POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Simple Structures For Complex Data: Optimizing metadata storage & retrieval using graph databases

Relatore: Prof. Pier Luca Lanzi
Correlatore: Prof. Lenore Zuck

Tesi di Laurea di:
Alessandro Panebianco, matricola 783941

Anno Accademico 2013-2014

Alla mia famiglia
Ai miei amici, in particolare a Daniele ed a Mario

Ringraziamenti

Un particolare ringraziamento ad entrambi i miei relatori, Prof. Pier Luca Lanzi e Prof.ssa Lenore Zuck, per il loro costante e pronto ausilio; mi hanno dato la possibilità di lavorare con la necessaria libertà fornendomi allo stesso tempo fondamentali linee guida che mi hanno portato alla conclusione di questa tesi con la giusta serenità. Un immenso grazie alla mia famiglia, senza la quale non avrei mai potuto affrontare questo arduo ma fantastico percorso. Infine, voglio ringraziare tutti i miei amici più cari che hanno aggiunto valore a questo già di per sè indescrivibile traguardo.

Contents

List of Figures

List of Tables

Abstract

Analyzing and processing the dynamic structure of today's data represents a challenge for developers which prompts them to explore alternative methods with respect to the standard ones. In this work we will present the benefits and the potential of the novel Graph databases, capable of handling complex data better than the traditional approaches. Relational databases for example, seem to not scale the task in the new emergent challenges. On March 2, 2009, the Foreign Intelligence Surveillance Court (FISC) found that the NSA had failed to comply with court ordered restrictions on searching "BR metadata," telephone business records containing information about American as well as foreign citizens. The NSA claimed that "from a technical standpoint, there was no single person who had a complete technical understanding of the BR FISA system architecture." Motivated by this court sentence, after an accurate simulation of a telephone dataset, we then present an application that, using graph databases, ensures that only a specific number of entries is permitted to be queried. Then, we introduce another feature of this application which allows to perform run time queries designed to modify the dataset in real-time.

Riassunto

Al giorno d’oggi, analizzare ed elaborare le struttere dinamiche dei dati moderni rappresenta una importante sfida per gli sviluppatori che li spinge ad esplorare sempre più, nuove tecnologie. Intorno agli inizi degli anni 2000, prende piede il movimento NOSQL("not only SQL"), sorto appunto dall’esigenza di garantire atomicità, consistenza, isolamento e durabilità ai dati con strutture più complesse rispetto a quelle viste sino a quel punto. In questa tesi illustriamo i benefici dei moderni Graph Databases, capaci di gestire dati a struttura complessa con risultati più perforamanti rispetto agli approcci tradizionali. Il nostro lavoro è stato motivato dalla sentenza della Surveillance Court (FISC) del 2 marzo 2009 in cui si affermava che la National Security Agency (NSA) non aveva mantenuto le restrizioni previste dalla corte durante le analisi dei "BR metadata", registrazioni telefoniche contenenti informazioni su cittadini americani e stranieri. La NSA si giustificava affermando che "dal punto di vista tecnico, non c’era nessuno che avesse una totale comprensione dell’architettura BR FISA".

In questa tesi, dopo un’accurata simulazione di un dataset telefonico, presentiamo una applicazione che, sfruttando le potenzialità dei graph databases, assicura che solo uno specifico numero di "records" sia autorizzato ad essere consultato, in linea con la sentenza di corte sopra menzionata. Una ulteriore funzionalità dell’applicazione permette inoltre di effettuare query a run-time con lo scopo di modificare il dataset in tempo reale.

Chapter 1

Introduction

During the past decade in the world of computer science we have witnessed significant changes in how data is perceived. The vast amount of information that today fills the web demands new complex data structures; moreover, an inverse proportion exists between complex structures and resources utilized for storing them. This fact motivates programmers to pursue constant optimization during their analysis.

In most cases, the first challenging task that developers encounter is storing these complex structures and in particular focusing on which database schema is best to adopt for the application scope.

Historically, analysts have learned different databases models: the *Hierarchical* model, the *Network* model and the *Relational* among others. Relational databases have been leading the market for the past fifty years, along with the SQL query language. As it often happens in the market, the most successful product is not the best solution for each application. This reason expedited the movement NOSQL(not only SQL) in the early 2000s. The name tried to give a name to the need of a growing number of non-relational, distributed data stores; people in the NOSQL enviroment claimed that traditional relational database systems several times did not provide atomicity, consistency, isolation and durability guarantees which are fundamental for any type of application. It has been decades since the relational model default has kept developers from individuating their real back-end requirements. The NoSQL movement instead, has offered programmers the chance to discover what they actually require from databases, and to find out that there is no one solution that fits all. From the advent of the relational model, several things have changed. Firstly, data generated and processed by modern applications increased exponentially; on the contrary, the wait for the queries response time had dropped to sub-second units, contrasting from the 80s when "calculators" were computing for entire days; another important aspect has changed: the data structure. Data today is completely dynamic and developers are increasingly unwilling to force a specific structure on data a priori.

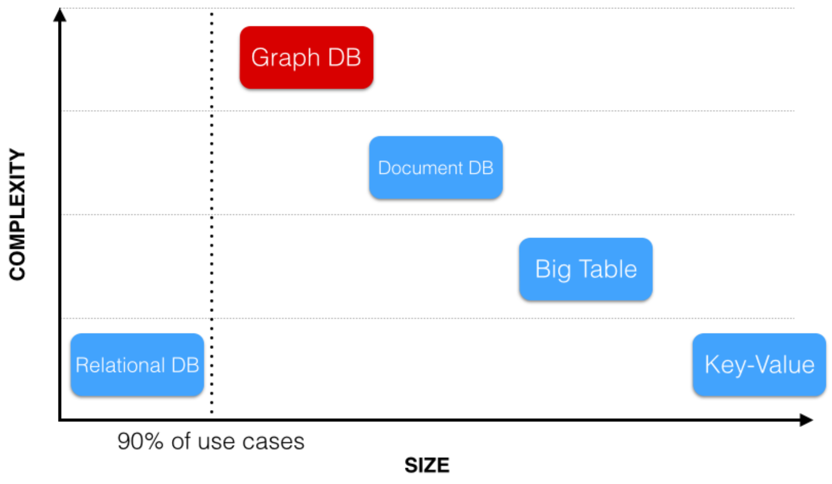


Figure 1.1: Data representation in terms of size and complexity

One NoSQL technology that has emerged a couple of years ago is the graph database. Graph database are often found in scenarios where the data model is considerably connected, including social, telecommunications, logistics, master data management, bioinformatics and fraud detection. Graph databases partially recall the network database model mentioned above. The main difference is that network databases are still mostly based on a hierarchical data model in terms of parent-child relationships. This also means that in network databases we cannot relate arbitrary records to each other, which makes it hard to work with graph-oriented datasets. For example, we may use a graph database to analyze what relationships exist between entities. Another difference is that network databases use fixed records with a predefined set of fields, while graph databases use the more flexible Property Graph Model which we present in chapter 4.

