{PR(T_1)}{C(T_1)} + ... + \frac{PR(T_n)}{C(T_n)}\right) \tag{5.4}$$

In this formula:

- Let's assume that a node **A** is linked to by other nodes T1, ..., Tn.
- The **damping factor**, denoted as **d**, is a value that falls within the range of 0 (inclusive) to 1 (exclusive), with the commonly used setting being 0.85.
- **C(A)** represents the **outDegree** score of node A.

PAGERANK APPLICATION

The decision of using the PageRank instead of simple inDegree or outDegree may be crucial in those situations in which the score we want to get it's an actual score and not a simple count of relationships as it happens in Degree Centrality.

Another constraint that might us opt for this algorithm is the case in which we want that the importance of a node is defined also by the importance of the nodes it is surrounded with.

Furthermore the **Bloom** tool of Neo4j has PageRank in its GDS library,

thus we can also have a graphical representation of the various scores of our Graph.

An example is shown in the Figure 5.5 and in the Table 5.4.

In this example we used the same use-case showed in the example 1 of Inner Centrality. The printed score is the one of the Node that has the highest PageRank score, represented in the Figure with the red circle.

The different intensities of colors represent the different values of PageRank scores, where a darker color implies a highest score value.
(Here the Company whose name has been omitted for privacy reasons consists of an American Holding Company in the telecommunication sector).

Table 5.4: **PageRank Example**

Customer	**PageRank Score**
American Holding Company	9.24015

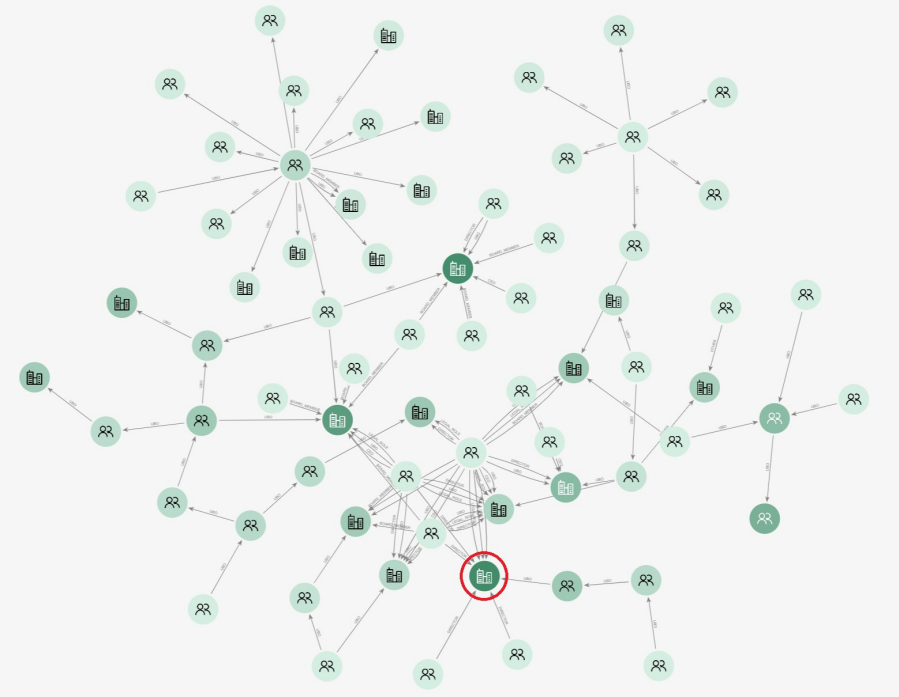

Figure 5.5: PageRank Example

EFFICIENCY COMPARISON

From all the performed analysis we can measure very fast execution times that goes from *90 ms* for the easy tasks like the Manual Leaks connections, to *298 ms* for the first example that links together around 100.000 Nodes with almost 100.000 Relationships, making the computation very time-efficient along with ease of visualization.

In general, a well-optimized relational database performs the links between the Nodes would need few seconds of processing, since the Relationships of the Graph would be replaced by Table Joins.

5.2 Community Detection

Identifying communities in graphs is an important and well-researched issue[19], involving finding closely connected groups of nodes and effectively distinguishing them from the rest.

Communities are prevalent and inherent in various real-world networks, such as social networks, collaboration networks, and web graphs, thus algorithms for **Community Detection**, also known as graph clustering, are essential for analyzing and comprehending large-scale networks.

Community detection techniques demonstrate how nodes are organized into clusters or communities, with nodes within each community sharing similar characteristics in terms of properties and/or topology. Additionally, it serves as an important tool for reducing the complexity of networks.

In our analysis we never use Node Properties as metric to compute our results, we only rely on the topology of the graph. Also in this chapter we will use only the information related to how Nodes are interconnected to each others through relationship to separate Nodes into communities using Weakly Connected Components and the Louvain algorithm.

5.2.1 Weakly Connected Components

Given a directed graph, a **Weakly Connected Component** or **WCC** is a sub-graph of the original one in which all the nodes that belong to that sub-graph are *weakly connected* to each other.

Two nodes $\underline{v}$ and $\underline{w}$ can be defined as being **weakly connected** to each other if, while considering all relationships of the graph as undirected, there

exists a path that connects the two nodes together.

Furthermore all the graph can be divided and subgrouped into several Weakly Connected Component, where there are no Relationships between nodes belonging to different components.

This allows us to build and define a "compact" version of the graph, in which each component is independent to the others, as showed in the Figure 5.6 where the full graph can be seen as the composition of 2 smaller graphs.

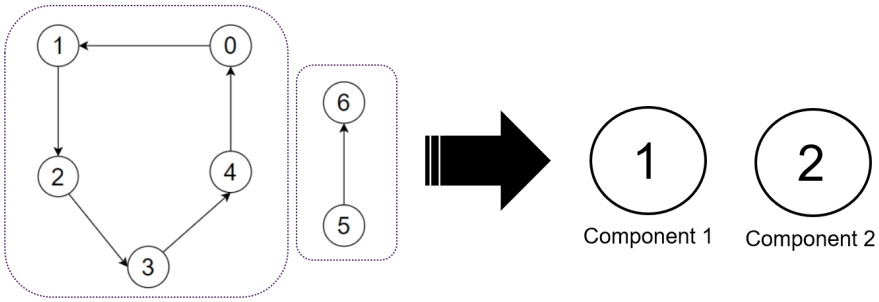

Figure 5.6: Graph Compression

WEAKLY CONNECTED COMPONENTS APPLICATION

In our analysis we decided to apply the WCC algorithm to the ICIJ Entity nodes, and on the relationships of types INTERMEDIARY OF, OFFICER OF, and SAME NAME that interconnect them.

We opted to use the algorithm on these type of nodes because they are the most numerous in the algorithm, with a proportion of 16 times bigger than the second more popular ones, thus they are the one that are more costly in terms of computing resources to perform analysis on.

