Máster Universitario en

Ingeniería y Tecnología

de Sistemas Software

Universitat Politècnica de València

in collaboration with the

Technische Universität Wien

Analysis and style of Graph-Oriented Database Technology, focused on the Neo4j System

Final Master's Project

(Trabajo Fin de Máster)

2017 - 2018

Author: Italo Lombardi

Tutors: Prof. Dr. Juan C. Casamayor

Prof. Dr. Reinhard Pichler

To my family, who inspire me to reach high.

Acknowledgments

Mis más sinceros agradecimientos a todos los profesores del **Máster Universitario en Ingeniería y Tecnología de Sistemas Software**, de manera especial al **Profesor Juan Carlos Casamayor**, mi tutor, y a la **Directora Profesora María Carmen Penadés**.

Juan Carlos, con su curso de TGD, despertó en mí el interés por las bases de datos orientadas a grafos, tecnología que considero muy importante para mi futura carrera laboral. Su profesionalidad y amabilidad, que noté desde el primer día, fueron el impulso para pedirle que fuera mi tutor para esta tesis. Espero haberle hecho sentir orgulloso de haber aceptado esta tarea.

María Carmen, demostrando profesionalidad y paciencia hacia mi en todo momento, dándome la oportunidad de participar en dos ocasiones en intercambios Erasmus, siempre resolviendo las solicitudes y dudas surgidas en su momento.

Special thanks also go to my Tutor in **Vienna**, **Prof. Reinhard Pichler**, who allowed me to live this fantastic experience that has changed me profoundly.

Gracias también a la **Profesora Silvia Abrahão** que me recomendó el Màster.
Sin ella, todo esto no hubiera sido posible y espero que algún día nuestros caminos académicos vuelvan a cruzarse.

Mi mayor afecto va a **Carlos**, **Enio**, **Juanjo** y **Marco** que han sufrido conmigo durante la redacción de los mil proyectos que hemos realizado. A **Fernando**, **Jacinto**, **Patty** y **Yandry**, compañeros durante y despúes de los exámenes y a todos los otros compañeros del Máster. Os considero mucho más que sólo amigos.

Hablando de amigos, quiero agradecer a **Fabricio** y **Anca**, **Daniel** y **Temi**, **Paola P.**, **Cristhian**, **Felix**, **Irving**, **Jorge**, **Omar**, **Paolo** y a todos los demás que han hecho de **Valencia** mi segundo hogar.

Grazie anche agli **amici** di **Fisciano** e di **Buonabitacolo**, che, anche nella lontananza, riescono sempre a trovare un momento per condividere con me gioie e dolori.

Dedico questa tesi a **tutta la mia famiglia**, che mi ha sempre supportato, sperando di averli resi fieri dopo questo duro lavoro.

Resumen

El objetivo del trabajo fin de máster es enfocarse en los aspectos fundamentales de las bases de datos orientadas a grafos, en particular sobre Neo4j, considerado uno de los sistemas de gestión de bases de datos más importantes y estables, para manejar datos usando un almacenamiento nativo en grafos.

Cada día, las bases de datos orientadas a grafos son cada vez más populares que las tradicionales bases de datos relacionales, por lo tanto, al principio, veremos una descripción general y una comparación entre las dos tecnologías.

En segundo lugar, al entrar en más detalles sobre el sistema Neo4j, mostraremos todas sus características externas e internas, que le han permitido alcanzar un amplio dominio.

Posteriormente, nos enfocaremos en Cypher, el lenguaje declarativo del Neo4j, analizando las funciones principales que se ofrecen.

Una vez adquirido el conocimiento, se mostrará cómo crear y administrar un proyecto, incluida la instalación y configuración, tratando de informar al lector sobre los errores más comunes, que se pueden hacer durante las fases de creación de la base de datos.

Finalmente, mostraremos uno de los casos de uso más famosos de Neo4j, eBay ShopBot, un sistema que usa un grafo para almacenar e identificar productos rápidamente, para ayudar a los clientes durante las compras en el sitio web.

Palabras clave: Grafo, Base de datos orientada a grafos, NoSQL, Neo4j, Cypher, Modelado de datos, eBay ShopBot

Abstract

The primary aim of this Master's Thesis is to focus on the fundamental aspects of graph-oriented databases, with a particular emphasis on Neo4j, considered one of the most important and stable graph database management system to handle data using native graph storage.

Every day, Graph Databases are becoming more popular than traditional Relational Databases, for this reason, this work will begin with an overview and a critical comparison between the two technologies.

The research will follow with an in-depth analysis of the Neo4j system, and it will show its external and internal characteristics, which have allowed it to reach enormous popularity.

Furthermore, it will focus on Cypher, the Neo4j's declarative language, analysing the primary functions offered.

Once knowledge is acquired on this subject, this assignment will discuss how to create and manage a project, including installation, configuration and most common mistakes, that can be made during the database creation phases.

In conclusion, this work will show one of the most famous use cases of Neo4j, the eBay ShopBot, a system which uses a graph to store and identify products fastly, to help customers during the shopping on the website.

Keywords: Graph, Graph Database, NoSQL, Neo4j,
Cypher, Data Modeling, eBay ShopBot

Table of contents

List of figures

Chapter I

1) Introduction

1.1) Background motivation

The last decades have seen a radical change in the computer world, with the birth of an increasing number of applications, aimed at the processing of large amounts of data.

In particular, the advent of social networks has considerably increased the amount of data generated, causing problems of storage and management.

The need to develop new applications that can not be addressed with the traditional relational database approach has led to the search for alternative solutions.

NoSQL data storage and management systems are attracting increasing interest, and graph-oriented databases can be the solution to many problems currently present.

These databases do not presuppose a rigid structure or a schema describing the properties that data must have and the relationships between them. In fact, non-relational databases aim to be more flexible and faster depending on the type of problem to be solved.

1.2) Goal

The primary aim of this Master's Thesis is to focus on the fundamental aspects of graph-oriented databases, with a particular emphasis on Neo4j, considered one of the most important and stable graph database management system to handle data using native graph storage and processing.

1.3) Structure of the dissertation

This work will begin with an overview and a critical comparison between Graph Databases and Relational Databases.

The research will follow with an in-depth analysis of the Neo4j system, and it will show its external and internal characteristics, which have allowed it to reach enormous popularity.

Furthermore, it will focus on Cypher, the Neo4j's declarative language, analysing the primary functions offered.

Once knowledge is acquired on this subject, this assignment will discuss how to create and manage a project, including installation, configuration and most common mistakes, that can be made during the database creation phases.

In conclusion, this work will show one of the most famous use cases of Neo4j, the eBay ShopBot, an information system which uses a graph to store and identify products fastly, to help customers during the shopping on the website.

Chapter II

2) Preliminary

This dissertation has been written to take a tour into the graph database world, with emphasis on the system *Neo4j*, developed in the last few years, to find out how to solve new problematics issues connected with the enormous increasing volume of data to manage.

An overview of the fundamental concepts is essential to make it possible to understand all the topics. The first notion is databases, to identify the general functionality, and then graphs will be covered to introduce Neo4j.

2.1) Databases - Overview

A database is primarily *a means of organising information*. This is not something limited to electronics because, essentially, everything organised as a structured set of information, can be considered as a database. However, the increased number of information has motivated the necessity to migrate to a more reliable database system that isn't paper-based. [1]

The essential characteristics of databases are: [21]

- *Large*: they can have enormous dimensions and in general much larger than the available central memory.
- *Shared*: different users and applications must be able to access common data according to appropriate methods. This feature reduces data redundancy since repetitions are avoided, and consequently the possibility of inconsistencies, due to the presence of non-actualized data on different systems, is also reduced.
- *Persistent*: databases have a lifetime that is not limited to that of the individual executions of the programs that use them.

The data contained in a database can be divided into two categories: [22]

- The ***metadata***: that is the **database schema**, which describes the data structure, the restrictions on the possible values, the relationships existing between the sets and sometimes also some operations that can be performed on the data.
  The schema must be defined before creating the data, and it is independent of the applications that use the database.
- The actual ***data***: that is the representations of particular facts that conform to the definitions on the database scheme.
  They are organised in homogeneous sets, among which relations are defined. The structure of the data and the relationships are described in the schema with appropriate mechanisms of *abstraction* depending on the **data model** adopted, which consists of the set of concepts used to organise the data of interest and describe the structure so that it is understandable to a computer.

A good data model should be characterised by:

- *Expressiveness*: the model should allow representing naturally and directly the meaning of what is being modelled.
- *Ease of use*: the model should be based on a minimum number of mechanisms that are easy to use and understand.
- *Feasibility*: the mechanisms of the abstraction model and the relative operators for data manipulation must be feasible in an efficient way on a database system.

A database system, usually called **Database Management System** (**DBMS**), allows the interaction with the data stored via a predetermined language.

These kinds of interactions can be categorised into three sections:

- *Data definition*: actions that modify the organisation of the data.
- *Data manipulation*: which includes both the operations to manipulate the actual data stored (information creating, updating and deleting) and the actions where data is selected from the database to be reused in another application.
- *Administration*: actions of user management, performance analysis, security and other higher-level activities.

The task of the DBMS is therefore to guarantee:

- *Reliability*: the ability of the system to keep the contents of the database intact.
- *Privacy of data*: which means that each user is enabled to perform only specific actions on data, through authorisation mechanisms.
- *Efficiency*: the ability to perform operations using a set of resources (time and space) that are acceptable to users.
- *Effectiveness*: the capacity of the database to make the activities of its users productive.

Currently, there are different database management systems, each one with defined characteristics, advantages and disadvantages.

2.1.1) Relational Databases

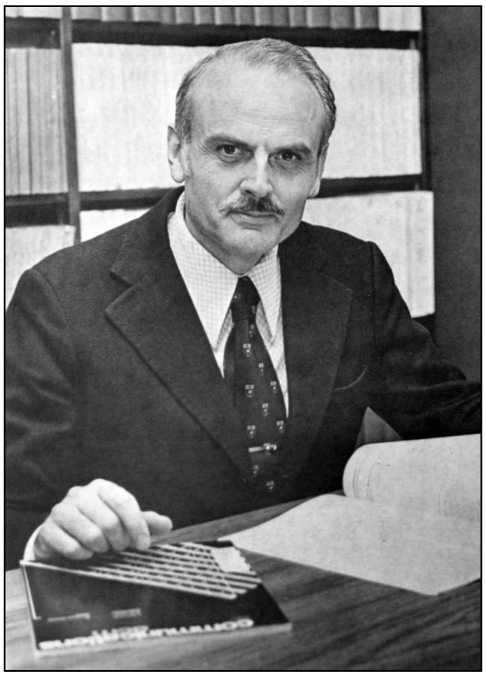


Figure 2.1.1: Edgar Frank Codd

Relational Databases[1] are probably the most familiar kind of databases used by 21st-century computer scientists.

They are based on the *Relational data model* [17] proposed by **Edgar Frank Codd**[2] (*Figure 2.1.1*), while he was working for IBM[3] on hard disk research projects, in 1970.

The model is planted on a brand of mathematics called *relational algebra*.

Codd, taking advantages of the power of mathematical abstraction, developed a simple but powerful structure for databases, used to manage vast amounts of data efficiently.

He wrote several papers proving that, using a mathematical representation called *tuple calculus*[4], sets of data would be an excellent way to organise and access

[1] https://docs.oracle.com/javase/tutorial/jdbc/overview/database.html
[2] https://en.wikipedia.org/wiki/Edgar_F._Codd
[3] https://www.ibm.com/
[4] https://www.tutorialcup.com/dbms/relational-calculus.htm

information, using a proper query language. That became the inspiration for declarative query languages such as **Structured Query Language** (**SQL**)[5].

In other words, the relational model provides a natural way to view data and serves as a specification for a Relational Database Management System (RDBMS).

The relational model is based on two concepts, ***relation*** and ***table***, two ideas of a different nature but easily referable to each other. The notion of relation comes from mathematics, mainly from the *Set theory*, while the concept of a table is simple and intuitive. The presence of these two concepts, one formal and the other intuitive, led to the great success obtained by the model. [22]

The relational model responds to the requirement of data independence, which provides a distinction in the description of data between the *physical* and the *logical* level:

- the users who access the data and the programmers who develop the applications only refer to the logical level;
- the data described at the logical level are then realised employing appropriate physical structures, but to access the data, it is not necessary to know them.

In mathematics, considering two sets D_1 and D_2, the *Cartesian product*, written D_1 x D_2, is the set of ordered pairs (v_1, v_2), such that v_1 is an element of D_1 and v_2 is an element of D_2, or in other words, is the resulted set generated from all combinations of each element in both sets.

For example, given the set A = {1,2,3} and B = {a, b} the Cartesian product A x B consists of the set of all the possible pairs in which the first element belongs to A and the second to B. In this case, it is composed of six pairs:

A x B = {(1,a), (1,b), (2,a), (2,b), (3,a), (3,b)}

A *mathematical relation* on the sets D_1 and D_2, called *domains* of the relation, is a subset of D_1 x D_2. Given the sets A and B above, a possible mathematical relation is constituted by the set of pairs {(1,a), (1,b), (3,b)}.

Any subset of this Cartesian product is a relation.

Mathematical relations can be represented graphically in tabular form. The two tables in the *Figure 2.1.2* describe the Cartesian product A x B and the mathematical relation on A and B discussed in the example.

[5] https://www.w3schools.com/sql/

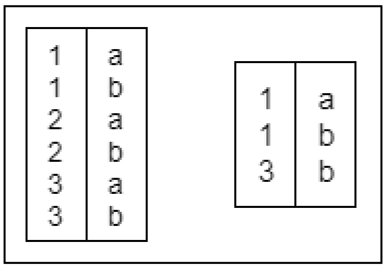

1 a
1 b
2 a
2 b
3 a
3 b
1 a
1 b
3 b

Figure 2.1.2: Tabular representation of a Cartesian product and a relation

If giving some conditions for the pair selection, it is possible to select only the pairs with specific characteristics. For example, to select pairs with a specific value at the first position: $\{(x, y) \mid x \in D_1, y \in D_2, \text{and } x = *value*\}$

These definitions could be extended to define a general relation on n sets (domains). In this hypothesis, given $n > 0$ sets (D_1, D_2, …, D_n), not necessarily distinct, the Cartesian product is defined as the set of n-tuples $(v_1, v_2, \ldots, v_n)$ such that v_i belongs to D_i for $1 \leq i \leq n$:

$$D_1 \times D_2 \times \ldots \times D_n = \{(v_1, v_2, \ldots, v_n) \mid v_1 \in D_1, v_2 \in D_2, \ldots, v_n \in D_n\}$$

In defining relations we specify the domains from which we chose values.