In this work we manage data with a complex and highly connected structure. Our dataset simulates a telephone graph following a power law where each node represents a user and each edge represents a call made between two users. After having thoroughly explained in chapter

3 why graph databases best fit our dataset compared to the relational databases, we implement an application which performs queries on an embedded graph database modifying at run time the structure of the dataset.

1.1 Motivation behind this work

On March 2, 2009, the Foreign Intelligence Surveillance Court (FISC) found that the NSA had failed to comply with court ordered restrictions on searching "BR metadata", telephone business records containing information about American as well as foreign citizens. The NSA claimed that "from a technical standpoint, there was no single person who had a complete technical understanding of the BR FISA system architecture."(Page 8 of Docket Number: BR 08-13.)[1] A series of Foreign Intelligence Surveillance Court (FISC) documents concerning the US National Security Agency's use of bulk telephony metadata was recently declassified and released. A March 2, 2009 order from the FISC is particularly interesting. It reveals that the NSA was querying all the identifiers on an NSA alert list against BR metadata that it received daily. The FISC ruled that this violated previous FISC orders, and that the NSA was entitled to query only identifiers for which there was a "reasonable articulable suspicion" (RAS) that the identifier was relevant and related to terrorist activity. Only about 2000 of the identifiers on the alert list had been determined to have RAS. One of the more remarkable justifications that the NSA gave for its actions was that "from a technical standpoint, there was no single person who had a complete technical understanding of the BR FISA system architecture.". Judge Walton noted that the NSA's explanations "strained credulity" and that "the court is exceptionally concerned about what appears to be a flagrant violation of its order in this matter".

To build a technical understanding of the ruling and the NSA's arguments, we must first interpret certain key terms: "BR metadata", "RAS". BR stands for "Business Records". For telephony, according to the declassified documents, metadata includes the sender/receiver, originating device, time of conversation, call duration, and the trunk number, (i.e., the point at which a cell-phone conversation enters the main phone system, which identifies the location to within about a square kilometer.6)

[1]Extracted by *A Modest Proposal to the NSA* Chris Kanich, Alessandro Panebianco, Robert H. Sloan, Richard Warner, and Lenore D. Zuck

RAS is "reasonable articulable suspicion" that a phone number under investigation is "relevant and related to terrorist activity". The NSA queried the BR metadata using identifiers that were not RAS-approved, and misrepresented its query practices to the court by describing them as consistent with the above requirements.

At least recently, the FISC allowed quite generous searches from RAS identifiers. An August 2013 Obama administration white paper on the bulk collection of telephony metadata together with a recent report from the Privacy and Civil Liberties Oversight Board (PCLOB), make it clear that the NSA is currently allowed to search in the BR metadata up to a distance of three from RAS identifiers, referred to as three "hops". In other words, treating the BR data as an undirected graph whose nodes are phone numbers and whose edges are phone calls, the NSA can examine the record of any phone call both of whose phone numbers are within three edges of a RAS phone number.

Given this scenario we then propose a technique that would allow the NSA to ensure compliance as well as proofs of compliance. We then propose algorithms to help explain how identifiers might be added to the RAS list.

1.2 Goals of this thesis

1.2.1 Approach

After the generation of a dataset (which is described in the next subsection), we focus on finding the best database candidate to help us simulate the query process described in the motivation section. We end up choosing the novel Graph Databases which perfectly meet our demands and lead us to perform our analysis with solid results. In particular, we use Neo4j, a highly scalable, robust (fully ACID) native graph database. In order to reproduce the entire query process, we decided to develop a Java application along with with the Neo4j APIs. Our choice was driven by the fact that the Java APIs offered by Neo4j were perfectly documented and simultaneously well performing. We exhibit performances in chapter 5.

1.2.2 Dataset

One of the goals of this dissertation is to remain consistent with the scenario described in the court sentences we examined generating an accurate phone dataset. A handful of papers written in the past fifteen years analyze various telephone call graphs. For this investigation, we base primarily on *Aiello et al.*, who found that the telephone call graph data they examined (calls for one day), treated as an undirected graph, had a power law degree distribution with a power law exponent of 2.1 and an average degree density of 3.16. A brief overview of the power law is presented in Chapter 2.

1.2.3 Run time actions

Having considered the scenario described in the motivation section, we added some functions to our Java application in order to perform run time actions on the dataset. In line with the court sentences, we propose a mechanism that detects special paths in the phone dataset and automatically labels them, creating an automatic addition of an identifier to the RAS list. The full implementation in described in chapter 4.

1.3 Organization of the thesis

The following chapters of this dissertation are organized as follows. Chapter 2 presents a background knowledge of Graph Theory that we consider necessary to have a full understanding of the whole thesis; a brief description of the Power Law graphs will conclude the second chapter. Chapter 3 will be dedicated to the approach we followed and the reasons why we decided to use Graph Databases. In Chapter 4 we describe the implementation phase, starting from the generation of the dataset and ending with the creation of a Java application that interacts with the graph databases and performs run-time actions. Chapter 5 exhibits the results we obtained with our experiments and finally, chapter 6 presents our conclusions which end this dissertation.

Chapter 2

Background

2.1 Graph Theory

Graph databases find their origin in the Graph theory; basic mathematical concepts have been used to define the original schema of this novel technology. In this chapter we are presenting the main aspects which the reader should possess in order to have a full understanding of this dissertation.

2.1.1 Graphs

The first concept we analyze is what a graph is. A graph can be broadly defined as a representation of a network, "a collection of objects connected in some fashion". In our case for example we use the idea of graph to represent a network of telephone calls.

With node, or vertex, we refer to each object in a graph. The connections between nodes are called edges or links. Every edge connects one node to another.

Formally, we define a graph G as an ordered pair $G = (V,E)$ where V is a set of nodes, or vertices, and E is a set of edges (links). The elements of E are 2-element subsets of V.

In this dissertation, each node represents a user and each edge is a call between a pair of users. To define who calls who, we need to recall the definition of *undirected* and *directed* graphs.

2.1.2 Undirected and Directed

In an undirected graph the edges of a graph are unordered pairs of nodes so for example if in an undirected graph we write we write {2,3} to indicate the edge that connects node 2 to node 3 it will not be different from the form {3,2}

In a directed graph instead we take into account the direction of the edge, this is the reason why we graphically substitute edges with arrows (and we usually use parenthesis instead of brackets to denote ordered pairs). An edge (2,3) starts from node 2, ends at node 3 and it will have the exactly opposite meaning of an edge (3,2). An example of directed graph is shown in figure 2.1

In this dissertation we refer to directed graphs where a user (node) A calls a different user (node) B if there is an edge that starts from A (*initial vertex*) and finishes at B (*terminal vertex*). Follows the reason why we do not allow *loops* which are cases where initial vertex and terminal vertex coincide; it would not make sense for someone to call her own number.