If we want to perform an analysis on how the different ICIJ Entities are related to each other, for example to know starting from an Entity which are the other ones that could be even remotely connected or related to it for any reason, performing an algorithm would become easily infeasible in terms of time computing when we add more and more Leak Papers in the Dataset, that translates into millions and millions of nodes.

Applying the WCC separation on the Graph allows us to need to focus only on the sub-graph in which the starting entity comes from, knowing by the definition of WCC that there will not be any relationship that will link a sub-graph to another one.

In Neo4j Browser we can see for any ICIJ Entity the Id of the component it belongs to and the total amount of Components present in the graph, and in Bloom we can see the sub-partition of the graph as nodes of different colors, as shown in the Table 5.5 and in the Figure 5.7.
(Since it is impossible to show the all Graph in a single Bloom scene in the figure it is represented only a part of the whole Graph)

From the Table 5.5 we can see how the compacted Graph becames much easier to handle, counting only **57** components that can be seen as 57 **Supernodes**, each one containing Nodes and Relationships belonging to the component.

Table 5.5: **Weakly Connected Components Application**

ICIJ Entity	**ComponentId**
Morpho	0
Anymar De Talhouët de Boisorhand	3
Ulysse	3
Antoine De Rochechouart de Mortemart	5
Victoria	5
Victurnien	5
...	...
...	...
...	...
Total Number of Components	**57**

5.2.2 Louvain Algorithm

The **Louvain**[20] method, a multi-phase, iterative heuristic for optimizing Modularity presented by Blondel, et al in 2008.

This method is popular for its fast speed and ability to produce high-quality communities, making it one of the most commonly used tools for detecting communities in a series.

It is based on the concept of **Modularity**, a metric that can be seen as

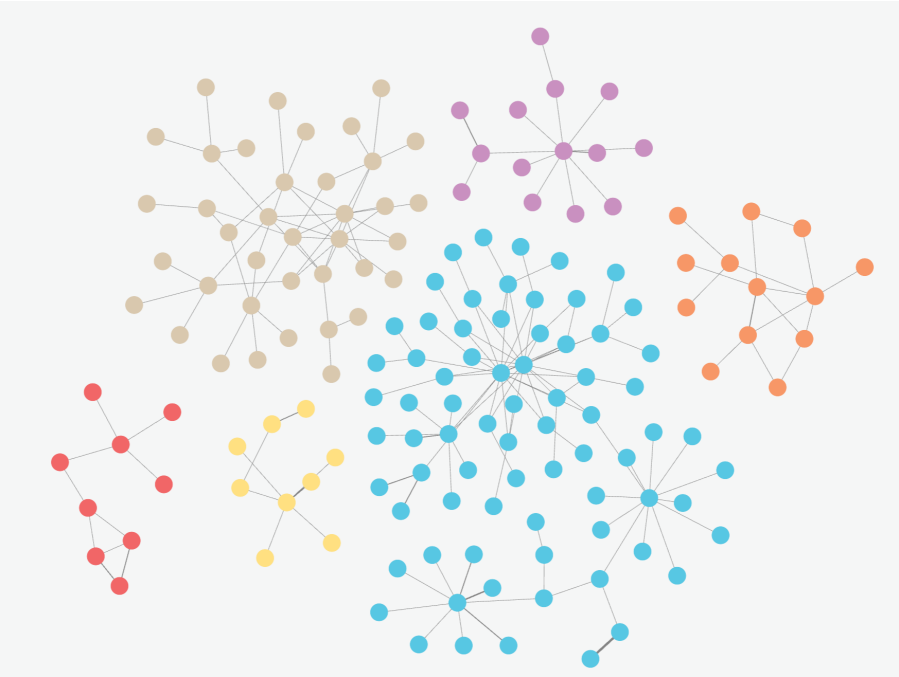

Figure 5.7: Weakly Connected Components Application

the one measuring the "strength" of the group division inside the graph.

it is a ***Greedy Algorithm*** which means that it aims to make the "locally optimal" choice at each stage, thus it may not produce an optimal solution, but it leads to locally optimal solutions in a reasonable amount of time.

The Modularity of a clustering $\boldsymbol{C}$ is expressed by the following formula:

$$Q(C) = \sum_k \frac{m_k}{m} - \sum_k \left(\frac{v_k}{v}\right)^2 \geq 0 \tag{5.5}$$

where $\boldsymbol{m}$ represents the total number of relationships in the Graph, $\boldsymbol{v}$ is equal of $\boldsymbol{2*m}$ and $\boldsymbol{k}$ is the number of different clusters in the Graph.

Louvain algorithm is utilized to maximize the Modularity for each community by employing a two-step recursive algorithm, which combines communities into a single node and performs Modularity clustering on the condensed graphs.

The algorithm mechanism is showed in the following pseudo-code and in the Figure 5.9:

LOUVAIN PSEUDO-CODE:

- **Phase 1**: Modularity is optimized by allowing only local changes to node-communities memberships;
- **Phase 2**: The identified communities are aggregated into Super-Nodes to build a new network;
- **Go To Phase 1**.

– The phases are repeated iteratively until no increase in modularity is possible.

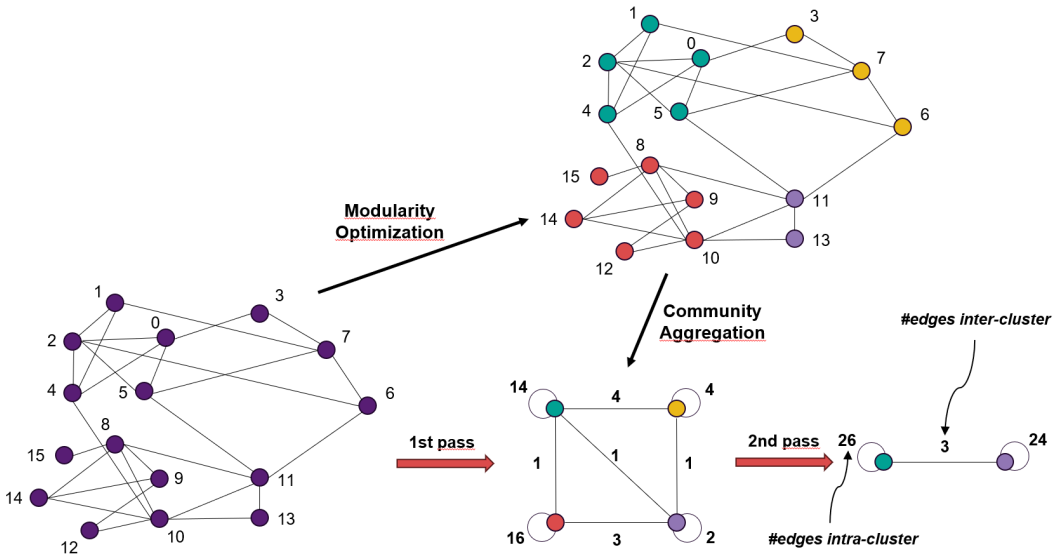

Figure 5.8: Louvain Algorithm

LOUVAIN APPLICATION

For our analysis we applied the Louvain algorithm in the use case of links between Customers and ABEDs, similar to what we did in the examples for Inner Centrality.