The number n of the components of the Cartesian product (and therefore for each n-tuple) is called *degree* of the Cartesian product and the relation. The number of elements of the relation is called, in Set theory, *cardinality of the relation*.

Each n-tuple is *ordered internally*: the i-th value of each comes from the i-th domain. This helps us to interpret the data in the relation because by exchanging the order of the domains, the meaning would change completely.

To alleviate this problem, it is possible to modify the definition of relation, introducing *attributes*: to each domain occurrence in the relation, it is associated a name, called attribute, that describes the element to which it refers.

In this hypothesis, therefore, the order is irrelevant, because it is no longer necessary to speak of "first domain", "second" and so on, but it is sufficient to refer to the attributes. The correspondence between attributes and domains is established employing a function $dom : X \rightarrow D$, which associates at each attribute $A \in X$ a domain $dom(A) \in D$.

Then, it is possible to say that a **tuple** on a set of attributes X is a function t that associates a value of the domain $dom(A)$ to each attribute $A \in X$.

With these assumptions, therefore, we can provide a new definition of relation: a relation on X is a set of tuples on X.

The difference between this new definition and the traditional mathematical one lies only in the definition of tuple: in the mathematical relation, there are n-tuples whose elements are identified by position, while in the new definition, the elements are identified through the attributes, with a non-positional technique.

This definition, therefore, leads to a new notation: if *t* is a tuple on X and
A $\in$ X, then *t*[A] indicates the value of *t* on A.

A relation can, therefore, be used to organise relevant data in the context of an application of interest. Therefore, usually, a single relationship is not sufficient for this purpose: a database is generally made up of several relationships, whose tuples contain common values, where necessary to establish correspondences.

The advantages of the relational model, compared to other existing models[6], are varied:

- It requires representing what is relevant from the user's point of view only.
- The logical representation of data, consisting only of values, does not refer to the physical one, which can change over time. This, therefore, allows the physical independence of the data to be obtained.
- Since all the information contained in the values is simple, it is easy to transfer data from one context to another (for example, if you need to transfer a database from one computer to another).

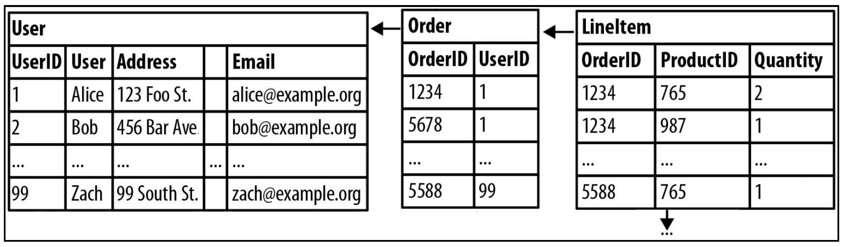
Thus entering in detail in the definitions related to the relational model, we can distinguish the level of the schemes from that of the instances:

- A ***relational scheme*** consists of a symbol *R*, called *name of the relation*, and a set of *attributes* X = $\{A_1, A_2, \dots, A_n\}$. Usually it is indicated with *R*(X).
  A domain is associated with each attribute.
- A ***database schema*** is a set of relational schemas with different names in order to distinguish between the many:
  $$\boldsymbol{R} = \{R_1\,(X_1), R_2(X_2), \dots, R_n\,(X_n)\}$$
- A ***relational instance*** on a fixed scheme *R*(X) is a set *r* of tuples on X.
- A ***database instance*** of a fixed schema $R = \{R_1\,(X_1), R_2(X_2), \dots, R_n\,(X_n)\}$, is a set of relations $\boldsymbol{r} = \{r_1, r_2, \dots, r_n\}$ where each r_i, for $1 \le i \le n$, is a relation on the schema $R_i(X_i)$.

[6] For example https://en.wikipedia.org/wiki/Hierarchical_database_model

Therefore, relational databases are collections of relation variables that present the user with tables (*relations*) of data values in columns (*attributes*) and rows (*tuples*) (*Fig. 2.1.3*) [18]:

- Each relation has a name and is made up of named attributes of data.
- Each tuple contains one value per attribute.
- The data within the database is stored following a well-defined structure, called *relational schema*, which represents all the relationships that the entities in a data source have.



User

UserID	User	Address		Email
1	Alice	123 Foo St.		alice@example.org
2	Bob	456 Bar Ave		bob@example.org
...	...	...	...	...
99	Zach	99 South St.		zach@example.org

Order

OrderID	UserID
1234	1
5678	1
...	...
5588	99

LineItem

OrderID	ProductID	Quantity
1234	765	2
1234	987	1
...	...	...
5588	765	1

...

Figure 2.1.3: Example of a Relational Database

By definition, all tuples within a relation should be distinct. The concept of key was introduced to maintain the uniqueness of them in a relation. [19]

The **key** of a relation is the non-empty subset of its attributes, that is used to identify each tuple uniquely in a relation. The attributes of a relation that make up the key are called *prime* or *key attributes*.

The superset of a key is usually known as **superkey** which uniquely identifies each tuple within a relation and may contain additional attributes.

A minimal superkey is called **candidate key**. Usually, there are more than one candidate keys in a relation schema, but only one is chosen as **primary key** which is the obligatory unique identifier for every tuple in a relation.

Data integrity is ensured by a set of rules or constraints, which specifies a condition and a proposition that must be maintained as true. Database modifications may violate these constraints, could create anomalies in the data.

In the relational data model, there are two principal integrity constraints:

1. **Entity integrity constraint**: which specifies that each attribute of a primary key must be not null, in order to provide uniqueness of tuples.
2. **Referential integrity constraint**: expressed in terms of **foreign key**, which is an attribute or a set of attributes within one relation that matches the key of some other relation.

This constraint specifies that if a foreign key exists in a relation either the foreign key value must match a key value of some tuples in another relation (or the same), or the foreign key value must be null.

Codd by describing a model based on relations, without navigational links or pointer structures, created something uniquely powerful, flexible and of permanent relevance. Several are the objectives of the relational data model:

- Allow a high degree of independence between the application program and the stored data. In this case, application programs must not be affected by modifications to the internal data representation.
- Provide techniques like the *normalisation*[7], to deal with data semantics, consistency and redundancy problems.
- Enable the expansion and standardisation of set-oriented data manipulation languages.

Disparate are the reasons why relational databases are the most popular databases used in the world at the moment[8]:

- ❖ Relational databases can cover different real-life scenarios easily, using systematic methods of managing and storing data.
- ❖ The data schema is independent of any particular programming language, version of the application, or purpose.
- ❖ SQL, used in programming and designed for managing data held in an RDBMS, became a standard of the *ANSI* (*American National Standards Institute*[9]) in 1986, and of the *ISO* (*International Organization for Standardization*[10]) in 1987.
  It is declarative because it describes what results are required, rather than methods to obtain them. For its nature, it is easy to learn and understand, making querying data simple.
  It is free-format, which means that parts of statements not required a specific locations on the screen to be executed.
  To support all the required interaction with the data, SQL has two major components:
    - ➢ *DDL*: Data Definition Language used to define the database structure;
    - ➢ *DML*: Data Manipulation Language for retrieving and updating data.

[7] https://beginnersbook.com/2015/05/normalization-in-dbms/
[8] https://db-engines.com/en/ranking
[9] https://www.ansi.org/
[10] https://www.iso.org/

Relational DBMSs have dominated the database sector for about forty years, but changes in information technology during the last decade have highlighted some limits.

A first limitation is regarding the rigid schema and low flexibility that relational databases have. This means that they impose a schema even before we put any data in it, but the actual reality shows that many domains require applying different database schema to all the elements that make up.

The second and bigger limitation is related to relationships: the relationships between two or more entities are treated in several ways depending on their cardinality[11]. A many-to-many relationship[12], for example, is a type of cardinality that refers to the relationship between two entities A and B in which A may contain a parent instance for which there are many children in B and vice versa.

These relationships are created using *joining tables*[13] (*Figure 2.1.4*). In these tables are usually present identifiers (primary keys) used to create relationships between different entities.

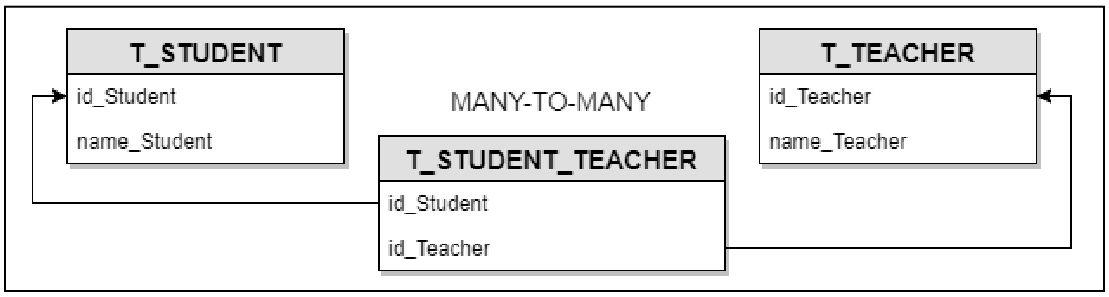


Figure 2.1.4: Example of a Join Table

This approach works when it comes to relating small amounts of data, but the simplicity of the Relational Database Systems comes with other problems:

- *Size matter*: the query response times get worse when tables grow longer.
- *Difficult relationships*: when the relations become more complex, these systems start to become very difficult to work with. Join operations become notably tricky to program and extremely resource intensive for the DBMS.

[11] https://www.sqa.org.uk/e-learning/SoftDevRDS02CD/page_44.htm
[12] https://en.wikipedia.org/wiki/Many-to-many_(data_model)
[13] https://www.w3schools.com/sql/sql_join.asp

2.1.2) NoSQL Databases

The explosion of web content marked a new era for databases, with a new generation categorised under the name **NoSQL databases**[14] (the acronym of **Not Only Structured Query Language**).

The first and most important motivation was born out of the frustration with relational systems, which cannot be used to manage all kinds of scenarios in our reality.

Using the official definition from Wikipedia: "*A NoSQL database provides a mechanism for storage and retrieval of data that is modeled in means other than the tabular relations used in relational databases*"[15] [16].

Various are the motivations for this approach, which include simplicity of design, more flexibility and solutions more suitable for the problem it must solve. The used data structures are different from those used in relational databases, making some operations faster in NoSQL[17].

Depending on the used database, the benefits can be variable. There are those that focus on being able to scale well and others that aim for data consistency.

We can primarily categorise them into four different categories:

- *Key-Value stores*
- *Column-Family stores*
- *Document stores*
- *Graph databases*

[14] http://nosql-database.org/
[15] https://en.wikipedia.org/wiki/NoSQL
[16] http://nosql-database.org/
[17] An example in the next chapter about Graph Databases

In **Key-Values stores** (*Figure 2.1.5*), created to be always available and support extreme loads, keys and values are aligned with an inherently schema-less data model.

The data is stored in an item that contains a key along with the actual data. It is therefore quite similar to a *Hash Table*[18].

This method is the simplest to implement, but also the most ineffective if most operations concern only part of an element.

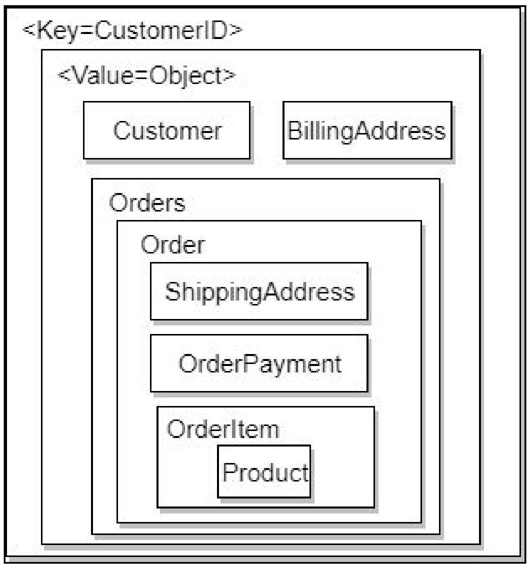


Figure 2.1.5: Key-Values stores

With a Key-Value store, the access of the value is possible only by lookup based on its key. The DBMS does not know the content of the value, which is considered just a big blob of mostly meaningless bits. The advantage of the opacity of the value is that we can store whatever we like it. The database may impose some general size limit, but other than that we have complete freedom. [20]

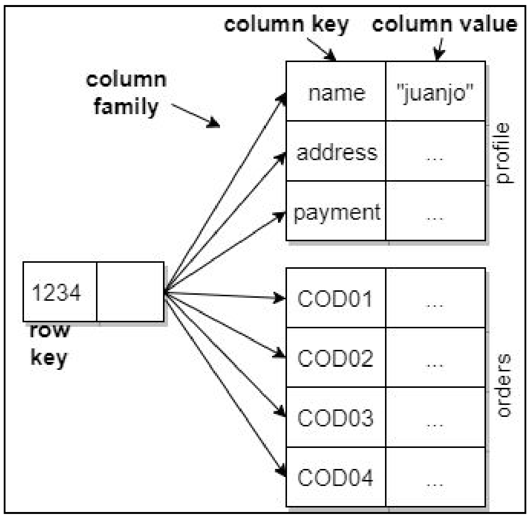


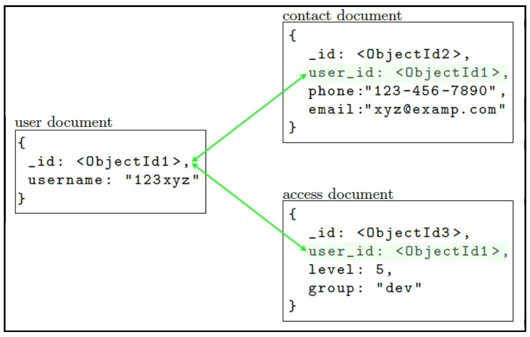
Figure 2.1.6: Column-Family stores

In **Column-Family stores** (*Figure 2.1.6*), the information is stored in columns that usually are not immediately defined. That is another example of a very task-oriented solution type.