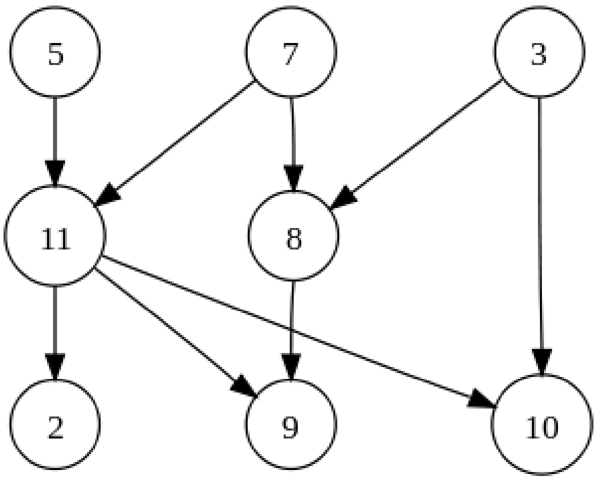


Figure 2.1: An example of directed graph

2.1.3 Neighborhood and Degree

Another aspect which we are interested to recall is the neighborhood and degree of a node.

We define two vertices adjacent if there is a common edge that connects them. The degree of a vertex is the total number of vertices adjacent to the vertex. In an equivalent way, we can define the degree of a vertex as the cardinality of its neighborhood. We use the degree of a node or equivalently, its neighbors, to generate our dataset. The details of the implementation are described in chapter 4.

2.1.4 Density and Average Degree

We also want to recall the concept of density of a graph and average degree, since during the generation of our dataset we follow the work done by *Aiello et al.* which gives us the guidelines to set these parameters for our phone graph.

The density of a graph *G = (V,E)* measures how many edges are in set E with respect to the maximum possible number of edges between vertices in set *V*. The density in a directed graph, which has no loops, can have at most |V| * (|V| - 1) edges, so the density of a directed graph is |E| / (|V| * (|V| - 1))

The average degree of a graph G represents another way to measure how many edges are in set E with respect to the number of vertices in set V. The average degree of a graph G represents another way to measure how many edges are in set E with respect to the number of vertices in set V.

2.1.5 Paths

As we already mentioned in the introduction of this dissertation one of the goals of this thesis is to find specific patterns in the graph. When we write the word pattern we explicitly refer to the definition of path in a graph. We can think of a path as a way of walking the graph in a unique way, starting from an origin and finishing to a specific destination. More formally we define a path P as an ordered list of directed edges: *P = ((n1,n2),(n2,n3),...,(nk,nk+1))*. The initial node of the first edge of a path is the origin and the second node of the last edge is the destination. The origin along with the destination are called endpoints of the path.

2.1.6 Distance

A fundamental concept for this work is the distance in a graph. As we explained in the previous chapter, we know that NSA is currently allowed to search in the BR metadata up to a distance of three from RAS identifiers, referred to as three "hops".
In a graph *G*, the distance *d(x,y)* between two nodes x and y is the length of the shortest path from x to y, considering all possible paths in G from x to y. The distance between any node and itself is 0. If there is no path from x to y then distance *d(x,y)* is infinity.

In our case we apply this definition to our phone dataset and we want to recall that 2 vertices distant 1-hop is equivalent to say that the same

2 vertices have a distance equal to 1.

2.2 Power law graphs

For the simulation of the phone dataset we focus primarily on the model presented in the paper by *Aiello et al.*; they demonstrate the consistency of the model they propose with the behavior of certain massive graphs derived from telecommunications data.

Aiello et al. show that the degree sequence of so called call graphs is nicely approximated by a power law distribution. They define call graphs as graphs of phone calls handled by groups of telephony carriers for a given time period. This definition perfectly matches with our scenario, since we simulate call graphs extracted by presumably all the carriers of the USA. *Aiello et al.* also state that there are three standard models for what they call uniform random graphs. Each has two parameters. The first parameters controls the number of nodes in the graph and the second one controls the density, or number of edges. Following their example, the random graph model *G(n, m)* assigns uniform probability to all graphs with n nodes and m edges while in the random graph model *G(n, p)* each edge in an n node graph is chosen with probability p. In this dissertation we are implicitly using this model through the python library *igraph*. The details are described in chapter 4.

Aiello et al. use a random graph model that was originally derived from massive graphs generated by long distance telephone calls. They explain that these so-called call graphs are analyzed in different time intervals. Our dataset wants to simulate all the calls made in one day since the bulks of data given to the NSA, as stated in the court sentences, were released every day. Summarizing, the structure of our dataset consists of an edge in the graph for every finished phone call: every phone number which either sends or receives a call is a node in the graph and finally when a node originates a call, the edge is directed out of the node and contributes to that node's outdegree.

Chapter 3

Approach

In this chapter we illustrate the approach we have followed to implement our solution. We explain the process that took us to adopt the graph databases for our application and we show how we have adapted the Neo4j graph databases for our purposes.

3.1 Graph Databases

3.1.1 Reasons behind the adoption of Graph Databases

When relational databases were first designed, their main scope was to manage structures similar to what real existing paper forms looked like: tables for the major part. At the same time, relational database design had to make sure that relationships among those tables were able to model the real world situations. The drawbacks was that relationships were only a metaphore of joining tables which obviously put some limits on the potential of relational databases, like for example the necessity of elucidate the semantics of those relationships, or weighting them depending on their strenght. Another difficulty was that datasets were also not simple, there were often large and not uniformly distributed tables that had to be joined causing a heavy work for the entire model. The large increase of connected data led to a consequent growth of the number joins which impacted negatively on performances of the relational databases, creating the need for a new business change.

We know that depending on the scope od the application we have consequences on the design of the final schema, obtaining some quite simple queries and some others more complicated:

1. Difficulty is proportional to the number of JOIN tables since they combine business data with foreign key metadata.

2. The foreign key constraints bring additional maintenance and development overhead only to run the database.

3. Not uniform tables often contain null columns which need to be checked in the code although a schema already exists

4. Several expensive joins are needed just to discover what a customer bought.

5. Queries like "what articles did a person buy?" are less expensive than "which person bought this product who also bought that article?" and this is what a typical recommendation system is all about. An index can be added but the recursive queries like "Which person bought this article who also bought that article" tend to be pretty heavy as the recursion grows.

In this dissertation we face the quintessential of connected domains, a call graph; as we previously mentioned relational databases often conflict with highly connected domains. In order to emphasize this observation we show a simple but effective example of query with connected data.

3.1.2 SQL example

For this example we refer to a simplified social network-like dataset containing users and their friend relationships. In a relational database, we should typically have two relational tables for storing social network data: one for storing user information, and another one that stores the relationships between users (see figure 3.1)

How our actual tables with data look like is shown in table 3.1 and table 3.2.

To draw a parallel with our phone dataset, there would be a table for storing user information such as phone number, device number, trunk number, etc. and another one for storing calls between users. We prefer to use social network dataset because we consider it to be the most simple example to understand and at the same time, the closest to our analysis.