Louvain method is focused on finding **densely connected communities** within a network instead of focusing on finding disconnected components, helping to understand the structure and organization of complex networks.

Louvain gives us a "Lighter" division of our Nodes in the graph and using simple words we can say that the algorithm divides Nodes into partitions

in which "***each group can be seen as the set of nodes in which the majority of the information shared by the nodes is stored***".

We can see the lighter division strength of the Louvain algorithm by seeing our results, in particular we can see how the total number of components is now **28494**, instead of the 57 of the WCC example.

We can see it also graphically, where Nodes have been assigned to different groups even if there is one Relationship that connects them, that happens because the majority of information shared by the nodes is mainly kept among the nodes represented by the same color.

The results of the analysis are shown in the Table 5.6 and in the Figure 5.9 using the same format of the previous example on WCCs.
Also here the names of the Customers have been omitted The number used as Ids of the components are randomly selected in the Neo4j algorithm, that is the reason we have numbers greater than the total amount of components

Table 5.6: **Louvain Algorithm Application**

ICIJ Entity	**ComponentId**
Customer x	77922
Customer y	67584
Customer z	61467
Customer w	61467
Customer k	61467
...	...
...	...
...	...
Total Number of Components	**28494**

EFFICIENCY COMPARISON

Regarding the Community Detection algorithms' computational costs we can see how there is a huge difference and improvement due to the use of Graph Databases. Here to compute the WCCs on the entire database we need to take in account 1321614 Nodes, and the amount of time needed to compute

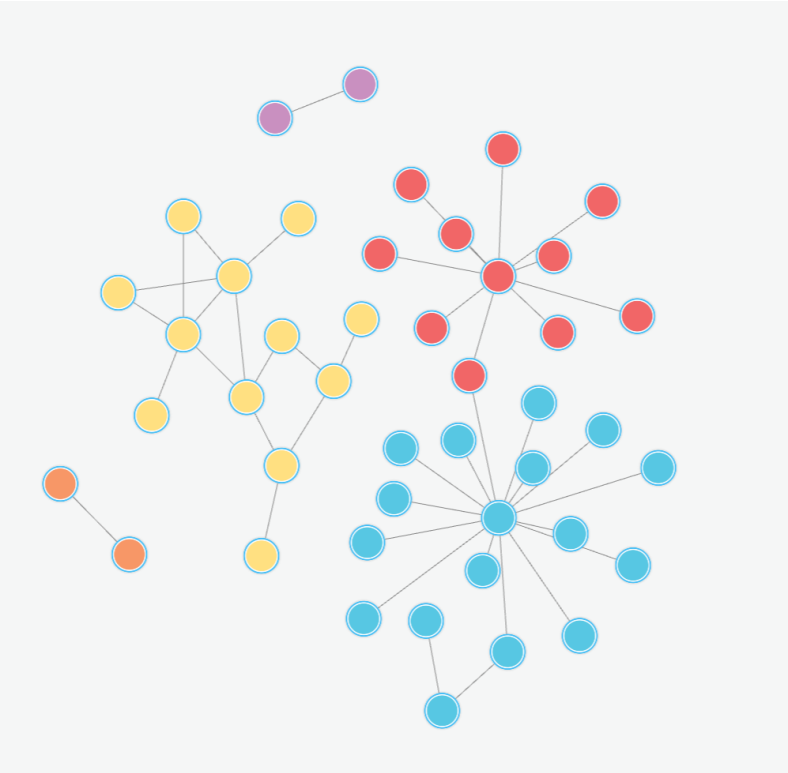

Figure 5.9: Louvain Algorithm Application

the WCCs is around *400 ms*.
If we wanted to perform the same process to a Relational database the expected computational time might go from several seconds to even minutes basing on the size of the Data. This because relational databases are not suitable to be used to perform these types of analysis.

5.3 Path Finding

Algorithms that belong to the **Path Finding** group are the ones in which the final goal is finding a Path from a ***Source Node*** to a ***Destination Node***, listing all the traversed nodes and relationships.

Fundamentally, a Path Finding algorithm navigates through a graph by beginning at a single node and investigating nearby nodes until it reaches the end point, usually with the goal of identifying the most cost-effective path.

Another goal could be the one of exploring the possible routes that can be travelled starting from a single node.

In our analysis we will perform 2 applications belonging to the first type of goal, called *Shortest Path* and *All Shortest Paths*, and one belonging to the second type of goal, called *Breadth-First-Search*.

5.3.1 Shortest Path

In the realm of graph theory, the **Shortest Path** problem involves determining the most efficient route between two points in a graph by minimizing the total cost of Relationships along the path.

Since in Neo4j we deal with *unweighted* relationships, the total cost can be easily defined as the "***number of traversed nodes***" through the path.

There are a lot of possible algorithms that can be used to compute the Shortest Path in a graph. In our case the ***shortestPath()*** function of Neo4j relies on the **Dijkstra's algorithm** to compute it.

Dijkstra's algorithm is the most commonly used algorithm to compute the Shortest Path.

It operates by keeping track of a queue of nodes sorted by their distance from the starting point, then iteratively chooses the node with the smallest known distance and adjusts the distances of neighboring nodes based on this selection.

This cycle continues until the shortest path is found for all nodes. A simple pseudo-code[21] is shown in the Figure 5.10.

SHORTEST PATH APPLICATION

For our application of the Shortest Path algorithm we put our focus in finding the path that links together 2 Customers.

An idea for our application might be finding out the minimum number of nodes that will be affected by the behavior of 2 Customers that have been detected as suspect, and then set as Source and Destination nodes.

From the result of the algorithm shown in the Figure 5.11 we can see how for this kind of analysis the **direction** of the Relationships that appear in the path does not matter. This because we are only interested in the presence or not of a link among 2 Nodes, not on the direction this link goes to.

From the Figure 5.11 that shows the result of the algorithm we can see that the minimum number of steps (traversed nodes) to go from the Source

```
1  function Dijkstra(Graph, source):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] ← INFINITY
5          prev[v] ← UNDEFINED
6          add v to Q
7      dist[source] ← 0
8
9      while Q is not empty:
10         u ← vertex in Q with min dist[u]
11         remove u from Q
12
13         for each neighbor v of u still in Q:
14             alt ← dist[u] + Graph.Edges(u, v)
15             if alt < dist[v]:
16                 dist[v] ← alt
17                 prev[v] ← u
18
19     return dist[], prev[]
```

Figure 5.10: Dijkstra's algorithm Pseudo-Code

node *(Customer 1)* to the Destination one *(Customer 2)* is **8**.