The data model includes the concept of a very wide, sparsely populated table structure that includes a number of families of columns that specify the keys for this particular table structure.

[18] https://en.wikipedia.org/wiki/Hash_table

Document stores (*Figure 2.1.7*) are the evolution of the key-value method. Compared to the classic relational databases, rather than storing data in tables with fixed fields, they put information into a document, with unlimited fields of unlimited length, represented in XML[19], JSON[20], BSON[21] or whatever kind of document we want.



```
{
  _id: <ObjectId2>,
  user_id: <ObjectId1>,
  phone:"123-456-7890",
  email:"xyz@examp.com"
}
```

```
{
  _id: <ObjectId1>,
  username: "123xyz"
}
```

```
{
  _id: <ObjectId3>,
  user_id: <ObjectId1>,
  level: 5,
  group: "dev"
}
```

Figure 2.1.7: Document store

In contrast to the Key-Values stores, a Document store can see the structure in the aggregate, imposing limits on what we can place in it, defining allowable structures and types. In return, more flexibility in access is offered: it is possible to submit queries to the database based on the fields in the aggregate, we can retrieve part of the aggregate rather than the whole thing, and the database can create indexes based on the contents. [20]

Last but not least, **Graph Databases** that will be described further besides this dissertation with an emphasis on *Neo4j*, which is one of the oldest and most used in its category[22].

[19] https://www.w3.org/XML/
[20] https://www.json.org/
[21] http://bsonspec.org/
[22] https://neo4j.com/news/neo4j-named-most-popular-graph-database-forrester-research/

2.2) Graphs - Overview

Graphs were first described in an academic paper, on the **Seven Bridges of Königsberg** in 1736, by the mathematician **Leonhard Euler**[23] *(Figure 2.2.1)*. [6] [14]

He was trying to solve a problem that we know as the **seven bridges of Königsberg**[24], a beautiful medieval city in the Prussian empire, in today's Russia, situated on the river Pregel.

This long river runs through the city, create an island in the middle of it, knows as the *Kneiphof*, and simultaneously cut Königsberg into four parts.

By the time of the article, the four parts of the city, which we will refer to them as **A**, **B**, **C** and **D**, were connected by seven bridges, labelled **1**, **2**, **3**, **4**, **5**, **6** and **7** in the picture below (*Figure 2.2.2*).

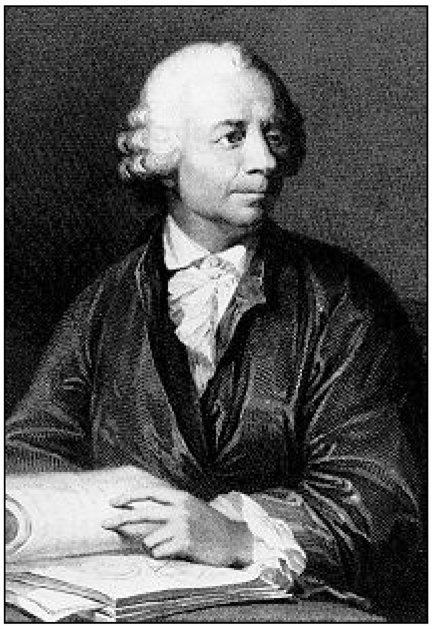


Figure 2.2.1: Leonhard Euler

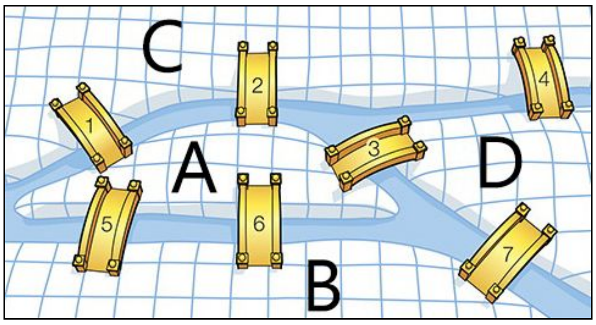


Figure 2.2.2: The seven bridges of Königsberg

The nature of the problem was to *take a tour of the city*, visiting *every one of its parts* and *crossing every bridge*, without having to walk a single bridge or street twice. It was mostly a **pathfinding problem**[25], which can be solved taking the traditional *brute force*[26] method or, like Euler did, applying a mathematical algorithm.

[23] https://www.biography.com/people/leonhard-euler-21342391
[24] https://g.co/kgs/vyhjDH
[25] https://en.wikipedia.org/wiki/Pathfinding
[26] http://intelligence.worldofcomputing.net/ai-search/brute-force-search.html

Euler assumed that the only things that mattered for his pathfinding problem were the parts of the city and the bridges, which connect them.

With this theory in his head, he drew "the world's first graph" (*Figure 2.2.3*):

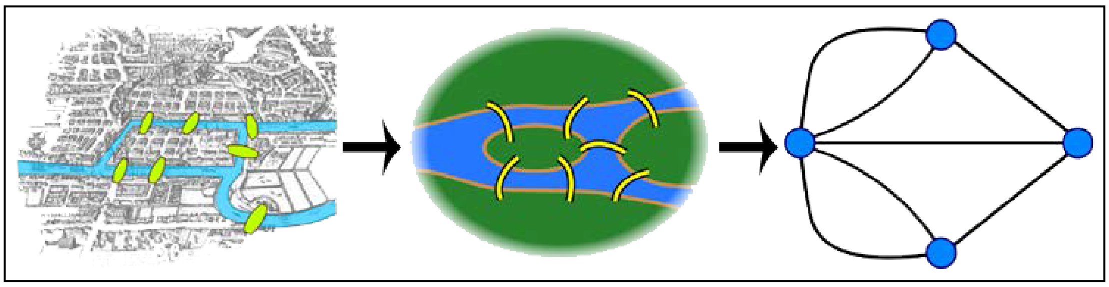


Figure 2.2.3: The world's first graph

Starting somewhere in any one of the four parts of the city, the walker had to leave that part crossing one of the bridges to go to another part. Then, he will have to cross another five bridges, leaving and entering different parts of the city, and finally, he will end the walk in another part of the city.

To find that path, Euler proved that all he had to do was to apply an algorithm that establishes the degree of each part of the city.

In other words, the **degree** was the number of bridges connecting the different parts of Königsberg (*Figure 2.2.4*).

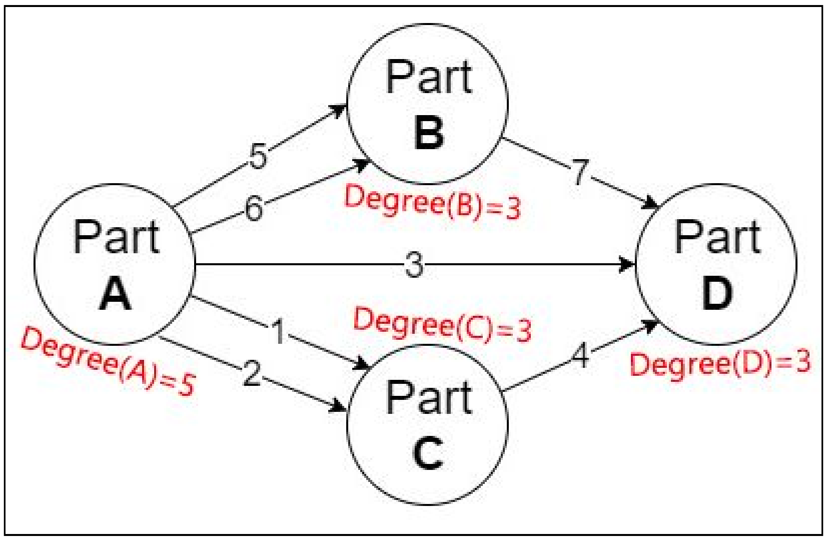


Figure 2.2.4: Königsberg Graph with degrees

Therefore, Euler explained two things about the degree of every part of the city:

The first and last parts could have an odd number of bridges because the walker leaves from the first part and then will arrive at the last part, but the other two parts must have an even number of bridges because he will arrive and leave from these parts of the city.

By proving that no part of the city had an even number of bridges, he also provided that the required walk cannot be done.

Using this example about the **Eulerian Walk** [15], we can confirm that the concepts and techniques of his research are universally applicable. In order to do a *walk on any graph*, the graph must have zero or two vertices with an odd degree, and all intermediate vertices must have an even degree.

However, what is a graph?

A **graph** (*Figure 2.2.5*) is an abstract, mathematical representation of two or more entities connected or related to each other. Entities are represented as *vertices* and how those entities relate to the world as *edges*. [16]

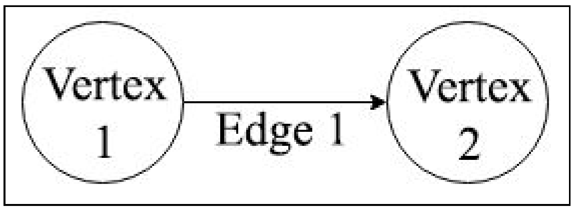


Figure 2.2.5: Two vertices connected by an edge

Graphs are extremely useful in understanding and allow us to model all kinds of scenarios.

For example, consider the image below (*Figure 2.2.6*), where are represented two people connected by the edge "friend_of".

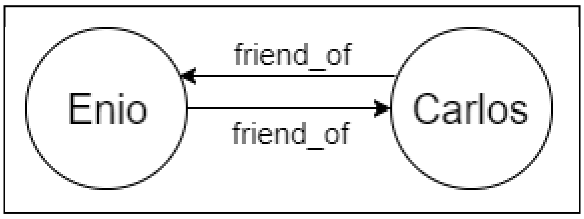


Figure 2.2.6: Two nodes connected by the relationship "friend_of"

We can merely say that these people are friends only reading the label associated with the edge.

Graphs allow us to represent mostly all kind of situations and, they can be easily enlarged by putting other relationships entities and *labels* to indicate its role in the graph (*Figure 2.2.7*).

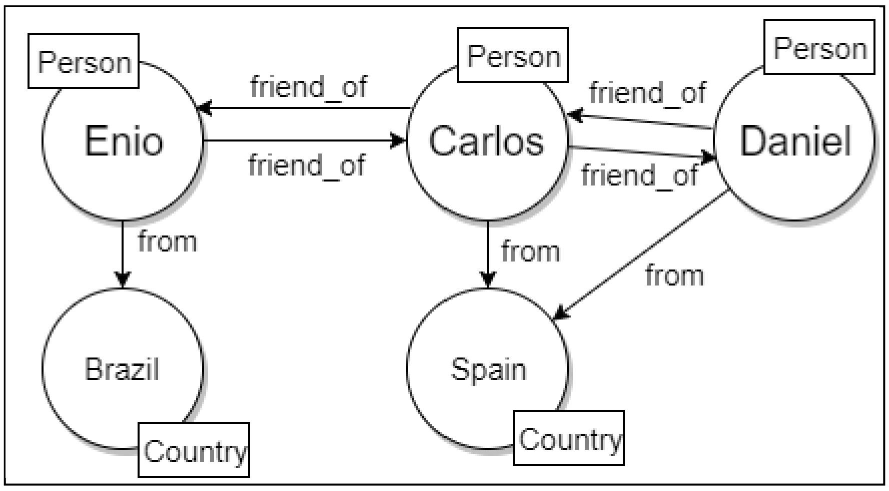


Figure 2.2.7: Graph with five entities and seven relationships

A graph like the one in the image above can be called *labelled property graph*[27] and it has the following characteristics: [1]

- It contains vertices (also called *nodes*) and edges (or *relationships*).
- Vertices and edges can contain properties (key-value pairs).
- Vertices can be labelled with one or more labels.
- Edges have a name and are directed from a start node to an end node.

All technologies used primarily for transactional online graph persistence, accessed directly in real time from an application, are called **Graph Databases**[28], which is the main topic of this dissertation.

[27] https://goo.gl/N4qJaT
[28] https://neo4j.com/developer/graph-database/

Chapter III

3) Graph Databases

3.1) Graph Data Model

A graph structure uses *nodes (vertices)* and *relationships (edges)* to store data persistently, allowing to represent it without the distortions of the relational data model, and, most important, the possibility to apply various types of graph algorithms on these structures. [1] [5] [6]

Graph data model has no fixed schema imposed from the database; therefore it is considered a good fit for dealing with semi-structured data: when nodes or relationships have fewer or more properties, it is not necessary to alter the design.
The most interesting aspects of relationships are:

- Relationships are *directed*, from a start node to an end node.
  They cannot be dangling but can be self-referencing so in this case the start- and endpoint is the same node.
- Relationships are *explicit* because they cannot be inferred or established at query time through a join operation.
- Relationships are *first-class citizen*[29] of the model; they have the same expressive power as the nodes representing the entities in the database. [1]
- Relationships can have properties like nodes, which are values associated with them that can specify some characteristics of that relationship.

In Neo4j the graph data model has been enriched with other concepts:

- *Node labels*: a way to categorise the nodes in a graph. They can have zero or an unlimited number of labels assigned and allows to create subgraphs or some schema in the database easily.
- *Relationship types*: similar to node labels, they categorise relationships.
  Every relationship must have only one type and are used during traversals across the graph.

[29] https://en.wikipedia.org/wiki/First-class_citizen

3.2) Graph Databases

Graph databases, using graph theory as a basis, store data in the form of nodes, relationships and properties. That helps build a graph of data, optimised for transactional performance, which is related directly to the data.

An example of a graph database in the "real world" could be a crime diagram for a TV show *(Figure 3.1)*, where people are related to each other. [5]

1998
The Central City Tribune
DOCTOR ACCUSED OF KILLING WIFE
NORA ALLEN DIES
NORA ALLEN MURDERED BY HUSBAND

Figure 3.1: Crime Board

The power of graph databases can be resumed in one single sentence:

"*Graph databases offer a remarkable performant and flexible data model, aligned with today's agile software delivery practices*[30]*"*.