Listing 3.1: SQL code defining tables for social network data

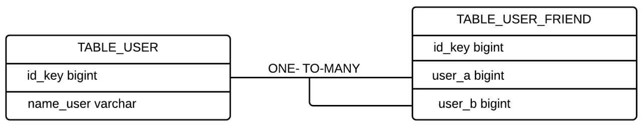


Figure 3.1: Relationship between the user and the friend table

Table 3.1: table_user

id_key	name_user
1	Alex
2	Bob
3	Carl
5	Dan
6	Erin

Table 3.2: table_user_friend

id_key	user_a	user_b
100	1	2
101	3	5
102	4	1
103	6	2
104	4	5
105	1	4

```
create table table_user (                                 #A
        id_key bigint not null,
        name_user varchar(255) not null,
        primary key (id_key));

create table table_user_friend (                          #B
        id_key bigint not null,
        user_a bigint not null,
        user_b bigint not null,
    primary key (id_key));

alter table table_user_friend                             #C
    add index FK417045CBC6132571 (user_a),
    add constraint FK417045CBC6132571
        foreign key (user_a) references table_user (id_key);

alter table table_user_friend
    add index FK417045CBC6132572 (user_b),
    add constraint FK417045CBC6132572
        foreign key (user_b) references table_user (id_key);
```

#A Table definition for storing user information

#B Table definition for storing friendship relations

#C Foreign key constraints

The table table_user stores columns with information, table table_user_friend instead contains two columns pointing to table table_user using a foreign key relation. Both the primary key and the foreign key columns are indexed for faster search operations; this is a common way to proceed with relational database models.

The next step is how we query some data. For example, it is pretty straightforward to retrieve direct friends of a particular user:

```
select distinct user_b from table_user_friend where user_a = 1;
```

We are interested in retrieving all friends of some user's friend. This is when commonly we have to use a JOIN between the table_user_friends and itself.

In this thesis we focus on queries which explore the database in depth, meaning that we are interested in knowing which people a user called and in turn, which people these people called and so on. This is a first sketch of what the concept of "hops" is but we will define it more formally in the next chapter. In the social network example this type of query is the popular friends of friends query. To perform such a query to find friends of friends of a user, we would need another join operation: To retrieve a four hops query we would need to perform four joins. Five hops would mean five joins and so on.

This approach is the typical correct approach that a SQL developer would follow but the drawback is that if for example we are only interested in getting only one user's friend of friends, we would perform a join for all the data contained in the table_user_friend table and then throw away all the entries which we are not interested in. If we consider a modest data set this concern would not exist, although we know that for a big and connected dataset as in our case the call graph, the performances would tremendously decrease. In order to show this flaw in relational databases we have run the friends of friends query on a modest dataset of only one thousand users, and we set an average of each user having fifty friends. Mathematically speaking, we multiply

1000 times fifty and we have around fifty thousand tuples.

The only thing we try to optimize is adding an index that we have by the way defined in the SQL code above showed. We have launched the friends of friends query with number of hops equal to two, three, four, five. In the table 3.3 we illustrate the execution time for each of those.

Table 3.3: Execution time: friends of friends query.

# Hops	Execution time(in seconds) for 1000 users	Records returned
2	0.032	~900
3	0.244	~999
4	10.676	~999
5	96.541	~999

Results in table 3.3 show how with the number of hops up to two and three, SQL seem to lead to sound performances (considering also that we have added indexes). What we notice with four and five is instead a considerable decrease of performances reaching more than ten seconds for the first query and more than a minute and a half for the second one even though the number of rows returned is the same.

Inefficiency of SQL joins

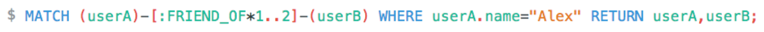
As we mentioned earlier joining several tables and discarding most of the results does decrease the performances with a relational database system. We have noticed it in the example from the previous section, especially when we brought our query to reach five hops of distance from the user. We retrieved around fifty thousand of entries but we discarded almost the entire result set for in the end getting only about 1000 entries. We now start introducing Neo4J and show what results brings with the same dataset.

3.1.3 Neo4j

Since Neo4j is a graph database it structures data as a graph, using nodes for vertices and relationships for edges. We then represent users as nodes and if two of them are friends there will be a relationship that connects them that represents the friendship relationship. The real big difference with the relational databases is the way data are queried: Neo4j refers to a concept named graph traversal, taken from graph theory and bases its fundamental concepts around it. It represents the alternative candidate of using the SQL joins on rows and columns, opening a novel way to retrieve data.

3.1.4 Traversing the graph

The traversal simply consists in visiting nodes moving across their relationships; it is strictly related to the graph model and it makes it the most important feature. The most important factor for the traversals is that queries are localized and they run only on the data that is actually required, different from the SQL example where we have discarded the major part of the resulting sets, thus avoiding heavy actions and leading to a a general better performance. For this example we run a little piece of Cypher (the graph query language created by the Neo4j development team).



```
$ MATCH (userA)-[:FRIEND_OF*1..2]-(userB) WHERE userA.name="Alex" RETURN userA,userB;
```

Figure 3.2: Traversing with Neo4j

For now we are interested in explaining how the query is structured. It starts from a subset of nodes and it navigates edge by edge (relationship by relationship) collecting all the nodes that match a certain pattern specified in the query. In figure for example 3.2 we match all those users who are up to 2 hops distant from a certain user, in this case the user called *Alex*. Neo4j has been conceived to stop as soon as the traversal impact the nodes matching a certain pattern thus there is no wasting of space as it was for the SQL example, improving performances. There are so many ways we can retrieve data with Neo4j and we are going to explain some of them in the next chapter, showing the solid potential of graph databases.

We will see in the next chapter how to handle nodes variables and properties with Cypher but let us look at the performances so we can make a comparison with the SQL experiment numbers. The dataset it is exactly the same as before. Results are listed in table 3.4.

We immediately observe how results with Neo4j win over the SQL ones, as we stated before the main improvement that is also the one

Table 3.4: Execution time: friends of friends query with Neo4j

# Hops	Execution time(in seconds) for 1000 users	Records returned
2	0.041	~900
3	0.067	~999
4	0.072	~999
5	0.074	~999

we are more interested in starts when the number of hops is equal or greater than 3. The deeper we go, the better results we get using graph databases. We once again want to highlight that one of the main factors which slowed down performances for the SQL queries is situated in the query structure: for each level of depth there is a join with the table itself to perform and this means to add extra Cartesian product operations which are known to be computationally heavy. And we also point out that the other main cause of the bad performances is the fact that in the SQL case, most of the resulting dataset is discarded. Opposite is the method how Neo4j navigates the graph, avoiding the inclusion of extra data that doesn't belong to the actual resulting dataset. So far we have reasoned on a quite small dataset, let us now try to enlarge the dataset until it reaches the amount of data that we are going to use for our application, that is to say more than one million nodes.

3.1.5 Large scale queries comparison

The dataset we have queried so far led us to certain conclusions but as we previously mentioned in this dissertation we are interested in querying big datasets, reaching and exceeding the one million nodes. We cannot confirm yet that the principles stated in the previous section are valid also for larger datasets, that is the reason why we now run our last experiments increasing the number of nodes so we can fully justify our choice of adopting graph databases for our application. We leave everything with the same structure we have seen in the previous section, same tables but populating the table_user with 1 million tuples; recalling the fact that the average friends for each user is fifty users, we touch about 50 millions tuples in the table_user_friend table. Same schema, same queries. Table 3.5 shows the results relative to the SQL performances.