This is an information that we will use again in the next application. *(Here the names of the Customers (pink nodes) and the ABEDs (blue nodes) have been omitted, while the names of the ICIJ Entities are shown)*

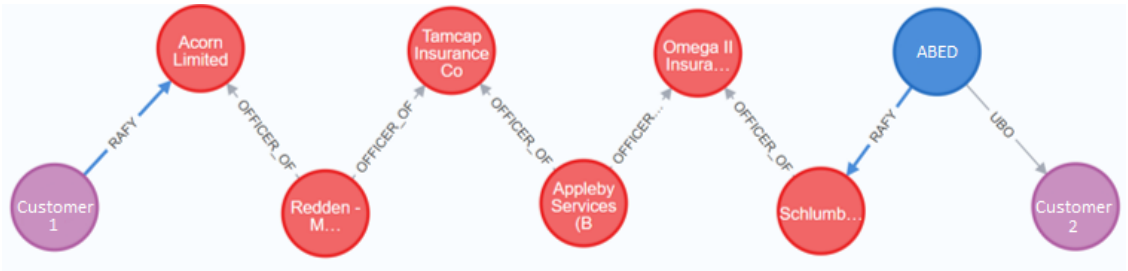

Figure 5.11: Shortest Path Application

5.3.2 All Shortest Paths

This analysis is strictly correlated to the previous one.

The result of this algorithm obtained from the ***allShortestPaths()*** function of Neo4j will show all the possible paths that go from the Source node to destination Node with a total cost that is the minimum possible.

ALL SHORTEST PATHS APPLICATION

The application of the All Shortest Paths algorithm in our case is just an extension of what we did in the previous application.

If we wanted to perform a wider analysis to see more Nodes that appear in the path between 2 potentially suspicious Customers, we can use All Shortest Paths to see all the paths of cost **8** that goes from the Source Customer to the Destination one, as shown in the Figure 5.12.

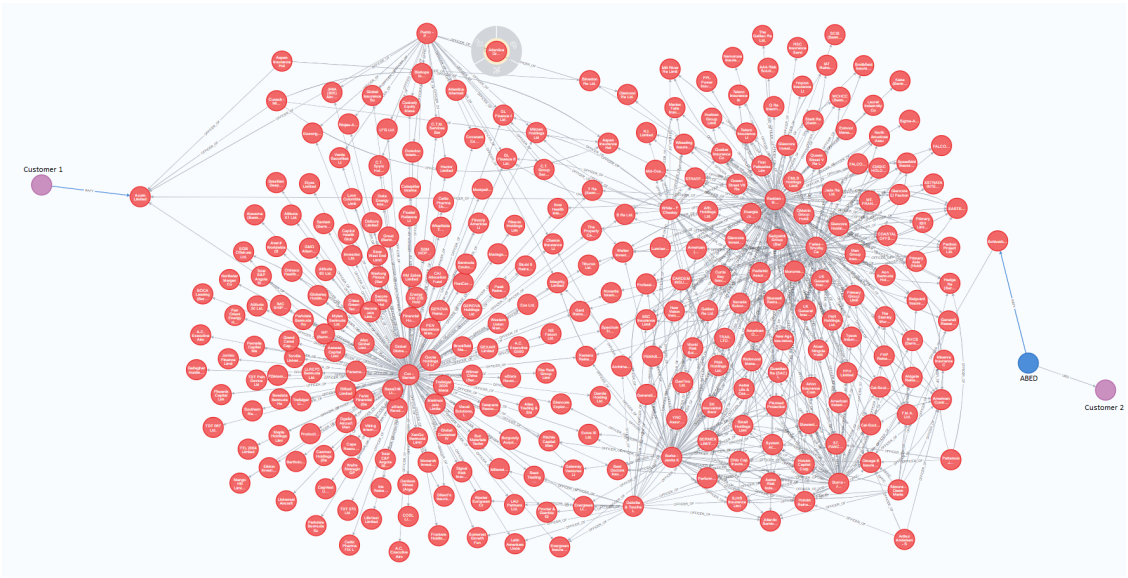

Figure 5.12: Shortest Path Application

5.3.3 Breadth-First-Search (BFS)

Breadth-first search (BFS)[22] is a basic algorithm used to explore a graph.

Given a graph $G = (N, R)$ with a designated starting node i, BFS systematically traverses the edges of the graph to find and "discover" all nodes that can be reached from i.

It calculates the shortest distance from i to each other reachable node. Additionally, BFS constructs a ***"Breadth-First Tree"*** rooted at i, which

includes all reachable vertices.

In BFS , progress is monitored by assigning each node one of three colors: *white*, *gray* or *black*.

Initially, all nodes are white, and those that cannot be reached by the initial node i remain white throughout the process. When a reachable node is encountered for the first time during the search, it turns gray, signifying that it is on the frontier, at the boundary between discovered and undiscovered nodes.

All gray nodes are stored in a queue. As the search continues, the edges of each gray node are explored and neighboring nodes are discovered.

Once all edges of a node have been explored, the node crosses the frontier and goes from gray to black.