In contrast to relational databases, where large dataset deteriorates performance, with graph databases, performance tends to remain almost constant, because there are not join operations, and queries are located to a portion of the graph.[31]

About flexibility, graphs are naturally additive, so, add new kinds of elements without disturbing the application functionality is easy. This idea follows perfectly today's incremental and iterative software delivery practices, called Agile Methodologies, because it allows evolving our data model step by step.

Some Graph Databases use native graph storage, like Neo4j, and others serialise the graph data into a different data store, like for example an object-oriented database. Both types present their benefits, but performance and scalability are crucial qualities offered almost only in a native graph storage database.

[30] https://www.versionone.com/agile-101/agile-methodologies/

[31] An example later in the chapter.

A crucial key to the success of graph structures is the capability to ***traverse a graph***[32], without performing an index lookup. This capability is also known as **index-free adjacency**[33], which provides a fast walk on the graph's nodes and relationships, enabling to hop from one node to the next by following the explicit pointers that connect the nodes.

3.2.1) Relational Databases vs. Graph Databases: A Comparison example

Social networks[34] might be the perfect examples to show the real power of this technology. As we can see in the image below *(Figure 3.2.1)*, represented as a graph data structure, user connected with arrows are friends. [2]

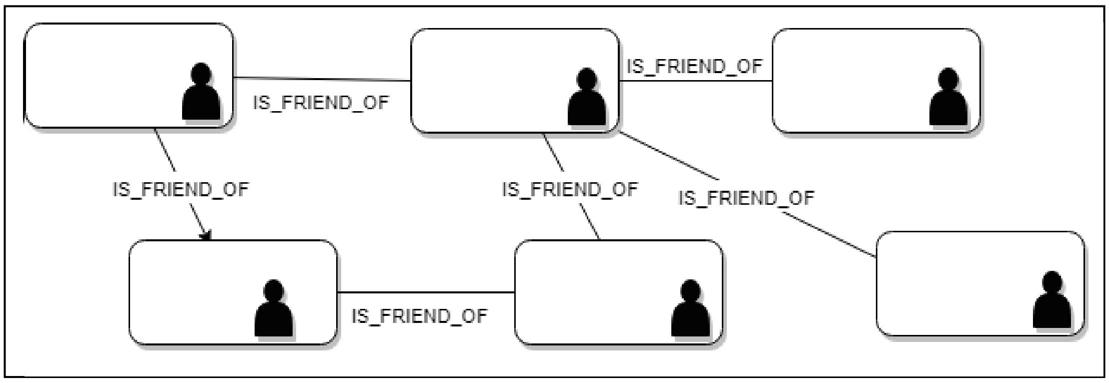


Figure 3.2.1: Hypothetical Social Network

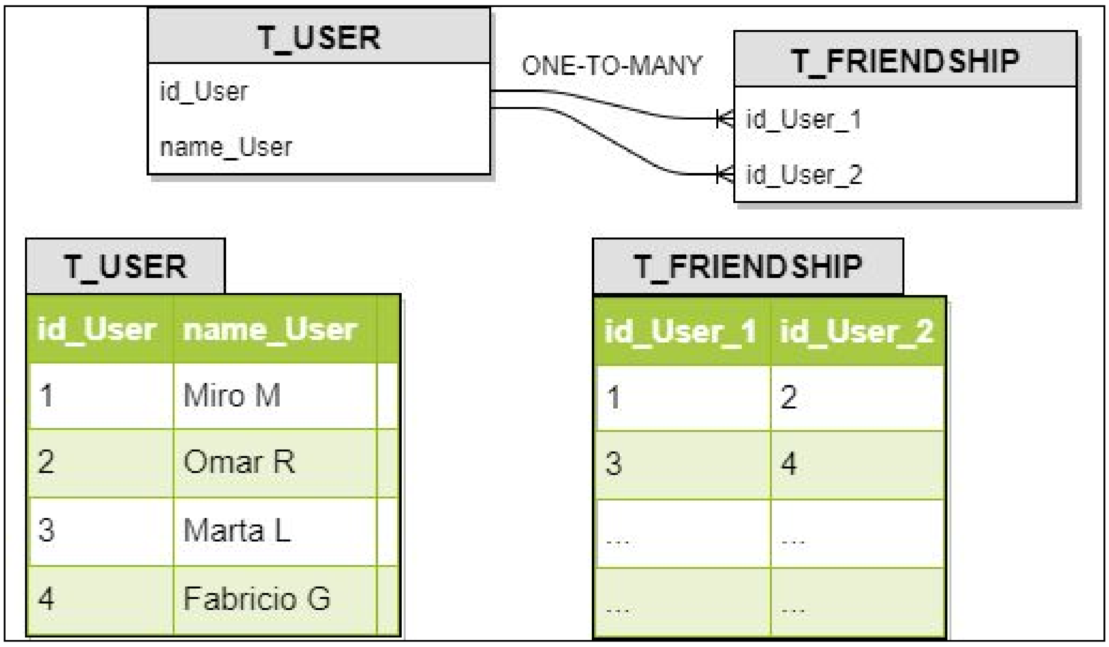
Analysing the high-abstraction level image above, it is possible to define and create both type of databases, a relational and a graph database. There are different ways to do that, but in this example is used the simplest one.

The relationships *IS_FRIEND_OF* is symmetric[35], and this means that is A is a friend of B, B is a friend of A as well.

3.2.1.1) Relational version

In a relational database, this social network would have two relational tables: one for the user information and the other for friendships *(Figure 3.2.2)*.

[32] http://btechsmartclass.com/DS/U3_T10.html
[33] https://en.wikipedia.org/wiki/Adjacency_list
[34] http://whatis.techtarget.com/definition/social-networking
[35] https://en.wikipedia.org/wiki/Symmetric_relation



T_USER

id_User	name_User
1	Miro M
2	Omar R
3	Marta L
4	Fabricio G

T_FRIENDSHIP

id_User_1	id_User_2
1	2
3	4
...	...
...	...

Figure 3.2.2: SQL Diagram of tables representing our Social Network

Analyzing some possible queries, to extract information about friends, we can easily find problems which can affect performance. Popular social networks, like Facebook[36] or LinkedIn[37], have features which suggest potential friends from our friendship network, up to a certain depth.

This kind of actions needs *join* operations for each needed level of depth. For example, to find friends of friends of friends of a user, our query would be:

```
select count(distinct *) from t_friendship tf1
inner join t_friendship tf2 on tf1.id_User_1 = tf2.id_User_2
inner join t_friendship tf3 on tf2.id_User_1 = tf3.id_User_2
where tf1.id_User_1  = *ID_USER*
```

Similarly, to iterate through the fifth level of friendship, we would need five joins in the query. In this case, a relational database engine needs to generate the **Cartesian product**[38] of the table *t_friendship* five times and then discard more than this product, to return only the records that we are interested in.

[36] https://www.facebook.com/
[37] https://www.linkedin.com/
[38] http://mathworld.wolfram.com/CartesianProduct.html

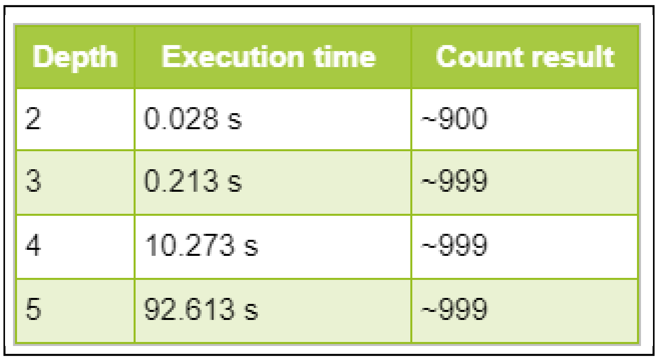
On a small dataset, this would not be a big concern, but with high numbers of users and relationships, this will affect performance significantly.

To demonstrate the performance of our hypothetical social network in a relational database, we can run some tests on a small dataset, increasing the depth of the search each time.

The preliminary test conditions are *1,000 users*, where each one has on average *50 friends*, so the table *t_friendship* has almost *50,000 records*. The use of the *cache* [39] is allowed, and the relevant columns have *indexes*[40] to maximise the performance of these join queries. The hardware configuration is not relevant to be specified.

After the execution of our tests, choosing the fastest execution time for each degree of separation, the results are impressive (*Figure 3.2.3*). The relational database can handle queries to depth two and three quite well, with fast response, but at depths four and five, there is a significant degradation of performance, although the count result does not change.

The number of the results is always of 999 users, at depth three, four and five, because the dataset is too small, in consequence, at this level of depth each user has a relationship with the others.



Depth	Execution time	Count result
2	0.028 s	~900
3	0.213 s	~999
4	10.273 s	~999
5	92.613 s	~999

Figure 3.2.3: Execution times for multiple join queries [Relational Database]

[39] http://searchstorage.techtarget.com/definition/cache-memory
[40] https://www.essentialsql.com/what-is-a-database-index/

The tremendous time response is due to the generation of the Cartesian product of the t_friendship table. With 50,000 rows, the resulting set has *$50{,}000^5$* records, which takes too much time and computing power to calculate it. Comparing the enormous number of records of the cartesian product and the count result of our queries, we can see that more than *99%* of records are discarded.

3.2.1.2) Graph version

From a "*graph point of view*", to implement this example, people will be represented as nodes, and every arrow as a relationship between each node, named "is_friend_of".

As mentioned previously, in a property graph every relationship must have a direction, thus to express the friendship between two people, two are the strategies that can be used: the first is using two arrows between both elements; the other, usually the most used, implies the use of only one relationship, where the direction is not considered.

For simplicity in this example, it will consider only one direction, because the relationships is symmetric and as it will specify later in this work, the relationship direction does not affect performance or stability.

The potential of graph databases is the powerful and efficient engine for querying data, mentioned before, called *graph traversal*.

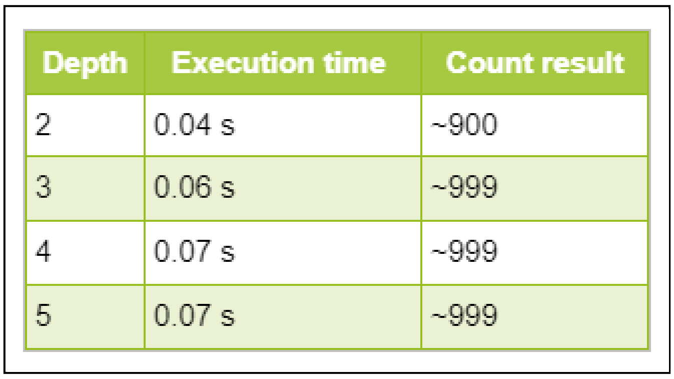
The **traversal** is the method of visiting nodes in a graph by moving between nodes connected with relationships. The traversal is localised only on the required data, without performing expensive grouping operations on the entire dataset, like relational database systems.

Most important characteristic is that the *relationship direction* does not affect the traversal, then, it is possible to move in every direction with the same efficiency.

To start the traversal, the first step is the selection of the starting node, then, the engine will follow all the friendship relationships and collect the visited nodes.

When the rules stop applying, the traversal stops. We refer to these types of queries as *pattern matching*[41] *queries* because we specify a pattern, we anchor that pattern to one or more starting points, and then the engine starts looking for matching occurrences of that pattern, ignoring non-matching patterns.

Under the same condition as the MySQL example, with the same queries as before, the image below *(Figure 3.2.4)* shows the execution times using a Graph database on our dataset.



Depth	Execution time	Count result
2	0.04 s	~900
3	0.06 s	~999
4	0.07 s	~999
5	0.07 s	~999

Figure 3.2.4: Execution times for graph traversals

Graph databases improve performance significantly, except for the first query, where the execution time is almost the same. The considerable difference of time is the key to the success of this approach on this kind of problems. During those executions, the engine does not create any Cartesian product of our records, but Neo4j, for example, merely visits *relevant* nodes in the database, keeping track of the ones visited, to skip them if already visited. The traversal stops when there are no more relevant nodes to visit.

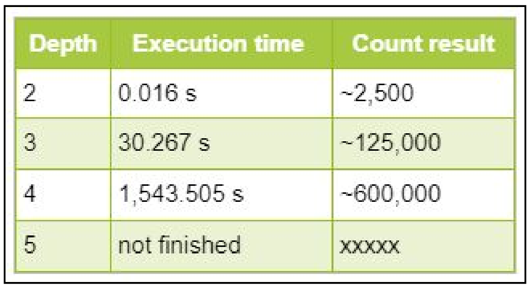
It is easy to understand that query performance is independent of the dataset size but is connected with the size of the result set.

[41] https://en.wikipedia.org/wiki/Pattern_matching

3.2.1.3) Extension of the example

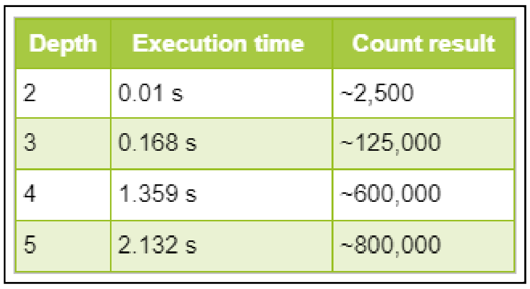
However, this example with only 1,000 users is insufficient. To conduct an effective example, the best decision is to increase the dataset by a thousand times than before.

After running those tests *(Image 3.2.5 and Image 3.2.6)*, the increased data does not affect graph databases' performance significantly but demonstrate worse results on a MySQL database.



Depth	Execution time	Count result
2	0.016 s	~2,500
3	30.267 s	~125,000
4	1,543.505 s	~600,000
5	not finished	xxxxx

Figure 3.2.5: Execution times for multiple join queries (1 million users)



Depth	Execution time	Count result
2	0.01 s	~2,500
3	0.168 s	~125,000
4	1.359 s	~600,000
5	2.132 s	~800,000

Figure 3.2.6: Execution times for graph traversals (1 million users)

In the relational example, queries at depth three and four have colossal execution time, and at depth five, the query cannot compute anything even after one hour of execution.

These results show that relational databases are optimised for single join queries, even on large datasets.

Social Networks are just an example, but there are a lot more, like music recommendation[42], bioinformatics[43], network sensors[44] and so on, where graph databases provide powerful tools to manage our data.