We notice that when the number of hops is equal to two, performances remain almost the same as the ones we have seen for the smaller

Table 3.5: Execution time: friends of friends query with 1 million users with SQL

# Hops	Execution time(in seconds) for 1 million users	Records returned
2	0.022	~2500
3	31.244	~125000
4	1550.349	~600000
5	stopped after 3 hours	/

dataset. Adding indexes confirm the fact that table joins successfully speeded up. Unfortunately, when we pass to a number of hops equal or greater than three we start loosing reasonable resulting times. We even had to stop the query after 3 hours when we set the number of hops equal to five. We conclude that when it is only a single join query, SQL can be taken into account even in considerable big datasets but when we switch to more than a single join, performances drop drastically. In table 3.6 we show now how Neo4j performed with the same dataset.

Table 3.6: Execution time: friends of friends query with 1 million users with Neo4j

# Hops	Execution time(in seconds) for 1 million users	Records returned
2	0.012	~2500
3	0.178	~115000
4	1.423	~600000
5	2.488	~800000

We immediately notice how the impact of the dataset enlargement seems to have not impacted Neo4j's performances. Results confirm to be solid and we see a little latency, due to the greater number of data retrieved by the query with respect to the almost 1 thousand tuples of the previous section example, in fact in this case the query returned a number of rows that reached almost one million of nodes returned, which we understand it is a big number. The winning aspect of Neo4j in this scenario is the fact that the traversal only visits nodes which are linked to the starting nodes and there is no need to discard any extra node since Neo4j keeps track of every node it visits. After these considerations we consider worthwhile to adopt the graph databases to develop our application.

Chapter 4

Implementation

In this chapter we analyze the details of the implementation process. The first part is dedicated to the explanation of how the dataset is generated. Following, we describe how we implemented our Java application to interact with the graph dataset and to search and modify particular patterns in the graph.

4.1 Dataset

4.1.1 The Property Graph Model

We introduced the property graph model in Chapter 1; we now highlight its salient features.

1. Nodes, relationships, and properties define the property graph.

2. Nodes include properties.

3. Relationships link and give a structure to the nodes. They have either an outgoing direction or an ingoing direction. They can also have a label and there is always a starting node and an ending node for each relationship. Relationships add semantic to the structure of the graph.

4. Relationships as well can have properties. Adding a property to a relationship help giving extra information for graph algorithm purposes but also , as we will see later in this dissertation, to run constrained queries at runtime. We can think of weighted edges to have an immediate example of a property applied on a relationship.

These simple primitives represent all we need to create sophisticated and semantically rich models as our phone dataset.

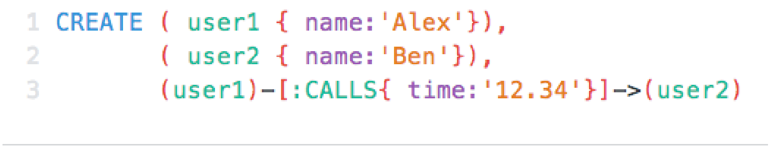
For telephony, according to the declassified documents, metadata includes several fields which we include in every node of our graph. Precisely, every node will include the following fields:

1. Call sender number;
2. Originating device number
3. Time of conversation
4. Call duration
5. Trunk number(i.e., the point at which a cell-phone conversation enters the main phone system, which identifies the location to within about a square kilometer).

In order to go further, we need a mechanism for creating, manipulating, and querying data. We need a query language. We briefly sketched some features in the approach chapter but let us now have a better overview of Cypher.

4.1.2 Cypher Query Language

Cypher represents an expressive and compact graph database query language. It is specific to the Neo4j tool but the way it is conceived makes it a perfect candidate for representing graphs using diagrams and describing them in a precise fashion. As the majority of query languages, Cypher is made of clauses. The first thing we need is the CREATE clause which we use for inserting data in the graph. Here is a simple example of a Cypher query to show how we are going to implement our database.



```
CREATE ( user1 { name:'Alex'}),
       ( user2 { name:'Ben'}),
       (user1)-[:CALLS{ time:'12.34'}]->(user2)
```

Figure 4.1: Cypher query example

With the CREATE clause shown in figure 4.1 we are generating nodes in our graph, in this example two nodes are added, *Alex* and *Ben*. We can obtain a first visualization thanks to the embedded graphic interface that comes together with the Neo4j tool in figure 4.2. Please notice that *user1* and *user2* are variables that will last only for one session. A session starts when the server is started and it ends when the server is stopped. In our case we are using these two variables to generate our first relationship. Since our dataset, as specified previously, is dedicated to a telephone dataset, from now on we will use only one type of relationship between nodes, "CALLS". What the reader can also notice is that with Neo4j we can set properties on the relationship too. In this case we fixed a time for the call happened between Alex and Ben.

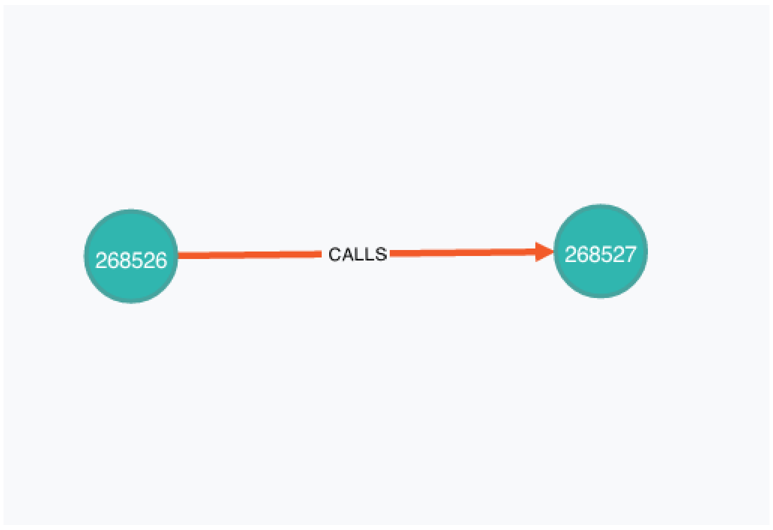
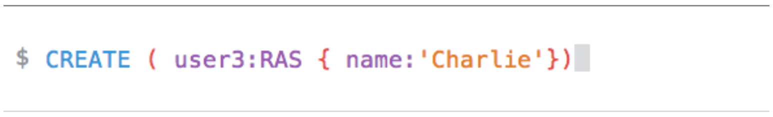


Figure 4.2: Cypher example structure

We think it is also important to spend a few words on the feature that the Neo4j team introduced in their version 2.0, that is the "label" function.

Labels are the closest instrument to the SQL indexes, they divide set of nodes in different groups, speeding up the whole query process and giving a more straightforward sense to the organization of the graph. A single node can have multiple labels by the way. In this dissertation labels are going to be useful for our application since they are going to allow us to perform live actions thanks to the algorithm we describe later in this chapter.

In figure 4.3 we show the correct syntax of how we use label in our telephone dataset. We want to make a distinction between nodes that are considered "reasonable articulable suspicion", meaning that they represent phone numbers under investigation and "relevant and related to terrorist activity", and the ones that are not RAS.