The algorithm is also described by the Pseudo-code showed in Figure 5.13.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]   // search the neighbors of u
13          if v.color == WHITE         // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)           // v is now on the frontier
18      u.color = BLACK                 // u is now behind the frontier
```

Figure 5.13: Breadth First Search Pseudo-Code

BREADTH FIRST SEARCH APPLICATION

For our application of the Breadth First Search algorithm we decided to set as an initial Node an ABED, seeing all the other nodes that appear in the tree having the ABED as source.

This kind of application can be applied to any type of Entity but it gives more information when applied to entities such as ABEDs or Customers.

The key idea at the base of this analysis is seeing, given as source an ABED or a Customer that has been detected as *"suspect"*, all the other entities that can be ever be "touched" by its behavior, and thus the nodes that might have been influenced somehow by an illegal activity performed by the starting ABED/Customer.

That's why using BFS we have as result only the nodes that are directly or indirectly reachable from the node chosen as Source.

From the result showed in the Figure 5.14 we can see how in this type of analysis the direction of the Relationships matters, since all the relationships follow a outgoing direction starting from the source node (*SOURCE ABED*).

(Here the names of the Customer (pink node) and the ABED (blue node) have been omitted, while the names of the ICIJ Entities are shown)

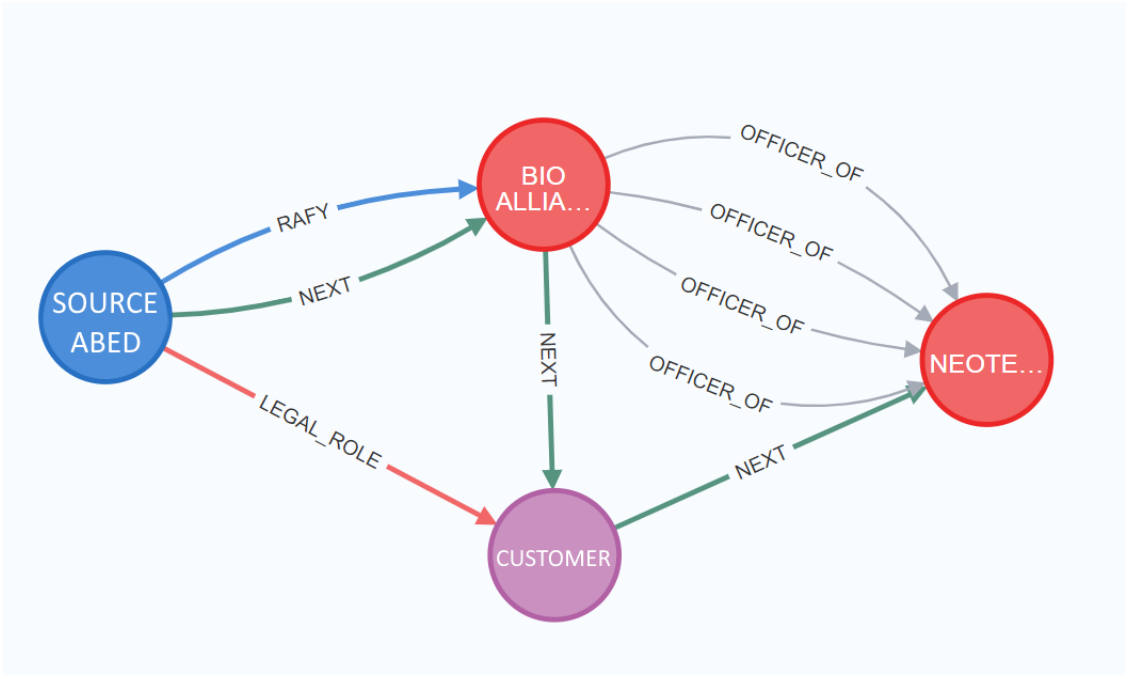

Figure 5.14: Breadth First Search Application

EFFICIENCY COMPARISON

Regarding the Path Finding algorithms an efficiency comparison between Relational and Graph database would be trivial. This happens because Path Finding is one of the key features of the Graphs, mainly relying on the Relationship concept.
Using a relational database to implement a path finding algorithm will lead to several difficulties to fulfill the need to model the relationships among the data. This may require proper database design and may require the use of complex joins and recursive queries to find paths, placing a significant load on database performances.

5.4 Topological Link Prediction

Topological Link Prediction is a fundamental problem in the field of graph theory and network analysis[23].

Its goal is to predict the probability of future or missing connections between nodes in a graph by exploiting the structural properties of the graph itself.

Thus the algorithms belonging to this family rely only on the structure of the Graph, called ***Topology***, consisting of the links between the several Nodes that compose the Graph.

For our analysis we will use two algorithms belonging to the Topological Link Prediction algorithm family, the *Common Neighbors* and the *Preferential Attachment*.

(Both of these algorithms are still on a ***Beta*** *state, thus they are still being developed by the Neo4j programmers).*

5.4.1 Common Neighbors

The **Common Neighbors** analysis is pretty self-explanatory. It consists of finding the Nodes that share at least one common Relationship among two Nodes chosen as Source.

In other words consists of the ***Intersection*** of the sets of *One-Step Relationships* that link the Source Nodes with other nodes in the Graph.
The idea behind this analysis can be seen also in the following Formula[24]:

$$CN(x, y) = |N(x) \cap N(y)| \tag{5.6}$$

where x and y are the 2 nodes chosen as ***Sources*** and *N(x)* and *N(y)* are respectively the *One-Step Relationships Set* of the x Node and the y Node.

This algorithm can give us a further information together with the knowledge of the neighbor nodes: when in a Graph two nodes share a lot of neighbors there could be a high probability that a direct link between the 2 nodes will be added in the future.

If we think the Graph as an example of social interaction between people the idea can be described in this form:
"Two strangers who have a friend in common are more likely to be introduced than those who don't have any friends in common".

The Algorithm provided by Neo4j will show in the Browser the nodes that have been detected as **Neighbors** and a **Score**, which represents the amount of neighbors detected by the algorithm.

COMMON NEIGHBORS APPLICATION

For our application of the Common Neighbors algorithm we assumed a use case in which we detected 2 potentially suspicious Customers that have been chosen as Sources *(showed in the Figure 5.15 with pink nodes)*, and we want to detect the ABEDs that have interacted to both these nodes.

This because ABEDs that belong at the same time to several Customers detected as suspicious might be the cause of illegal behaviors inside the company, or they can be related to these behaviors and thus being suspicious as well.

The results of this analysis are shown in the Figure 5.15 and in the Table 5.7.
(Here both Customers and ABEDs names have been obscured for privacy reasons)

Table 5.7: **Common Neighbor Application**

Common Neighbor Score
5

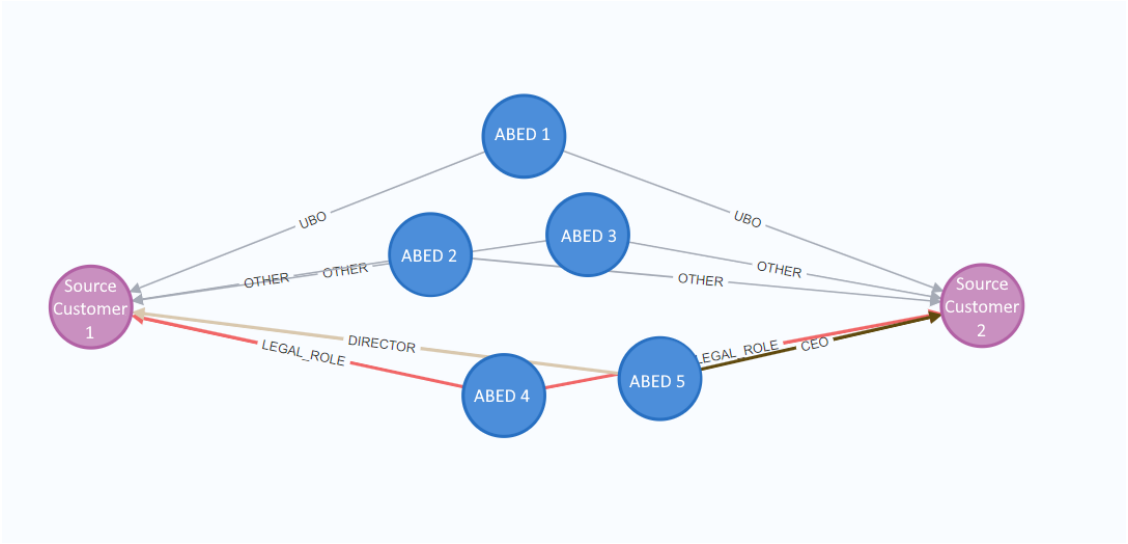

Figure 5.15: Common Neighbor Application

5.4.2 Preferential Attachment

Preferential Attachment[25] refers to the phenomenon in networks that expand over time, where the likelihood of a new edge connecting to a node with n neighbors is proportional to n.

This linear relationship is central to the Barabási-Albert model[26] whose base concept is:
"the more connections a node has, the higher its chances of attracting new links".

Thus, nodes with a higher degree are more likely to gain additional links as they are added to the network.