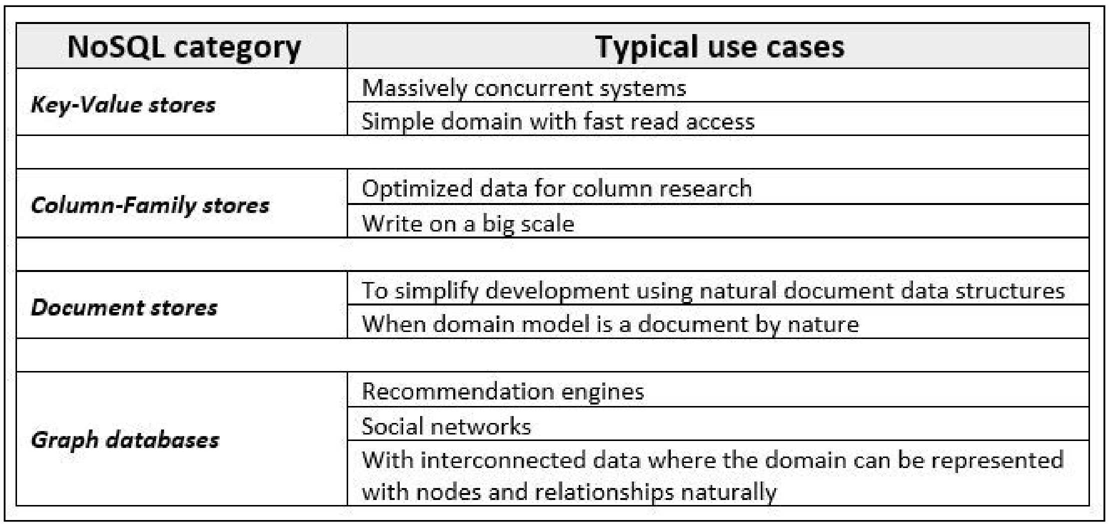
[42] https://www.rtinsights.com/music-recommender-system-musimap-neo4j/
[43] https://goo.gl/C2UHhK
[44] https://arxiv.org/pdf/1708.03878.pdf

3.2.2) Are Graph Databases always the best choice?

Referring to the previous topic concerning NoSQL, those kinds of databases are all task-oriented: they offer the right tool for a specific job *(Figure 3.3)*.
For that reason, there are some problems in which graph databases are not suitable or efficient solutions.

Reusing the Social Network example, if we limited our join operations only on depth two, relational databases became a good choice of implementation. In general, if we are trying to work with extensive lists of things, effectively sets, and not sophisticated join operations are required, the performance of the graph database would not be as good as other databases.

Furthermore, graph databases are great for complex queries. As a consequence, simple queries usually are served quite inefficiently than other kinds of databases. [2]



NoSQL category	Typical use cases
Key-Value stores	Massively concurrent systems
	Simple domain with fast read access
Column-Family stores	Optimized data for column research
	Write on a big scale
Document stores	To simplify development using natural document data structures
	When domain model is a document by nature
Graph databases	Recommendation engines
	Social networks
	With interconnected data where the domain can be represented with nodes and relationships naturally

Figure 3.3: Typical use cases for NoSQL databases

Why choose Neo4j?

Lots are the reasons to use Neo4j to create our projects, but according to the top ten proposed on its official webpage[45], *Neo4j is an extremely reliable product offering scaling capabilities, cluster support, extraordinary availability, high read and write speed, and full ACID (Atomicity, Consistency, Isolation, Durability) compliance.*

Since 2007, its community boasts more than 200 enterprise customers, at least 100 technology and service partners and more than 3,000,000 downloads.

[45] https://neo4j.com/top-ten-reasons/

Chapter IV

4) Neo4j

Neo4j, developed by ***Neo Technology, Inc.***[46] (operating from the San Francisco Bay Area in the U.S.) is an extremely scalable[47] and transactional *ACID* Graph Database Management System, which stores data structured as graphs. [1][2][5]

It is written in *Java*[48] and is *open source*[49].

4.1) Key characteristics - Overview

4.1.1) ACID support

Using the official **ACID** definition from Wikipedia: "ACID (acronym of **Atomicity**, **Consistency**, **Isolation** and **Durability**) *(Figure 4.1.1)* is a collection of properties of database transactions, designed to guarantee validity even in the event of errors." [50] In particular, Neo4j supports: [6] [9]

- *Atomicity*: Multiple database operations can be executed within a single transaction which ensures they are all executed atomically; if one or more operations fail, the entire transaction fails and will be rolled back.
- *Consistency*: Ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules.

[46] http://goo.gl/gKD5Bv
[47] Mark D. Hill. 1990. What is scalability?. SIGARCH Comput. Archit. News 18 (1990)
[48] https://www.ibm.com/developerworks/java/tutorials/j-introtojava1/index.html
[49] St.Laurent, Andrew M; "Understanding Open Source and Free Software Licensing" O'Reilly
[50] Haerder, Reuter; "Principles of transaction-oriented database recovery" ACM C.S.

- *Isolation*: Each transaction should appear as though it is being executed in isolation from other transactions, even though many of them are executing concurrently. For example, in an *isolated database*, as long as the write operation is not committed, the read operation has to work with the old version data. In other words, the transactions need to be executed, irrespective of what is happening in the system at the same time.
- *Durability*: The changes applied to the data by a committed transaction must persist in the database. These changes must not be lost because of any failure

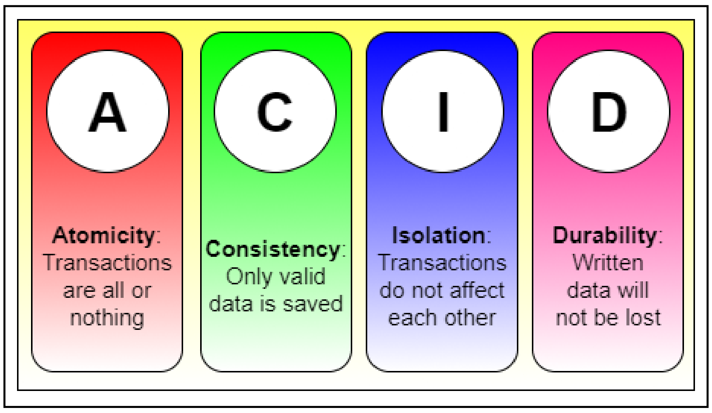

A
C
I
D
Atomicity: Transactions are all or nothing
Consistency: Only valid data is saved
Isolation: Transactions do not affect each other
Durability: Written data will not be lost

Figure 4.1.1: ACID

4.1.2) Transactional support

ACID support is useful when we are working with a database management system (DBMS) in an online system environment, where operations, usually, need to be answered extremely fast. This characteristic is not required of every DBMS because some systems do not require answers in real time because they have analytical purposes.

For example, these systems are called *Analytical Systems*[51]. [2][6]

We can differentiate two types of systems:

[51] http://searchbusinessanalytics.techtarget.com/definition/analytic-database

- **Online Transaction Processing** (**OLTP**[52]), which works on operational data to control and run fundamental business tasks. Usually, the operations are fast, like insert and updates, and need fast answers.
- **Online Analytical Processing** (**OLAP**[53]), which works on consolidated data, usually came from OLTP databases, to help with planning, problem-solving and decision support. Usually, the operations are complicated, and they can need many hours, depending on the amount of data involved.

Neo4j is typically an OLTP system, but at the same time offers support for analytical tasks, but, at the moment[54], it is not optimised for it.

Transactional support differentiates Neo4j from the majority of NoSQL databases and makes it the perfect solution for every kind of environment which takes benefits using a native graph storage.

In Neo4j, transactions are semantically identical to traditional database transactions:

- In order to maintain atomicity and consistency, writes occur within a transaction context, use *write locks*[55] on any nodes and relationships involved in the transaction. On successful conclusion of it, changes are saved to disk for durability, and the write locks are released. In case of a fail transaction, the writes are discarded, and the write locks released, maintaining the graph in its previous consistent state.
- When two or more transactions attempt to change the same elements concurrently, the system will detect this potential *deadlock*[56] situation, and serialise the transactions.
- To maintain isolation, writes within a single transactional context, will not be visible to other transactions, until they are confirmed and the transaction ended.

4.1.3) Scalability support

To support critical scalability, high availability, and fault-tolerance requirements, Neo4j could use two different techniques.

[52] https://en.wikipedia.org/wiki/Online_transaction_processing
[53] https://en.wikipedia.org/wiki/Online_analytical_processing
[54] End of 2017
[55] https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock
[56] Padua, David, "Encyclopedia of Parallel Computing". Springer. 9780387097657 (2012)

The first is the classic **clustering master-slave technique**[57], which creates clusters of database server instances, that work together to achieve these goals. [2]

4.1.3.1) Master-Slave Clustering

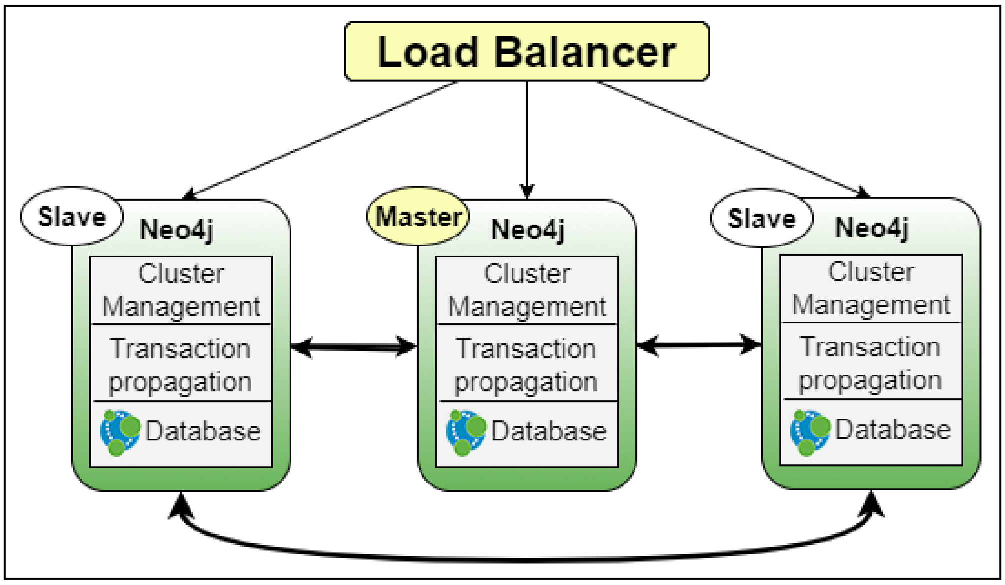


Figure 4.1.2.1: Neo4j high availability architecture

The architecture is composed of a single master instance and an indefinite number of slave instances *(Figure 4.1.2)*.

Most important characteristics are:

- Each server has the entire database and can respond to all query requests.
- The server should be optimised to respond to a particular subset of queries that can receive. For example, using a **load balancer**[58], it is possible to redirect requests to faster servers or, servers with the necessary nodes in the cache memory (**sharded cache**[59]), to improve performances.
- In case of potentially conflicting data in the database, the master server instance decides what to do, choosing the correct data version.

[57] https://neo4j.com/docs/operations-manual/current/clustering/high-availability/architecture/
[58] https://en.wikipedia.org/wiki/Load_balancing_(computing)
[59] http://jimwebber.org/2011/02/scaling-neo4j-with-cache-sharding-and-neo4j-ha/

- Problems with the master instance do not affect stability, because, with a *master election algorithm*[60], the system can choose a new master quickly.

This clustering technique provides two features:

- **Horizontal scalability**: provided by adding more machines to the cluster and distributing the load, with a load balance, over the cluster members.
- **Vertical scalability**: provided by adding more resource to the cluster machines to support write load.

As mentioned before, all servers can respond to all query requests, but writing through slaves is different from writing through the master.

The slaves need the confirmation from the master before returning to the client, so, this causes additional network traffic which can affect performance. For this reason, the recommendation is to direct all write requests directly to the master, but in high write load scenarios, that can be problematic.

To regulate load, writes can be *buffered* using a queue and directed to the master. In this case, writes are executed against the database in groups, and this can regulate the traffic, but, are required protocols which ensure ACID properties.

About reading, in high read load scenarios, using *multi-region cluster in multiple data centres* is an excellent solution to improve response times. Each client's read request can be executed by the cluster geographically closer to it, to reduce latency time.

4.1.3.2) Causal Clustering

Causal Clustering[61], introduced in version 3.2, is the second solution for ensuring redundancy and performance in a high-demand production environment.

This strategy makes use of two different roles[62] (*Figure 4.1.2.2*):

- **Core Servers** (CS): can execute read and write operations and ensure the safeguard of data by replicating all transactions using the Raft protocol.
  This protocol guarantees that data is safely durable before confirming transaction commit to the user, waiting for the commit of the majority of the Core Servers (the minimum number of needed commit is $N/2+1$, where N is the number of CS in a cluster). The first impact is on write latency, which grows with the number of CS in the cluster.

[60] For example: the Paxos algorithm. https://lamport.azurewebsites.net/pubs/paxos-simple.pdf
[61] https://neo4j.com/docs/operations-manual/current/clustering/causal-clustering/
[62] https://goo.gl/fsLoYE

To provide sufficient fault tolerance, the formula *CS = 2F + 1* can be used to identify the number of Core Server required to tolerate *F* faults. In case of multiple write faults, the server will become read-only to preserve safety.

- **Read Replicas** (RR): are responsible for scaling out graph workloads, capable of fulfilling arbitrary read-only queries.
  RR are asynchronously replicated from CS via transaction log shipping. Periodically a Read Replica will poll a Core Server for any new transactions, and the Core Server will ship those transactions to the Read Replica.
  Losing an RR does not impact the cluster's availability and the fault tolerance of the cluster.