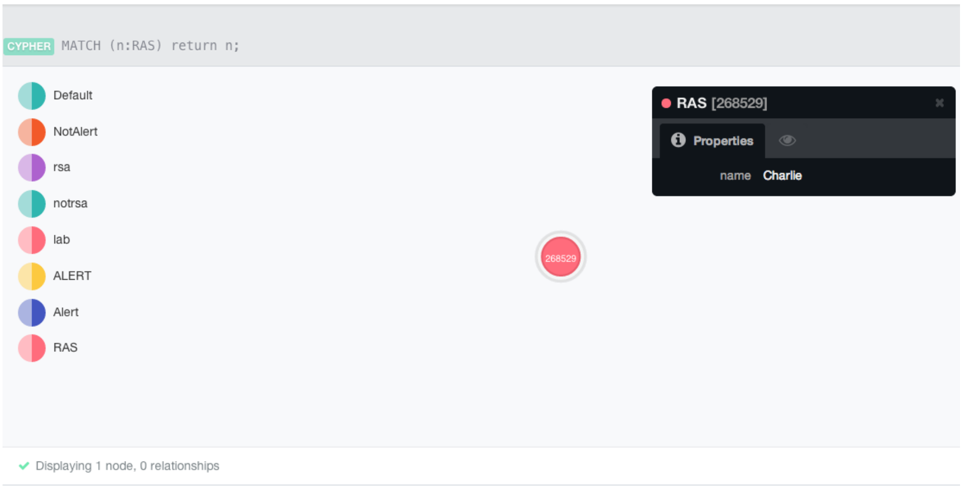
```
$ CREATE ( user3:RAS { name:'Charlie'})
```

Figure 4.3: Creating a label

The node *Charlie* is created and it has a label added on that specifies that it belongs to the RAS list. Like *Charlie*, in this thesis we label all the nodes that belong to the RAS list so that every query we want to start from the subset of nodes which belong to that list will be improved in terms of speed.

The Cypher clause MATCH represent the principal method of retrieving data from the database; it permits us to specify the pattern that we want to match in a quite straight way: it uses the ASCII characters to recall the nodes and the shapes of the relationships. For example every node will be "framed" in two parethesis *()* so that it looks like the well-known circular shape of a node. For the edges Cypher uses —>, two dashes and a greater-than sign(or a less-than sign, it depends on the direction of the edge) that represents the original edge shape. For example we try to retrieve the node *Charlie* we have previously created. The fact that we labeled it as a "RAS" node allows us to launch a query where we command that we only want the labeled nodes, without specifying the fact that we are looking for the user *Charlie*. Figure 4.4 shows the query that retrieves the node just created using only the label.

The other important aspect we want to focus on in this thesis is how to retrieve queries, recalling the social network example in Chapter III, "friends-of-friends"like, which in this thesis we call a "hops-distance" query. In particular we want to create an application that given a phone dataset with a subset of users belonging to a certain list(the RAS list), performs queries that return users who are k-hops distant from them. In a more simple way, if a user A called user B and user B called user C, user C is 2-hops distant from A. To retrieve such data with Cypher we



```
CYPHER MATCH (n:RAS) return n;
```

Figure 4.4: MATCH labeled nodes

use the following syntax showed in figure 4.5.

```
$ MATCH (userA)-[r:CALLS]->(userB)-[r2:CALLS]->(userC) return userA,userB,userC;
```

Figure 4.5: MATCH friends-of-friends nodes

4.1.3 Power law graph

In this thesis one of the goals is to create the dataset that reproduces the same scenario that NSA analysts daily faced. We want to be faithful to the descriptions found in the papers presented to the judge describing how the data was represented during the NSA analysis. First of all, we have to follow a model that fits telephone data. Aiello et al. found that the telephone call graph data they examined (calls for one day), treated as an undirected graph, had a power law degree distribution with power law exponent of 2.1 and average degree density of 3.16. We wrote a Python script using the function *Static_Power_Law()* belonging to the package *igraph* which allowed us to specify the following parameters:

1. *n* - the number of vertices in the graph

2. *m* - the number of edges in the graph

3. *exponent_out* - the exponent of the out-degree distribution,

We take advantage of the *Static_Power_Law()* function to generate Cypher statement which we use later to populate our graph database. In details, we first generate the graph following the parameters suggested by Aiello et al.; for each node of the graph just generated we print the parallel Cypher statement which generates a node in our graph db; for each node's neighbor, taking advantage of the function *neighbors()*, we also create the Cypher statement that generates an edge between the node and its neighbor. It can happen that when the Python script "walks" in a node, that node doesn't need to be created because it was already generated as a neighbor of another node, in that case we make sure that only an edge is created between the node and its neighbor thanks to the Cypher MERGE clause. From the definition, MERGE ensures that a pattern exists in the graph: either the pattern already exists, or it needs to be created.

We decided to generate 1 million nodes in order to recreate the average daily scenario the NSA had to face. It is relevant to highlight the fact that only 300 nodes are labeled as "RAS", meaning that they represent the only nodes which are eligible to be queried.

All experiments were performed on an MacBook Air with 4 GB of RAM and a 1.8GHz Intel Core i5 processor.

The output of the script is passed to the neo4j-console through the command -file output.txt. We by the way enclose our Cypher strings with BEGIN and COMMIT every about 1000 statements to speed up the populating process.

4.2 Java Application

After the dataset was populated our intentions are to build an application that interfaces with it and allow us to reproduce the query process the NSA analysts were supposed to do according to what they said in the court notes.

We choose to use to develop a Java application using Eclipse Standard/SDK, version: Kepler Service Release 1 and embedding Neo4j by including the Neo4j library jars in our build.

First thing we do is open the the existing database which has been previously filled with our phone dataset located in the variable DB_PATH.

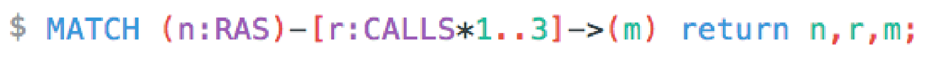
Listing 4.1: Embedding the db

```

graphDb = new GraphDatabaseFactory().newEmbeddedDatabase( DB_PATH );

registerShutdownHook( graphDb );
```

We also set some other properties on our code. In order to make sure the database shuts down when the JVM exits we create a *registerShutdownHook()*; we ensure that every operation we perform during the query process is wrapped into a transaction thanks to the *graphDb.beginTx()* function.

We then perform our queries calling *engine.execute()* and passing our Cypher query. Our scope, as mentioned in several parts of this dissertation is to query the dataset and obtaining a subset of all the nodes(users) who are distant 3 or less hops from the nodes labeled as "RAS". We recall that as we stated in the introduction of this thesis from the court notes, NSA is currently allowed to search in the BR metadata up to a distance of three from RAS identifiers, referred to as three "hops".

The Cypher query we are embedding in the *engine.execute()* function is the one showed in figure 4.6.



```
$ MATCH (n:RAS)-[r:CALLS*1..3]->(m) return n,r,m;
```

Figure 4.6: Cypher query

It is important to take a look at the query in the detail and analyze the syntax used. Writing *(n:RAS)* we are referring to all the labeled nodes that belong to the list of the suspects which the analysts are allowed to query. In the square brackets we specify using the Cypher * notation that we want to include in our result dataset all the nodes *m* which are distant maximum 3 hops from the RAS *n* nodes.

To fetch the dataset we use a Java map interface as shown in listing 4.3.