In Neo4j[27] the algorithm uses the **Cartesian Product** of the *One-Step Relationships Set* of the 2 Nodes chosen, as *Sources* as shown in the following Formula:

$$PA(x, y) = |N(x)| * |N(y)| \tag{5.7}$$

where x and y are the 2 nodes chosen as ***Sources*** and $N(x)$ and $N(y)$ are respectively the *One-Step Relationships Set* of the x Node and the y Node.

PREFERENTIAL ATTACHMENT APPLICATION

For our application of Preferential Attachment algorithm we chose as use case the one of links between Customers *(pink nodes in the Figure 5.16)* and all other Nodes *(yellow = ABEDs, green = manual leaks)*.

This application allows us to understand, given a pair of Nodes as input, how "high connected" they are inside the Graph.

This value has given as a **Score** based on the Cartesian Product of the 2 One-Step Relationships Sets of the source nodes, thus the score will be the product of the amounts of inner/outer relationships of each source node.

The higher the score, the more connected the chosen nodes are.

The results are showed in the Figure 5.16 and in the Table 5.8.
(Here Customers names have been omitted for privacy reasons)

Table 5.8: **Preferential Attachment Application**

Customer	**Number of inner/outer Relationships**
Source Customer A	15
Source Customer B	11
Preferential Attachment Score	$15 * 11 = 165$

EFFICIENCY COMPARISON

Also here the comparison between the two types of Graphs is trivial, since the concept of *"Neighbor"* is proper of a Graph.
The issues that will be encountered would be the same ones shown in the "Path Finding" section because it is necessary to model the Relationships from scratch as a new data type, greatly reducing the efficiency.

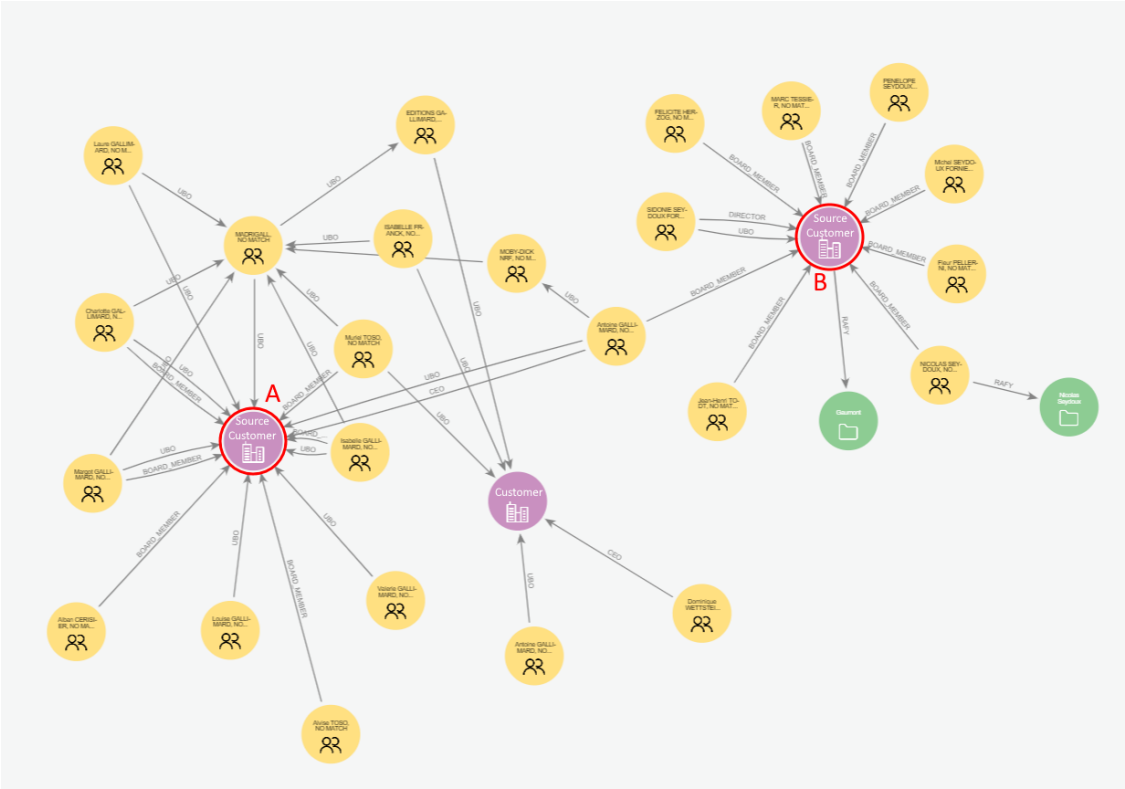

Figure 5.16: Preferential Attachment Application

Chapter 6

Further Implementations

The project discussed in this document is only the beginning of a wider study project that the company planned to continue in the next years.

The topics discussed in this document cover only the introduction of the potentiality that a Graph Database can offer in terms of analysis and research.

Here we focused only on the information that were proper of Graphs, such as *Nodes* and *Relationships*, without using information proper of Relational Databases that are build inside Neo4j as *Properties*.

Therefore the Database used for the analysis is just an initial form of what it has been planned to be. So far only few types of Nodes and Relationships have been declared and the Data Analysts of the company continuously work to enrich it in terms of new features and bigger sample sizes.

Regarding the used tool *Neo4j*, during our analysis we just focused on the use of the *Graph Data Science (GDS)* library or simple Cypher queries, but the tool has other libraries and features that can be used to improve and enrich our analysis.

6.1 Implementation of Transactional Data into the Database

The first further implementation that is currently in phase of development is the **implementation of Transactional Data** in the Graph Database.

With this we talk about all the relationships named TRANSACTION discussed in the section "Relationships" (4.2.2) that describe a money exchange

between two entities.

By introducing and developing this type of Relationship we have a direct connection between entities that is not a membership or relationship link like the ones discussed so far.

With the knowledge of the Transactions between entities, enriched with amounts, currencies and other information, we can describe the behavior of an entity just analyzing its habits and average quantities of exchanged money with other entities.

With this we can easily follow the money transfers starting from a Node that has been detected as suspicious from the analysts, or we can also directly help the analysts to graphically detect anomalous transaction by analyzing the TRANSACTION relationships of a Node in terms of *Degree* and *Amount*.

If for example a Customer has as habits to make an average number of transactions per month, if we see that the outer degree of the TRANSACTION relationship of that node, thus the number of money exchange to another Customer that client did in that month, is too different from its average, that could be a wake-up call to perform further analysis on that Client. A same example can be made taking in account the average Amount instead of the degree.

6.2 Uses and Enrichment of Node Properties

Another further implementation that can be done on this project in the one consisting of **Enrich the Properties of a Node** and furthermore use these properties as a metric for the analysis.

Currently the majority of the Properties that describe our entities are purely descriptive information of the personal data of Clients and ABEDs, or information on the Leak Paper a Manual Leak or a ICIJ Entity has been derived from.

By adding as properties metrics that has been computed with the algorithms discussed in the previous chapter we can add information of the actual behavior of that node in the Data population of the Graph.

With these kind of metrics and also adding other Properties that can further describe the behavior of a Node, we can asses the similarity of two Nodes by merging together information proper of its position in the graph, through

the amount and the type of Relationships with other Nodes (***Topology***), but also information proper of the Node itself.

By implementing these kind of improvements if the analysts detect a Node as suspect we can compute the similarity of other nodes to the suspicious one and if the score is high we can label other nodes as potentially suspects, as shown in the Figure 6.1.
(The red Node has been detected as "suspected", the orange Node is the "potential suspect" due to a Similarity of 0.8, the green Nodes have been exonerated due to their low Similarity)