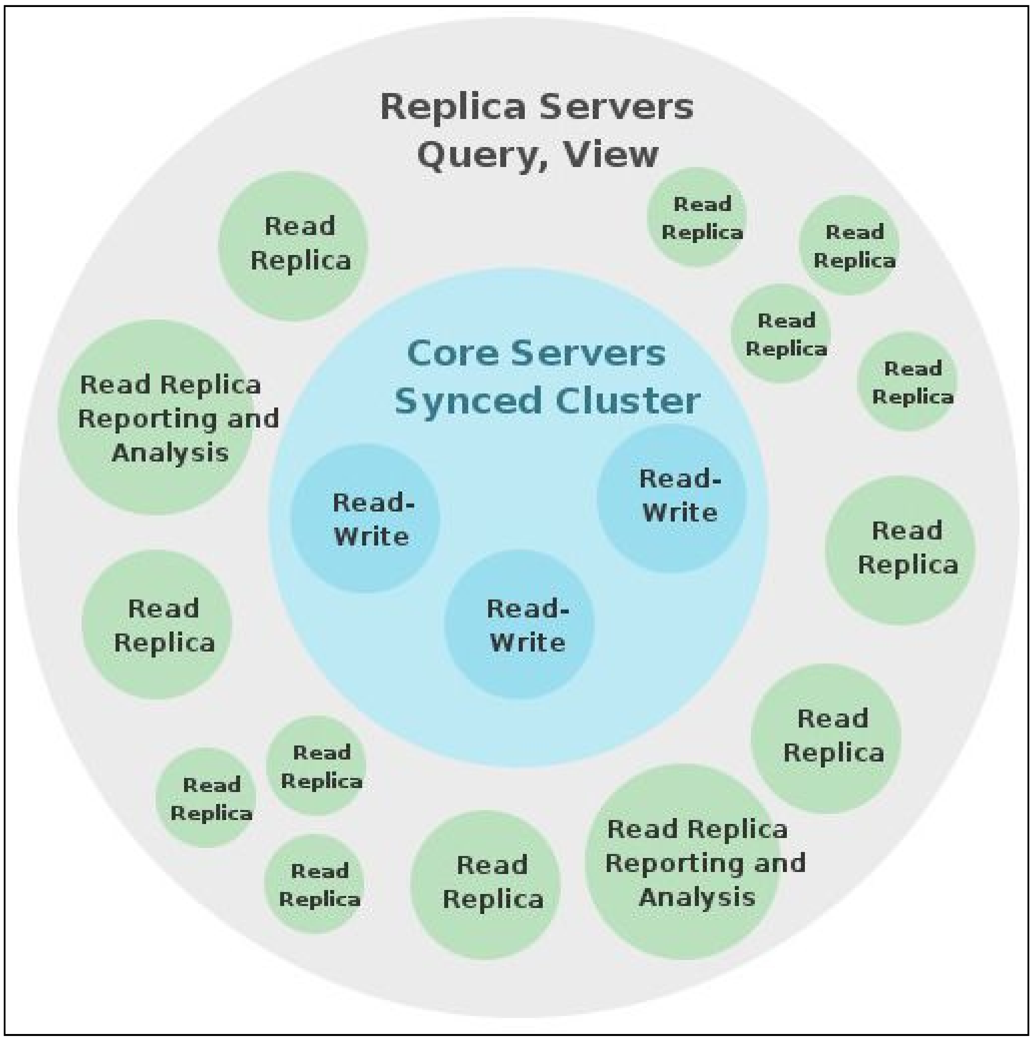


Figure 4.1.2.2: Causal Cluster Architecture

The basic idea of this strategy is the **causal consistency** which ensures that causally related operations are seen by every instance in the system in the same order. Client applications never see stale data and interact with the database as if it was a single server. Consequently, client applications enjoy read-your-own-writes semantics making interaction with even large clusters predictable and straightforward.

Therefore, depending on the nature of the workload, users want reads from the graph to take into account previous writes. Causal consistency makes it easy to write to Core Servers and read those writes from a Read Replica. For example, it is guaranteed that the write which created a user account will be present when the same user subsequently attempts to log in.

On executing a transaction, the client can ask for a bookmark which it then presents as a parameter to subsequent transactions. Using that bookmark the cluster can ensure that only servers which have processed the client's bookmarked transaction will run its next transaction. This provides a *causal chain* which ensures correct read-after-write semantics from the client's point of view.

4.1.4) Neo4j's declarative query language

Neo4j's query language is called **Cypher**[63], designed to be a human query language, simple to understand and use. [2][4][5][9]

It is ***declarative***[64] because Cypher is focused on the aspects of the result, rather than methods or ways to get to it.

It enables to ask to find data that matches a specific pattern, using pattern matching:

- A **pattern** is an occurrence of sequences that need to be found in a given data. They are used to describe the shape of the data and also provide the path from where it should start searching for occurrences of the provided pattern.
- **Pattern matching** is the process used to find a pattern against a data structure. This technique consists of specifying patterns to which some data should conform, and then checking to see if it does. This does not mean that the match has to be always exact because we can consider pattern matching as an extension of pattern recognition.

Cypher will be discussed in detail in the next chapters of this dissertation.

[63] http://neo4j.com/docs/developer-manual/current/cypher/#cypher-intro
[64] Lloyd, J.W., Practical Advantages of Declarative Programming

4.1.5) Pathfinding queries

Neo4j does not depend massively on indexes because the graph itself provides a natural adjacency index. As specified before, the relationships attached to a node, provide a direct connection to other related nodes, allowing a fast traversal efficiently, in contrast to joining data through indexes, which is many orders of magnitude slower.

Another significant characteristic offered in Neo4j, using its powerful traversal framework, is the capability to find out if there are useful paths between different nodes on our graph *(Figure 4.1.3)*. This kind of queries allowed us to: [2]

- See if exists a path between two nodes.
- Look for the optimal path[65].
- Look for the variability of the path, if a particular component of the path changes.

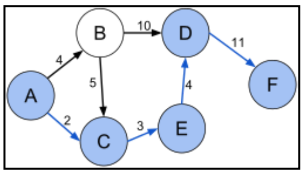


Figure 4.1.3: Shortest path problem

4.1.6) REST API via HTTP

The most common way to communicate with Neo4j is using its ***REST API*** via ***HTTP*** (*Representational state transfer*[66] *Application programming interface*[67] via *Hypertext Transfer Protocol*[68]*)*. It is possible to submit Cypher queries and execute many algorithms offered in Neo4j. [4][6][7]

REST is a *style* adopted when designing network applications, used on different communication protocols, but mostly on HTTP.

It uses four HTTP verbs[69] **GET**, **PUT**, **POST** and **DELETE** to perform specific actions within the application to access the database.

Some examples will be shown in the next chapters.

For almost the majority of use cases, the REST API is sufficient, but there are some cases where are required more complex operations and developing a server extension is considered a better solution.

[65] https://en.wikipedia.org/wiki/Shortest_path_problem
[66] https://neo4j.com/docs/rest-docs/current/
[67] https://en.wikipedia.org/wiki/Application_programming_interface
[68] https://tools.ietf.org/html/rfc2616
[69] https://restfulapi.net/http-methods/

4.1.7) Server extensions

Server extensions are used to execute Java code inside the server, extending or replacing the operations offered by REST API. [1]

To create them is used **Java API for RESTful Web Services (JAX-RS**[70]), a Java programming language employed to create API for web services according to the REST architectural pattern.

Each extension class is annotated[71], to indicate which HTTP requests it handles, the response formats and the formatting of URI templates.

Different are the benefits of server extensions:

- **Transactional**: different and complex operations can be executed in the context of a single transaction.
- **Encapsulation**: the extension is hidden behind a RESTful interface, that allows the possibility to modify the implementation easily.
- **Response formats**: responses can be controlled creating messages easy to understand, without graph-based terminology.

The three most important limitation of server extensions are:

- **Programming language**: they can be created only using JAX-RS because it is a JVM-based language.
- **Garbage Collection**: is required a control on the garbage collection to ensure that our code does not introduce any untoward side effects.
- **Dangerous**: the power of server extensions is the possibility to execute complicated code, but sometimes this is dangerous because can create errors in the graph.

4.1.8) Indexes

A database index used to improve performances, at the cost of additional writes and storage space, is a copy of the information in the database.

In other words, indexes are used to execute our queries on the database, locating data quickly.

They are similar to the indexes used in the relational databases[72].

[70] https://jcp.org/en/jsr/detail?id=339
[71] http://download.oracle.com/javase/1,5.0/docs/guide/language/annotations.html
[72] https://www.essentialsql.com/what-is-a-database-index/

4.1.9) Caching

Neo4j offers two different caching systems[73] [74]: a File Buffer Cache and an Object Cache. They were created to improve performances, but every one of them has different functions: [2]

- The **File Buffer Cache** is the classic cache which stores data contained in the database. Data are stored in the cache memory in the same format that they are on disk, and can be quickly retrieved if the same data is asked again.
  Another function is the combination of lots of small transactions in the cache before the flush to persistent storage, to improve writing operations, which usually are slow.
  There are limits with this cache because typically, the whole database cannot enter into it, for that reason, Neo4j is continuously monitoring the size of the cache, deciding the perfect moment to swap out old data.
- The **Object Cache** allows fast traversal of the graph and is split into two types:
  - **Reference Cache**, which stores nodes and their relationships. During a query execution, if the system has already done a lookup and the data has not changed, the query will hit the cache and give a response immediately.
  - **High-Performance Cache**, stores nodes, relationships and their properties, for quick query executions, but it is only available in the Neo4j Enterprise Edition[75].

4.1.10) Cache sharding

As mentioned before, the cache memory is used to improve query performances, but with massive graphs, it is not possible to fit everything into it. [1]

Partitioning data can solve this problem, using a technique called *cache sharding*, which consists of routing requests to servers which have the portion of the graph required in their cache memory.

In the next figure *(Figure 4.1.4)* is represented a possible system with three servers, connected using a load balancer. When a user executes a query on the database, it is a task of the load balancer to understand to which server redirect the request, depending on the portion of the graph that is interested in the query.

---

[73] https://neo4j.com/docs/operations-manual/current/performance/
[74] Other information inside the "Inside Neo4j" chapter
[75] Covered later in the chapter.

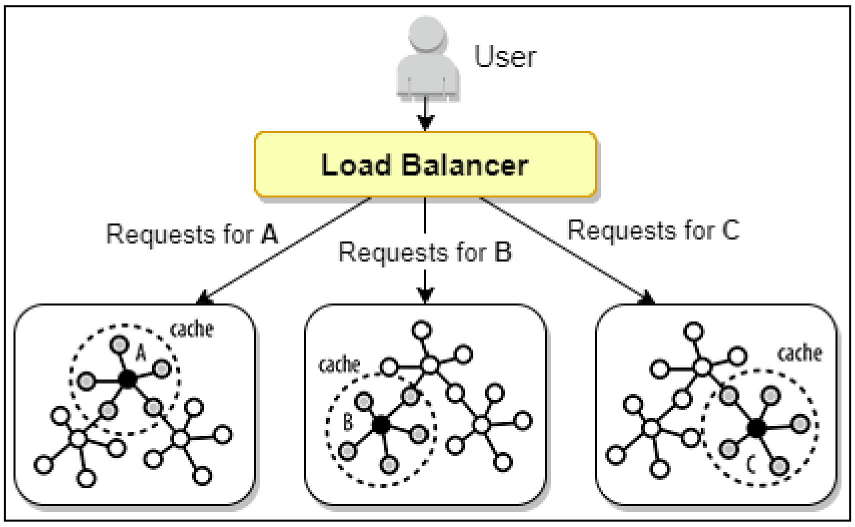
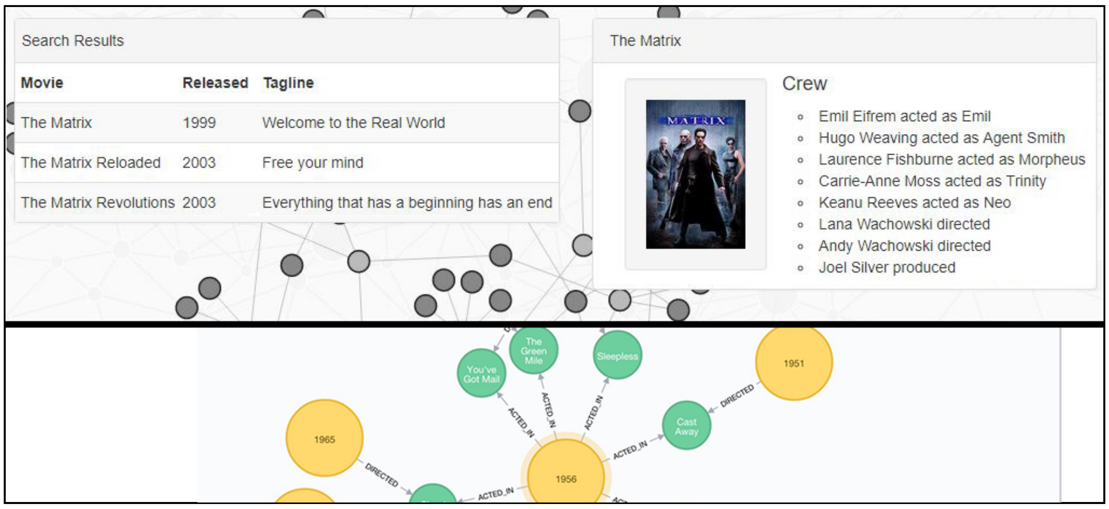


Figure 4.1.4: Cache Sharding

4.1.11) Browser - A visualisation framework

The Browser[76] framework in Neo4j gives the possibility to query the database using Cypher and then see the results like a graph. Working with this visualisation framework offers an instant feedback on the results because allowed us to see the graph generated from the executed query and potentially modify our query to optimise it.

The classic example is the *Neo4j Movies*[77] database, where are listed some movies with all the information about them *(Figure 4.1.5)*.


Search Results

Movie	Released	Tagline
The Matrix	1999	Welcome to the Real World
The Matrix Reloaded	2003	Free your mind
The Matrix Revolutions	2003	Everything that has a beginning has an end

The Matrix

Crew

- Emil Eifrem acted as Emil
- Hugo Weaving acted as Agent Smith
- Laurence Fishburne acted as Morpheus
- Carrie-Anne Moss acted as Trinity
- Keanu Reeves acted as Neo
- Lana Wachowski directed
- Andy Wachowski directed
- Joel Silver produced

Figure 4.1.5: Movies Database example using the visualisation tool

[76] https://neo4j.com/developer/guide-data-visualization/
[77] http://my-neo4j-movies-app.herokuapp.com/

4.1.12) WebAdmin - A monitoring framework

WebAdmin *(Figure 4.1.6)* was an administration tool offered in the oldest versions of Neo4j, where it was possible to view statistics about the database (counts of nodes, properties and relationships, disk usage and so on) or quickly update node values, indexes and more. It was being used to monitor the database, to identify performance problems or other errors. Some of its function are in the Browser visualization framework.