Listing 4.2: Fetching the resulting dataset

```

    try ( Transaction tx1 = graphDb.beginTx()){

            ExecutionEngine engine = new ExecutionEngine( graphDb );
            ExecutionResult result = engine.execute( "MATCH␣(n)-[:
                CALLS*1..3]->(m)␣return␣n,m" );

            Iterator<Node> iteratore = result.columnAs("n");

            while (iteratore.hasNext()) {
                Node nodeType = iteratore.next();

                for (String propertyKey : nodeType.getPropertyKeys())
                    {
                    System.out.println("\t" + propertyKey + "␣:␣" +
                        nodeType.getProperty(propertyKey));
                            }
                        }
```

In this way we have our resulting dataset in the map object and we can analyze what our result data sets from the Java application.

4.3 Run-time actions

So far we have reached our very first goal of this work. We can create subsets of the initial big data set and retrieve information on those nodes. By the way it would be great to perform live actions on those resulting data sets. We reasoned on which approach was best to follow given the instruments that Neo4j offers, and we ended up using once again the labeling process in a slightly different way from the one we used previously. We want to split our query process in two steps. At first we match the subsets as we have seen in the previous section but instead of only matching them, we already label them as "target" while in the second part of the query we perform our actions. The goal of these actions is finding out some special paths in the subsets we obtained from the queries above discussed. We want to create a mechanism which, given some rules as input, automatically suggests which nodes could represents the new RAS nodes, unloading both the judge's work and the analyst's one. We are not law experts so we define just one sample action to show what implementation should be followed. Based on the available data our first idea was labeling all those users who talked with more than 3 RAS suspects in a short period of time, one week for example.

Here is the code to achieve such a result:

Listing 4.3: Fetching the resulting dataset

```

    try ( Transaction tx1 = graphDb.beginTx()){

            ExecutionEngine engine = new ExecutionEngine( graphDb );

            ExecutionResult result = engine.execute(
                    MATCH (n:RAS)-[r:CALLS*1..3]->(m)
                    SET m:TEMP
                    WITH m
                    MATCH (m)-[r2:CALLS]->(q:RAS)
                    WHERE r2.time>1388534400 AND r2.time<1389052800
                    WITH m,collect([r2,q]) as paths
                    WHERE length(paths) = 3
                    RETURN m,paths
                    );

            Iterator<Node> iteratore = result.columnAs("n");

            while (iteratore.hasNext()) {
                Node nodeType = iteratore.next();

                for (String propertyKey : nodeType.getPropertyKeys())
                    {
                    System.out.println("\t" + propertyKey + "␣:␣" +
                        nodeType.getProperty(propertyKey));
                            }
                        }
```

In the next chapter we are going to present and discuss all the results that we have obtained from our application.

Chapter 5

Evaluation

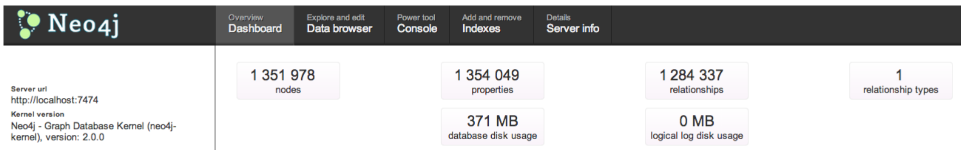
In the following chapter we proceed showing results in terms of time and space performances. We also add graphs which have been created thanks to the browser interface contained in the Neo4j tool. The interface has a fine design and some simple CSS rules can be applied to modify colors and dimensions of the graph. Due to its new introduction of this interface(available from the Neo4j 1.8 version), we notice some graphic and control issues, which by the way did not prevent us to have a meaningful view of the resulting graphs.

5.1 Dataset

Figure 5.1 shows the dataset we have created before running our application and its consequent queries. We exceed one million nodes reaching almost 1,4 million nodes with the same number of properties. Memory used for storing the entire database is 371 MB which we retain to be acceptable, considered the more than one million nodes present in the db.

5.2 Query Process

Here we show how the queries are visualized through the interface created and designed by the Neo4j team in the Neo4j tool versions after 1.8. As we stated previously in this dissertation, our goal is to query around 300 nodes out of more than 1 million nodes in the dataset. Those 300 nodes should simulate the RAS list that the NSA analysts possessed at the time of the querying process. Since we want to keep the results the most realistic possible, we performed our queries on 10 different sets



Server url
http://localhost:7474
Kernel version
Neo4j - Graph Database Kernel (neo4j-kernel), version: 2.0.0

1 351 978 nodes
1 354 049 properties
1 284 337 relationships
1 relationship types
371 MB database disk usage
0 MB logical log disk usage

Figure 5.1: Dimension of the dataset

containing 300 random nodes. We also choose to run the queries with 10 different sets also because as we described in the Random Graphs section, the degree of the nodes changes depending on the node id, the lower ids (meaning the first ones created) have a lower degree that is to say that they have less outgoing edges from the node compared to the higher ids nodes. We believe that adding these extra experiments we keep our results consistent.

We started recording performances with 1000 nodes and we gradually expanded reaching the peak of nodes at 1351978 nodes. In the first table we include the partial results with the first set of random nodes. Please note that before we have run our experiments we were aware that 300 nodes out of 1 million is obviously different from 300 out of 1000. In fact for the 1000,10000,100000 sets we kept the ratio as 300/1 milion when choosing the number of random nodes.

The results of the tests are shown in Table 5.1.

Table 5.1: Performances: 1 Set

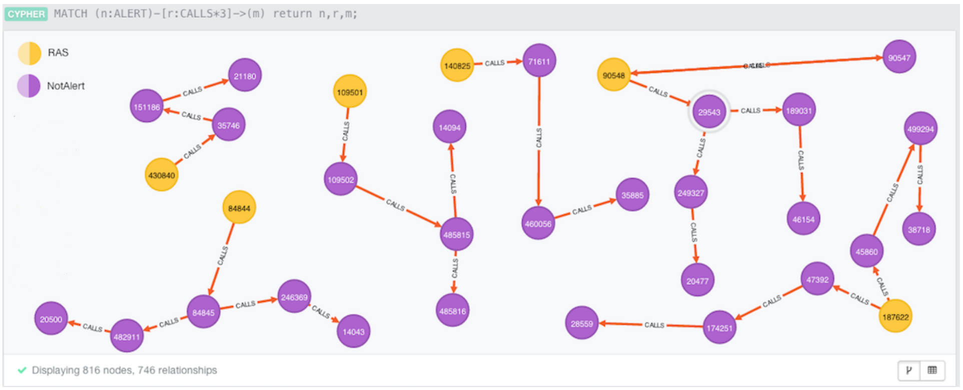
# Nodes	3 Hops query	3 Hops + Action[1]
~1000	187s	250s
~10000	243s	356s
~100000	296s	439s
~1000000	355s	506s
~1500000	578s	845s

Instead of showing all the tables with the different 10 random nodes sets, we show in table 5.2 the average of the ten simulations. We ran these experiments only on the biggest dataset meaning the one with 1351978 nodes.