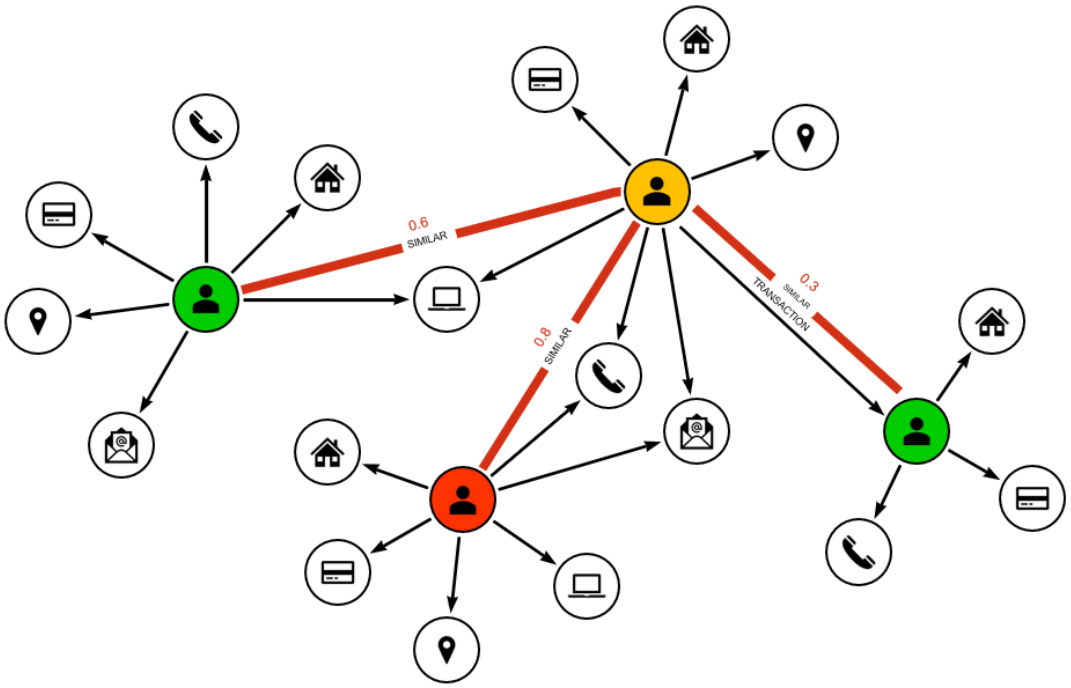

Figure 6.1: Suspicious Node Detection through Similarity

6.3 Neo4j Machine Learning Package

The last further implementation that might be added to this project is the one based on the further exploration and exploitation of the Neo4j GDS library, using the package dedicated to **Machine Learning**[28] *(currently in Beta/Alpha)*.

Enriching our Database with Properties that can be used as metrics to know the behavior of a node will be the basis to use elements of the Neo4j Machine Learning Package.

Elements such as *Node Classification/Regression* pipelines allow us to predict the class of unlabeled nodes based on existing node properties and relationships, allowing us for example to train a model to recognize a person likely to finance terrorism on the basis of their properties and relationships.

Node Classification and Regression are the equivalent of a classic Regression or Classification on a regular Relational Database, in our case we can use the potentialities of a Graph Database to add more functionalities to our Machine Learning applications such as the *Link Prediction*.

Link Prediction involves training a model to predicts the probability of a relationship forming between non-adjacent nodes, using the features created during training.

Thus using the Machine Learning package would allow us to compute a pipeline that, whenever a new Node is added to the Database, it perform a Classification and Link Prediction algorithms that will allow us to directly know if a node is suspect or not and which are the Nodes that are more probable it will link to in the future, making the analysis faster and faster.

Chapter 7

Conclusions

Research and analysis developed through the several chapters emphasized how *Graph Databases*, and in our case the *Neo4j* tool, provide a more effective solution for managing complex and interconnected data structures than traditional relational databases.

The limitations of relational databases, as discussed in earlier chapters, primarily revolve around their inability to efficiently process and query interconnected data without significant performance degradation. Relational databases require multiple joins, which not only increase query complexity but also slow down execution times, particularly in large-scale financial networks.

In contrast, graph databases eliminate these issues by representing data as nodes and edges, allowing for faster and more intuitive querying, especially when analyzing relationships between entities.

This advantage seems to be very useful in the financial sector, where fraud detection, anti-money laundering (AML) initiatives and regulatory compliance require the ability to store data and understand the intricate relationships between entities such as customers, transactions, and intermediaries, becoming crucial when investigating suspicious patterns or tracing fraudulent transactions across a network where data points looks unrelated to each other.

Introducing graph algorithms such as *Centrality*, *Community Detection*, *Link Prediction* and *Path Finding* into the financial crime compliance workflow significantly improves an organization's ability to detect anomalies and reduce risk.

Centrality measures help identify key players or influential nodes in a

network—individuals or organizations that are central to a web of suspicious activities, as well as identifying clients or people who possess characteristics that are abnormal for their type, being a red flag for possible illicit activities.

Community detection algorithms, such as the *Louvain* method and *WCCs*, allow investigators to group related entities and identify potential clusters of illegal activity or fraudulent schemes.

Furthermore, topological link prediction algorithms can estimate the likelihood of future connections between entities, providing early warnings of potential risk.

In the analysis performed on the company's adapted dataset and on the Offshore Leaks database it derives from, these algorithms were applied to identify suspicious relationships between customers, entities, and intermediaries involved in potentially illicit financial activities, and to compute metrics and graphic representations of nodes distributions and importance.

The use of Neo4j's *Cypher* query language and its Graph Data Science (GDS) library provided a means to gain a deeper understanding of the data set, revealing connections that were previously hidden and enabling more effective and accurate detection of fraudulent activity.

A significant finding of this research is that graph databases demonstrate superior performance, particularly in terms of scalability and ease of querying, compared to conventional systems based on SQL when handling complex, interconnected data.

Illegal financial activities typically involve intricate networks of intermediaries, shell companies and offshore entities, highlighting the relational nature of financial crime. This investigation shows how graph databases perfectly match these complex relationships, making them suitable for compliance activities involving the monitoring of convoluted chains of ownership, monetary transactions and corporate associations.

In the future, the integration of native machine learning from graph databases into financial crime compliance systems is a substantial advance. By using graph algorithms to predict fraudulent activities and assess risk, financial institutions can not only respond to financial crimes after they have occurred, but also take proactive measures to prevent them.

For example, predictive models trained on graph data can forecast suspicious behavior patterns, thus enabling preventive action against fraud.