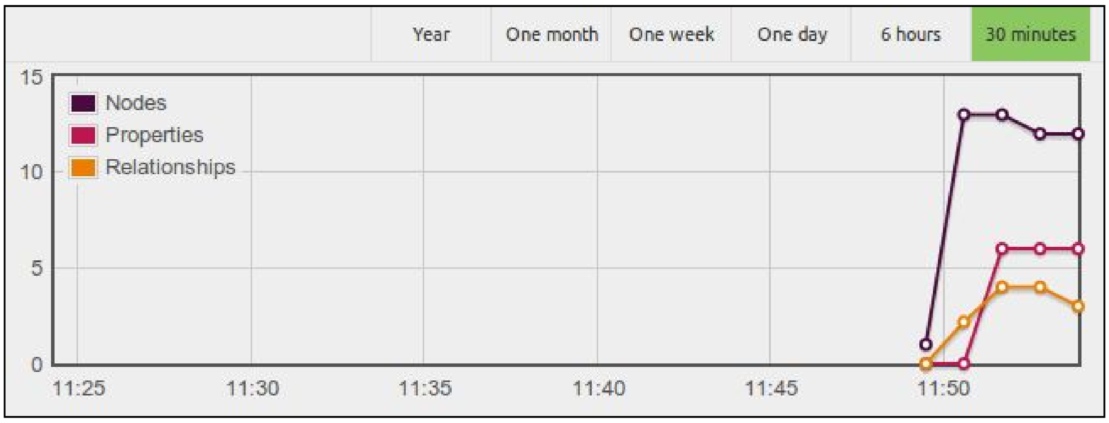


Figure 4.1.6: Neo4j WebAdmin Charts

4.1.13) Open source technology

Neo4j is an open source project *(Figure 4.1.7)*, which means that its source code is readily available with a licence offered by Neo Technology, Inc.

Different and impressive are the relevant aspects of an open source software. Most important are:

- *Lower change of vendor lock-in*: expert users can read and understand the code, in order to fix it, extend it or audit it, independently of the vendor.
- *Better security*: knowing what the system does, should be intrinsically more secure, even though there are many discussions about this topic[78].
- *Lighter support and troubleshooting*: identifying code which generates errors.
- *More innovation through extensibility*: because anyone can extend the software creating new components, called plugins.
- *Cheaper*: usually, users only need to pay if they derive value from the software or when they want premium support.

[78] https://goo.gl/f989h1

```
(Neo4j)-[:LOVES]->(Open Source)
```

Figure 4.1.7: Neo4j Advertising - Classical Cypher query style

Compatible bindings can be written in different languages, including Python[79] and Java.

Neo4j offers two different editions[80] *(Figure 4.1.8)*:

- **Community Edition**: The basic version, but fully functional.
- **Enterprise Edition**: This adds Enterprise functions to the Community Edition, like clustering, advanced monitoring and online backups.

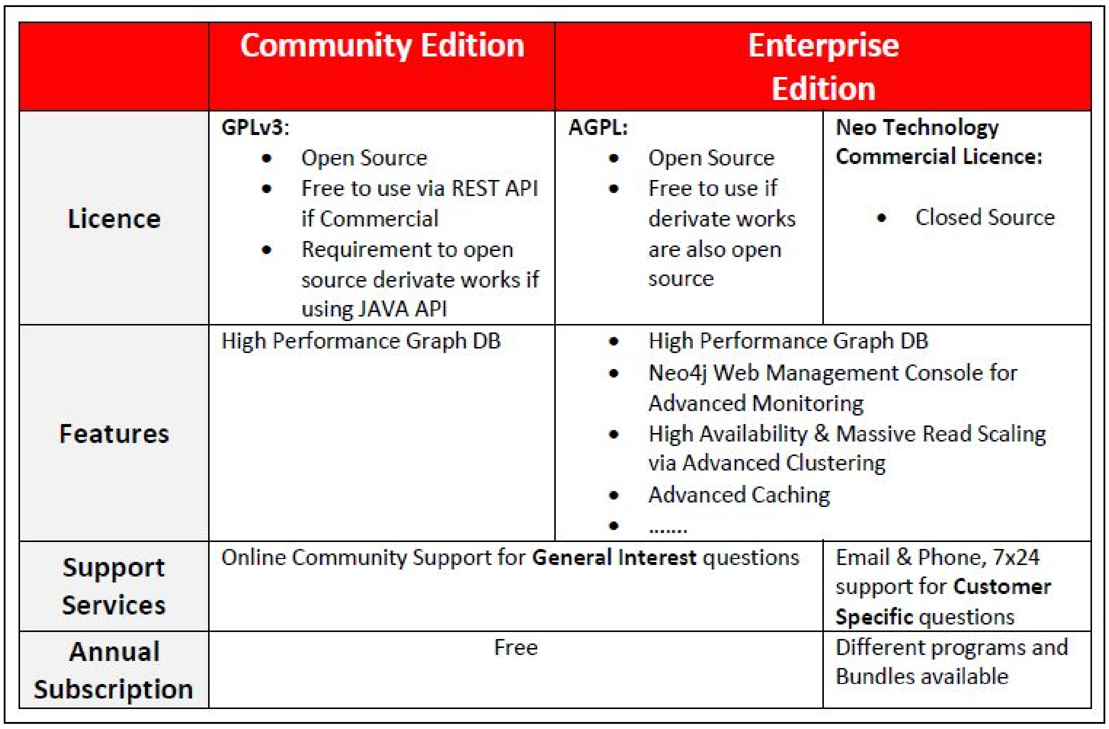
4.1.14) Licence GPL vs. AGPL

To promote both open source software and Neo4j, the owners have chosen specific licensing terms to use their graph database software:

- The Community Edition uses the **GNU Public License version 3** (**GPLv3**[81]) as its licensing terms. It is possible to copy, distribute, and modify the software as long as the programmers keep track of their modifications.
  For commercial applications, the distribution is possible without other requirements, only if the system uses REST API to communicate to the database. If the system uses Java API, the distribution is possible only if the software house provides the source code.
- The Enterprise Edition has two types of licences:
  - The **Affero GNU Public License version 3** (**AGPLv3**[82]), built for network software and it is free only if the system code is open source.
  - The **Neo Technology Commercial Licence** (**NTCL**[83]) where there are no requirements for the source code and customers have dedicated support services.

For *academic purposes*, Neo4j, Inc. offers the opportunity to get an Enterprise Edition with an ***Educational License*** for free, in consequence, each test showed during this dissertation, has been executed on that edition with all Neo4j's features enabled.

[79] https://www.python.org/
[80] https://neo4j.com/editions/
[81] https://www.gnu.org/licenses/gpl-3.0.en.html
[82] http://www.affero.org/oagpl.html
[83] https://neo4j.com/news/neo4j-licensing-guide/



	Community Edition	Enterprise Edition	
Licence	**GPLv3:** • Open Source • Free to use via REST API if Commercial • Requirement to open source derivate works if using JAVA API	**AGPL:** • Open Source • Free to use if derivate works are also open source	**Neo Technology Commercial Licence:** • Closed Source
Features	High Performance Graph DB	• High Performance Graph DB • Neo4j Web Management Console for Advanced Monitoring • High Availability & Massive Read Scaling via Advanced Clustering • Advanced Caching • …….	
Support Services	Online Community Support for **General Interest** questions		Email & Phone, 7x24 support for **Customer Specific** questions
Annual Subscription	Free		Different programs and Bundles available

Figure 4.1.8: Neo4j Editions and Licenses

4.1.15) Backups

Software and Hardware disasters can occur at any time, corrupting data and creating problems to services offered by graph databases.

Neo4j offers two different types of backup[84], offline and, for the *Enterprise edition*, an online version:

- **Offline**: works on any edition of Neo4j, is the simplest backup which involves *downtime* because it has to be performed on a closed database instance.
  Three are the steps to complete this operation:
  a. Shut down the Neo4j instance;
  b. Copy all Neo4j files[85] to a backup location;
  c. Restart the database instance.
- **Online**: it is a robust way to back up a single or clustered Neo4j database without requiring any downtime.
  Two options are available for this type of backup:
  1. **Full backup**: it is a full copy of the entire database, from the beginning of its creation. Usually, it is an extended operation during

[84] https://neo4j.com/docs/operations-manual/current/backup/
[85] Covered in the "Inside Neo4j" chapter

that the database could continue working, in consequence, to ensure that the final backup remains consistent and up to date, Neo4j ensures that any operation occurred during the copy process, will be copied into the backup.

2. **Incremental backup**: does not copy the entire database every time, but keeps track of the last copied operation and continues from that point until the last operation. This operation is more efficient than the full backup because it minimizes the amount of copied data every time.

Neo4j does not provide any backup scheduling functionality, but there are lots of external tools (like *Cron Job*[86]) which can help to schedule backups with the required frequency.

To **restore**[87] a Neo4j database from a backup is required a simple procedure:

a. Shut down the Neo4j instance to be restored;
b. Delete the corrupted or old Neo4j files;
c. Copy the backup files into the folder;
d. Restart the database instance.

Another procedure is the using of the restore command.

```
neo4j-admin restore --from=<backup-directory>
                    [--database=<name>]
                    [--force[=<true|false>]]
```

As it is possible to see, three are the parameters:

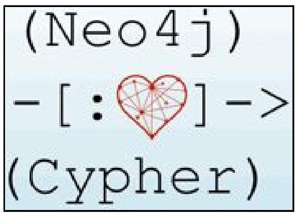
- From: the path where the backup is located;
- Database: (optional) the name of the database to restore;
- Force: (optional) if an existing database should be replaced or not.

Both procedures restore the entire database from the backup.
Partial recovery is, at the moment, not supported.

[86] http://www.oodlestechnologies.com/blogs/Scheduling-tasks-with-Cron-Job
[87] https://goo.gl/JkBiCb

Chapter V

5) Cypher



Cypher, the *Neo4j's declarative language*, uses graph pattern-matching as the primary mechanism for graph data selection[88]. [2][4][5][9]

Its declarative nature means that it is possible to query the graph by describing what is needed to get from it, rather than define methods or ways to get to the results.

Some keywords are similar in functionality to the SQL clauses[89], making it very easy to understand and use by an operations professional.

5.1) Most relevant clauses

The Cypher syntax is case-sensitive, and the two most relevant clauses are:

- **MATCH[90]**: matches graph patterns, allowing to locate the portion of relevant data for our queries.
- **RETURN[91]**: returns the results we are interested in.

In some Cypher manuals is presented the **START**[92] clause, but it was *deprecated* in the latest versions of Neo4j. This clause, employed in combination with explicit indexes, was used to find starting nodes in the graph, where the execution of our queries would start. It was important because avoided to scan the entire graph, improving performance incredibly.

An old simplistic Cypher query makes use of the START clause to anchor to the source, which is succeeded by MATCH clauses used to conditionally traverse through desired nodes, and finally, a RETURN clause that outputs the results.

Today's simplest query only consists of MATCH clauses followed by a RETURN clause.

[88] Covered in section 3.1.4
[89] https://db.apache.org/derby/docs/10.7/ref/rrefclauses.html
[90] https://neo4j.com/docs/developer-manual/current/cypher/clauses/match/
[91] https://neo4j.com/docs/developer-manual/current/cypher/clauses/return/
[92] https://neo4j.com/docs/developer-manual/current/cypher/clauses/start/

To show how those clauses are used, we can consider a hypothetical graph which shows some European cities and flights that connect them *(Figure 5.1)*.

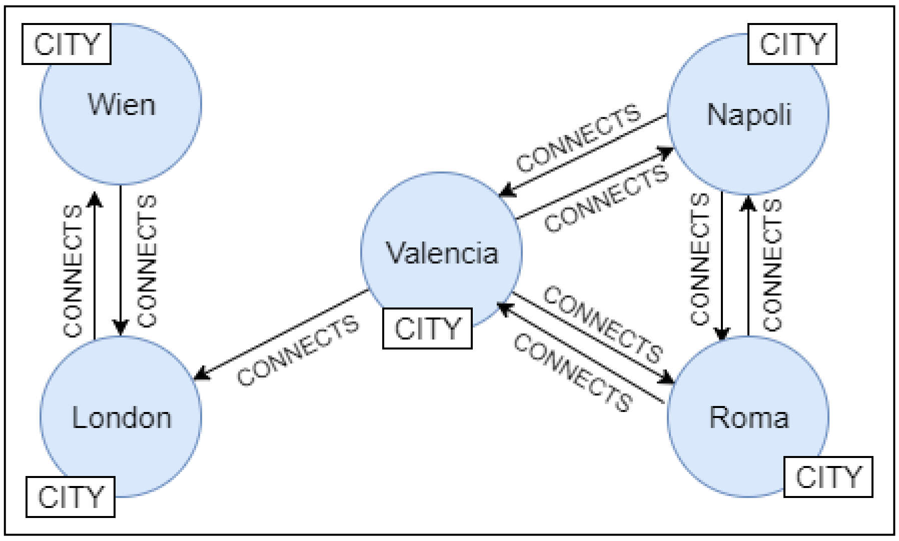


Figure 5.1: Hypothetical Flight Graph

A typical query would be the one that finds a connecting flight path from one city. The START-version query could be:

```
(1) START city1 = node:location(name = 'Valencia')
(2) MATCH (city1) - [:CONNECTS] -> (city2)
              - [:CONNECTS] -> (city3),
(3)       (city1) - [:CONNECTS] -> (city3)
(4) RETURN city2, city3
```

Analyzing the previous query row by row:

1) The START clause indicates the starting point inside the graph, and uses an explicit index called *location* to locate a place stored with the property *name* set to '*Valencia*.'
   This statement returns a reference to a node bound to the identifier *city1*.
2) The MATCH clause, using the identifier *city1*, specifies the pattern that we are looking for in our graph, using the pattern matching technique.
   This clause uses ASCII[93] characters to represent nodes and relationships:
   - Nodes are drowned with parentheses **(**node**)**.
   - Relationships are represented using pairs of dashes with greater-than (**- ->**) or less-than signs (**<- -**).