Table 5.2: Performances: Average 10 sets

# Nodes	3 Hops query	3 Hops + Action[1]
~1500000	594s	861s

In Figure 5.2 we can visualize thanks to the Neo4j interface tool how the results look like in a more graph theory style. As mentioned at the beginning of this chapter there are still some issues with this tool that have to be solved like for example the fact that the result graph is showed in a way that cannot be controlled; actions like zooming in or zooming out are not supported. In fact Neo4j team suggests to embed some other graphic tool to the project like for example Gephi. The goal of this thesis is not to create nice graphs that is why we recommend the reader to follow the suggestion by the Neo4j team.


CYPHER `MATCH (n:ALERT)-[r:CALLS*3]->(m) return n,r,m;`

✓ Displaying 816 nodes, 746 relationships

Figure 5.2: Graph visualization of the 3-hops query

What is represented in Figure 5.2 is the result of the query to get all the nodes 3 hops distant from the RAS nodes. There can be several types of structures depending on the degrees of the nodes involved in each subset. In yellow we represent the RAS nodes, in purple all the nodes not belonging to the RAS list.

Figure 5.3 shows an example of the run-time action result. Since there is a node that has made a call to 3 different RAS users in a period of

[1] All those users who talked with more than 3 RAS suspects in a week

time minor or equal than one week, it has been labeled with a "suspect" label. This label in the figure is red colored while the RAS nodes are yellow and the other nodes not having any label are colored in grey.

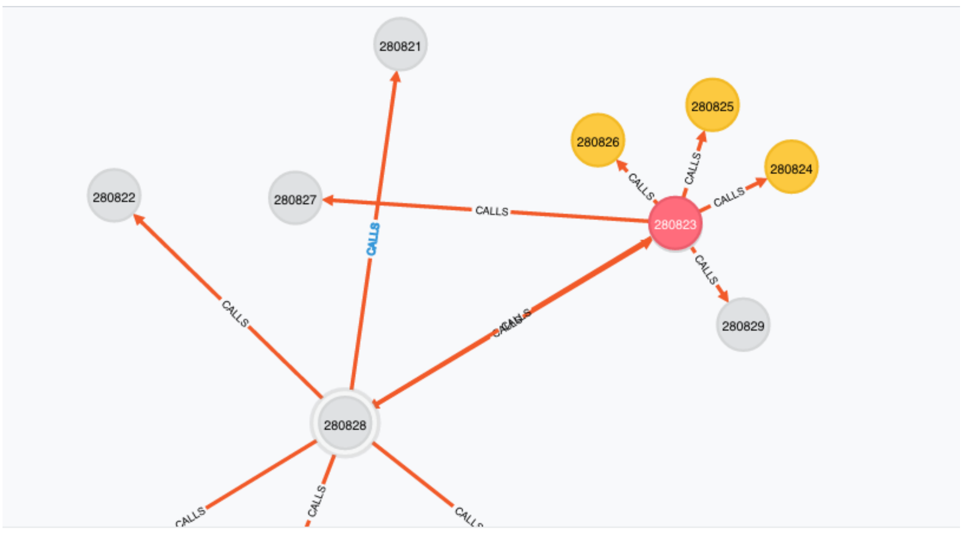


Figure 5.3: Graph visualization of the run-time action

Chapter 6

Conclusions

In this dissertation we showed that processing complex structured data led us to discover novel methods of managing data. Motivated by what the NSA answered when the FISC in 2009 accused them to have failed with the court ordered restrictions on querying the phone metadata, claiming that "from a technical standpoint, there was no single person who had a complete technical understanding of the BR FISA system architecture" we first generated a phone call dataset following a power law graph and then we have simulated the query process described in the FISC court sentences. We have also presented an automatic way to perform run time actions designed to be a valid and meaningful suggestion to improve the analyst work. We performed all our experiments using the modern graph databases which allowed us to make the whole process scalable and at the same time easy to understand, so that a statement such as the one made by the NSA could not be tolerated anymore.

Although there are different database models that can be used to manage data, we presented the reasons why we adopted the graph databases for processing telephone data. We believe that since the query process practically recalled the concept of distance in Graph Theory, we chose graph databases; they allowed us to perform straightforward queries, optimizing time and space.

We also consider our results to be consistent. As described in the the chapter "Performances" we ran our experiments on 10 different sets, each time extracting a subset of random nodes which ensured that there was no bias in the nodes' degree. Our dataset reached almost 1,5 million nodes which we assume to be a realistic number to simulate the daily work of a NSA analyst.

Future Works In the end, we intent to propose potential enhancements that may extend our work.

Firstly, we generated our dataset trying to follow both all the descriptions found in the court sentences and the material we had available but it would be quite interesting to process and analyze "anonymized" data from a real corporate such as AT&T.

In addition, we have showed some graph visualization of the query system using the web interface of the Neo4j tool. We reached our demonstrative scope but as we mentioned at the beginning of chapter 5, due to its recent release, the graphic interface has some little bugs that did not allow us for example to have an overall graph view. We aim to embed our project with a graphic tool such as Gephi to have some more detailed graph visualization of the data first of all and secondly, to have a more accurate control of the entire graphic interface.

Lastly, we developed a mechanism to make run time modifies on the entire dataset in case of a legally suspicious path in the obtained graph; we applied our computer science knowledge to achieve such a function but our expertises did not allow us to have a legal technical point of view. This is why we would be interested in initiating a collaboration with a law technical team to combine both the computer science and the legal fields and make this instrument fully valid for important future investigations.

Cited Literature

[1] FISA Ct.: In re Production of Tangible Things From [REDACTED], Order. Number BR 08-13, March 2009. http://www.fas.org/irp/agency/doj/fisa/fisc-021209.pdf.

[2] Chris Kanich, Alessandro Panebianco, Robert H. Sloan, Richard Warner, and Lenore D. Zuck: A Modest Proposal to the NSA, 2013.

[3] Aiello, W., Chung, F., Lu, L.: A random graph model for massive graphs. In: Proc. 32nd Annual ACM Symp. Theory of Computing. (2000) 171-180.

[4] Reinhard Diestel: Graph Theory. Electronic Edition 2000, Springer-Verlag New York 1997, 2000.

[5] Martin Charles Golumbic: Algorithmic Graph Theory, Second edition, Elsevier, 2004.

[6] Avi Silberschatz, Henry F. Korth, S. Sudarshan Database System Concepts, Fourth Edition, McGraw-Hill, 2001.

[7] US Department of Justice: Administration white paper: Bulk collection of telephony metatdata under section 215 of the USA Patriot Act (August 2013) Available from https://www.aclu.org/nsa-documents-released-public-june-2013.

[8] Privacy and Civil Liberties Oversight Board: Report on the telephone records program conducted under section 215 of the USA PATRIOT Act and on the operations of the Foreign Intelligence Surveillance Court. Available from https://www.fas.org/irp/offdocs/pclob-215.pdf.

[9] Jonas Partner, Aleksa Vukotic, and Nicki Watt: Neo4j in Action, MEAP Began, June 2012.

[10] Ian Robinson,Jim Webber,Emil Eifrem: Graph Databases, O'Really, July 2013.

[11] The Neo4j Manual v2.0.1: The Neo4j Team `http://docs.neo4j.org/chunked/stable/index.html`.