Summing up, the examination conducted in this thesis and its results showed

the significant benefits of adopting graph databases for financial crime compliance, including their operational effectiveness and ability to unmask intricate fraudulent schemes. Graph databases offer a detailed, relationship-focused perspective on data, which is essential for today's compliance responsibilities. As financial networks become more complex and fraud tactics more sophisticated, the capabilities provided by graph databases will be indispensable.

In the future, further integrations of graph databases with machine learning may be developed, focusing on monitoring financial transactions in real time, expanding the ability to identify fraud as it occurs. In addition, as graph database technology advances, there will be opportunities to create even more customized algorithms specifically designed for compliance objectives, increasing even more their relevance in the financial sector.

Ultimately, the shift to graph databases represents more than just a technological improvement. It is an essential transformation to address the growing needs for financial crime compliance in the digital age. In this evolving interplay between data and malfeasance, it is connections that help us prevent consequences and lead us toward a more open and equitable financial future.

Bibliography

[1] Graph Database Market by Model Type (RDF, LPG, Hypergraph), Offering (Solutions, Services), Analysis Type (Community Analysis, Connectivity Aralysis, Centrality Analysis, Path Analysis), Vertical, and Region - Global Forecast to 2028. [1]

[2] ACFE - Data Analysis Techniques for Fraud Examiners.[2]

[3] IBM - Relational Databases[3]

[4] Neo4j Documentation[4]

[5] Francis, Nadime, et al. "Cypher: An evolving query language for property graphs." Proceedings of the 2018 international conference on management of data. 2018.

[6] He, Zhenzhen, Jiong Yu, and Binglei Guo. "Execution time prediction for cypher queries in the neo4j database using a learning approach." Symmetry 14.1 (2022): 55.

[7] Neo4j GDS Documentation[5]

[8] DatabaseTown - Relational Database Benefits and Limitations (Advantages & Disadvantages) [6]

[9] About the ICIJ - International Consortium of Investigative Journalists.[7]

[10] ICIJ Offshore Leaks database.[8]

---

[1]https://www.marketsandmarkets.com/Market-Reports/graph-database-market-126230231.html?gad_source=1&gclid=Cj0KCQjwlvW2BhDyARIsADnIe-IgitT4qv6V4G5Brju___4g-A76eSrK07am2A9QSxaC24qMAQMzaNoQaAhpMEALw_wcB

[2]https://www.acfe.com/

[3]https://www.ibm.com/topics/relational-databases

[4]https://neo4j.com/docs/2.1.5/

[5]https://neo4j.com/docs/graph-data-science/current/

[6]https://databasetown.com/relational-database-benefits-and-limitations/

[7]https://www.icij.org/about/

[8]https://offshoreleaks.icij.org/pages/about

[11] ICIJ Pandora Papers Dataset with Neo4j. [9]

[12] ICIJ Cyprus Confidential [10]

[13] « ISF gate » : comment de grandes fortunes françaises ont pratiqué l'évasion fiscale au Canada. [11]

[14] Stergiopoulos, George, et al. "Risk mitigation strategies for critical infrastructures based on graph centrality analysis." International Journal of Critical Infrastructure Protection 10 (2015): 34-44.

[15] Zhang, Junlong, and Yu Luo. "Degree centrality, betweenness centrality, and closeness centrality in social network." 2017 2nd international conference on modelling, simulation and applied mathematics (MSAM2017). Atlantis press, 2017.

[16] Linton C. Freeman, Centrality in social networks conceptual clarification, Social Networks, Volume 1, Issue 3, 1978, Pages 215-239.

[17] Bianchini, Monica, Marco Gori, and Franco Scarselli. "Inside pagerank." ACM Transactions on Internet Technology (TOIT) 5.1 (2005): 92-128.

[18] PageRank - Neo4j Data Science. [12]

[19] Li, Ye, et al. "Community detection in attributed graphs: An embedding approach." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 32. No. 1. 2018.

[20] Ghosh, Sayan, et al. "Distributed louvain algorithm for graph community detection." 2018 IEEE international parallel and distributed processing symposium (IPDPS). IEEE, 2018.

[21] Mehlhorn, Kurt, Peter Sanders, and Peter Sanders. Algorithms and data structures: The basic toolbox. Vol. 55. Berlin: Springer, 2008.

[22] Cormen Thomas H.; et al. (2009). "22.3". Introduction to Algorithms. MIT Press.

[23] Huang, Zan. "Link prediction based on graph topology: The predictive value of generalized clustering coefficient." Available at SSRN 1634014 (2010).

[9] https://neo4j.com/developer-blog/digging-into-the-icij-pandora-papers-dataset-with-neo4j/

[10] https://www.icij.org/investigations/cyprus-confidential/

[11] https://www.lemonde.fr/les-decodeurs/article/2021/12/16/isf-gate-comment-de-grandes-fortunes-francaises-ont-pratique-l-evasion-fiscale-au-canada_6106313_4355770.html

[12] https://neo4j.com/docs/graph-data-science/current/algorithms/page-rank/

[24] Neo4j Documentation, Common Neighbors. [13]
[25] Kunegis, Jérôme, Marcel Blattner, and Christine Moser. "Preferential attachment in online networks: Measurement and explanations." Proceedings of the 5th annual ACM web science conference. 2013.
[26] Barabasi, A.-L., and Albert, R. Emergence of scaling in random ´ networks. Science 286, 5439 (1999), 509–512.
[27] Neo4j Documentation, Preferential Attachment. [14]
[28] Neo4j Documentation, Machine Learning. [15]

[13] https://neo4j.com/docs/graph-data-science/current/alpha-algorithms/common-neighbors/

[14] https://neo4j.com/docs/graph-data-science/current/alpha-algorithms/preferential-attachment/

[15] https://neo4j.com/docs/graph-data-science/current/machine-learning/machine-learning/