[93] https://en.wikipedia.org/wiki/ASCII

The relationship directions are expressed by the signs < and >.
Between the dashes, inside the square brackets (**[ ]**) and prefixed by a colon (**:**), there is the relationship name, which connected the two nodes in our query.
A colon can be used similarly to prefix node labels.

3) Another pattern clause of the MATCH clause, separated from the other with a comma (**,**).
4) The RETURN clause used to specify which information, matched during the execution, it will be returned.

Since the pattern in the MATCH clause can occur in many ways, using the START clause was a good practice to anchor the start node and limit the number of useless matched results.

From Cypher 3.2, this clause was removed, and the current recommendation is to use MATCH instead. Using the START clause explicitly will cause the query to fall back to using Cypher 3.1.

```
MATCH (city1:CITY {name = 'Valencia'}) - [:CONNECTS] ->
                (city2) - [:CONNECTS] -> (city3),
      (city1) - [:CONNECTS] -> (city3)
RETURN city2, city3
```

Curly braces (**{ }**) are used to specify node and relationship property key-value pairs.

CITY is the label associated to a node. It is possible for a node to have multiple labels and each one has to be separated with a colon.

During the execution, Neo4j binds the node with property name set to *Valencia* and the label *CITY* into the identifier *city1* and then, by pattern matching, searches nodes and relationships that have the same style.

In the example graph, the only two cities which fit in the specified pattern are *Napoli* and *Roma*, because *city1* is connected with both and they are connected to each other.

An analog way is to express the anchoring as a predicate in the **WHERE**[94] clause:

```
MATCH (city1) - [:CONNECTS] -> (city2) -
                  [:CONNECTS] -> (city3),
      (city1) - [:CONNECTS] -> (city3)
WHERE city1.name = 'Valencia'
RETURN city2, city3
```

WHERE, like the SQL clause[95], it is not mandatory, and it filters out data based on some criteria. For that reason, WHERE is not meant for pattern matching, and using it can cause performance degradation[96].

WHERE can be used to filter based on value ranges using < and >:

```
MATCH (p:Person)
WHERE p.age > 18
RETURN p
```

Other logical operators that can be used in the WHERE clause are: **AND**, **OR**, **NOT**, **EXISTS**, **IS NULL** and so on.

For example, to ensure a property exists on the nodes:

```
MATCH (p:Person)
WHERE EXISTS (p.age)
RETURN p
```

In Cypher, it is also possible to utilise *Regular expressions*[97], declaring a pattern by using ' =~ ' followed by the pattern.

The next query returns all nodes with an I at the first letter of the property called name.

[94] https://neo4j.com/docs/developer-manual/current/cypher/clauses/where/
[95] https://www.w3schools.com/sql/sql_where.asp
[96] Covered in section 5.4
[97] http://www.regular-expressions.info/tutorial.html

```
MATCH (p:Person)
WHERE p.name =~ 'I.*'
RETURN p
```

In the previous examples, we used **identification variables** (like *city1* or *p*) for nodes, to refer to them during the query, but in some cases, this can be omitted, if are not necessary. This identification can also be assigned to relationships.

Like in other languages, there are some rules about identifiers. They are case sensitive, can contain underscores and alphanumeric characters, but the first letter cannot be a number. It is also possible to use spaces in the identifier, but in this case, all the name has to be contained into back quotes.

Choosing good names can improve the readability of the query.

About **properties**, they can be related to nodes or relationships, and a good practice is to choose an explicit name that follows the same rules of nodes/relationships names.

In the Cypher queries, each property identifier and value pair has to be separated with a comma.

Many different value types can be used to save information in properties:

- **Array**: a collection of similar objects (like strings, integers) used to save information. An array of different types is not supported.
- **Boolean**: used to store boolean variables with only true or false value.
- **Numerical values**: to store any numbers.
- **String**: the most common way to save information, using alphanumeric characters.

About **relationships**, to restrict our pattern, we can use a particular syntax where is specified the maximum number of away relationships from our nodes. For example *[:CONNECTS*1..2]* means that we are looking for nodes placed no more than two CONNECTS relationships away from the first node.

Neo4j's official WebPage offers the possibility to run its DBMS without installation, to execute all Cypher queries on our data. This framework, reaching to this link (https://neo4j.com/sandbox-v2/), offers guides and allows to start building applications backed by Neo4j.

All the examples in this dissertation had been executed by this tool.

5.2) Other clauses

Other useful clauses which could be used to create complex queries are:

- **CREATE**[98]: grants the possibility to define new node or relationship. It can be used with the keyword **UNIQUE**[99] to avoid the creation of duplicate entities.

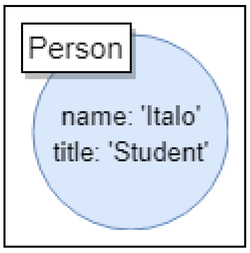


Figure 5.2: Node with the label Person and two properties

Create a new unique node *(Fig.5.2)*:

```
CREATE (:Person
     { name: 'Italo',
       title: 'Student'
     })
```

Create a new relationship between two nodes *(Figure 5.3)*:

```
MATCH (a:Person {name:'Italo'}), (b:Person {name:'Frank'})
CREATE (a) - [:IS_FRIEND_OF] -> (b)
```

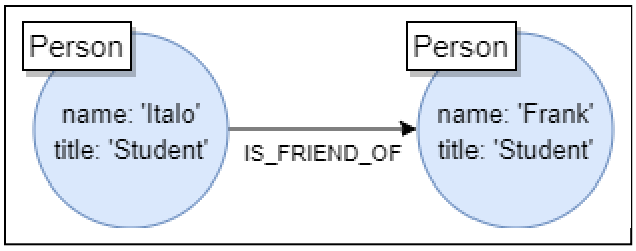


Figure 5.3: Two nodes connected with a relationship

- **SET**[100]: used to assign values to properties of nodes/relationships *(Fig. 5.4)*.

```
MATCH (p { name: 'Italo' })
SET p.age = 29
```

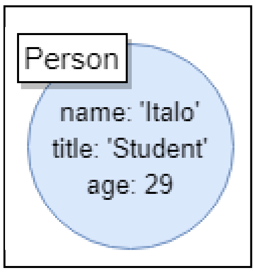


Figure 5.4: The same node of before with a new property

[98] https://neo4j.com/docs/developer-manual/current/cypher/clauses/create/
[99] http://neo4j.com/docs/developer-manual/current/cypher/clauses/create-unique/
[100] http://neo4j.com/docs/developer-manual/current/cypher/clauses/set/

In this example, if the node does not have the property *age*, it will be added. If the property in the specified node already exists, the value will be updated.

To add new label to a specified element *(Figure 5.5)*:

```
MATCH (p:Person {name : 'Italo'})
SET p:Italian
```

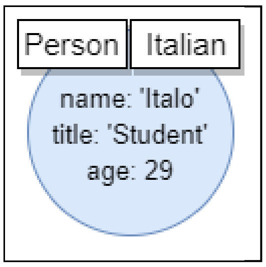


Figure 5.5: Node with two labels

- **DELETE**[101]: deletes nodes or relationships in the graph.

  The classic query to remove a node is:

```
MATCH (p { name: 'Italo' })
DELETE p
```

To delete a node with all its relationships going to or from it, we have to use another keyword: **DETACH**.

```
MATCH (p { name: 'Italo' })
DETACH DELETE p
```

To delete relationships only, considering a relationship called *KNOWS*:

```
MATCH (p { name: 'Italo' })-[r:KNOWS]->()
DELETE r
```

Using DETACH DELETE and removing the parameters in the MATCH clause, allows to delete all nodes and relationships in our graph:

```
MATCH (p)
DETACH DELETE p
```

[101] https://neo4j.com/docs/developer-manual/current/cypher/clauses/delete/

- **REMOVE**[102]: used to remove properties and labels from graph elements.

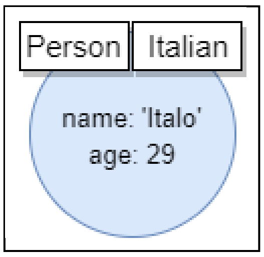


Figure 5.6: The same node of before without the *title* property

Remove property *(Figure 5.6):*

```
MATCH (p { name: 'Italo' })
REMOVE p.title
```

To remove labels *(Figure 5.7)*:

```
MATCH (p { name: 'Italo' })
REMOVE p:Italian
```

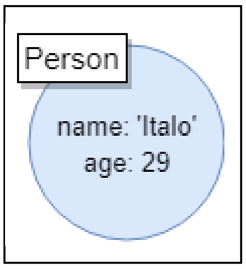


Figure 5.7: The node without the *Italian label*

- **ORDER BY**[103]**:** allows ordering data by properties. This clause has to be written after the RETURN. By default, the sort order is ascending, but with the keyword **DESC**, the order can be reversed.

```
MATCH (p:Person)
RETURN p
ORDER BY p.age DESC, p.name
```

- **LIMIT**[104]**:** limits the maximum number of elements returned. Without this clause, any applicable element would be returned. The LIMIT clause has to be written after the RETURN.
  To show the youngest ten people in our data:

```
MATCH (p:Person)
RETURN p
ORDER BY p.age
LIMIT 10
```

[102] http://neo4j.com/docs/developer-manual/current/cypher/clauses/remove/
[103] https://neo4j.com/docs/developer-manual/current/cypher/clauses/order-by/
[104] https://neo4j.com/docs/developer-manual/current/cypher/clauses/limit/

- **SKIP**[105]**:** offers an offset function. This clause allows skipping the visualisation of some results.
  Used in combination with LIMIT, can help the creation of pagination.
  For example, to see only the results between the 11° and the 15° row:

```
MATCH (p:Person)
RETURN p
ORDER BY p.name
SKIP 10
LIMIT 5
```

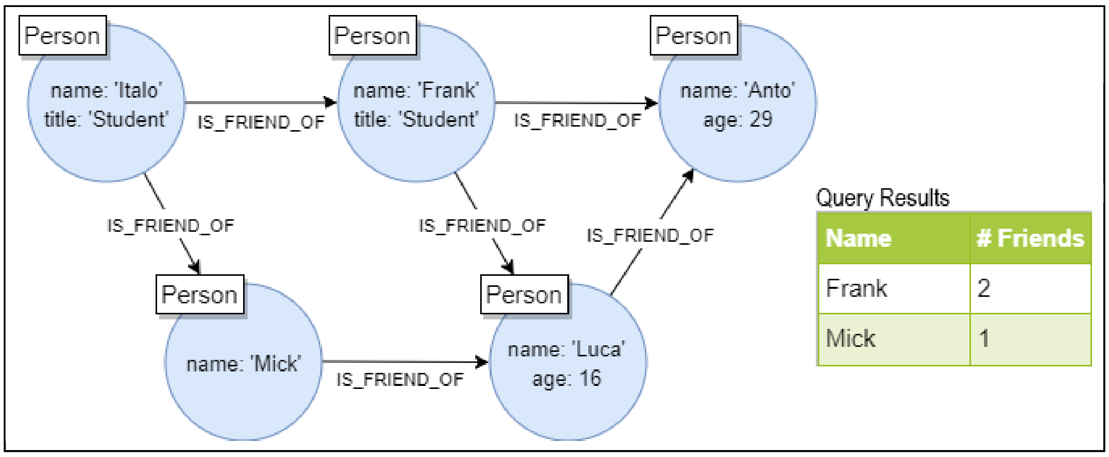
- **WITH**[106]: the primary function is to pipeline the output results of one query into the next, in the form of input. It can be considered as a chaining of queries and helps divide complex queries into several simpler patterns.
  WITH can be used to collect additional data from a query, and another essential function is the possibility to filter results, making queries more efficient by removing unneeded data.

  A possible example to see one of the many functionalities of this clause is:

```
(1) MATCH (a {name:"Italo"}) -[:IS_FRIEND_OF]-> (b)
(2) WITH b
(3) MATCH (b) -[:IS_FRIEND_OF]-> (c)
(4) RETURN b.name, COUNT(c)
```

1. The query matches the node with the property *name* to 'Italo' into the identifier *a* and then, searches his friends, binding them to the variable *b*.
2. This row could be read like "for every friend of the node *a*, execute the other part of the query".
3. For every friend of *a*, search his friends and binding them to *c*.
4. Return a list with the name of the friend of the node *a*, and the number of his friends, using the aggregating function **COUNT**[107] *(Figure 5.8)*.

[105] https://neo4j.com/docs/developer-manual/current/cypher/clauses/skip/
[106] http://neo4j.com/docs/developer-manual/current/cypher/clauses/with/
[107] https://neo4j.com/docs/developer-manual/current/cypher/functions/aggregating/



Name	# Friends
Frank	2
Mick	1

Figure 5.8: A friendship graph and its results of the previous query

- **MERGE**[108]: equivalent to a combination of MATCH and CREATE, ensures that the pattern exists in the graph, either by reusing existing elements that match the supplied predicates, or by creating new nodes and relationships.
  If all the properties in the query do not match a returned node, a new node will be created.
  Used in combination with CONSTRAINTS can help avoid duplicate elements.
  The simplest query is:

```
MERGE (p:Person { name : 'Marco'})
RETURN p
```

In the MERGE clause, it is possible to specify different actions when the node is found or when is created, using ON CREATE and ON MATCH:

```
MERGE (p:Person { name : 'Marco'})
ON CREATE SET p.date_creation = 01/01/2018
ON MATCH SET p.last_access = 01/01/2018
RETURN p
```

The previous query specifies two different actions based on the current data situation: If the node with property *name* 'Marco' does not exist, it will be created, and the property *date_creation* will be set to the current date.
If the node is matched, the value of the property *last_access* will only be updated with a new date.

[108] http://neo4j.com/docs/developer-manual/current/cypher/clauses/merge/

- **UNION**[109]: acts as a conjunction operation for queries. Combines the action of multiple queries to produce a final result with all the required data.
  If we are interested in all the results, duplicates included, it is possible to use the clause **UNION ALL**.
  The number and the names of the columns must be identical in all queries combined by the UNION clause.
  Considering an example of professor and university courses. *(Figure 5.9)*
  If we want to get the list of the names of both elements, usually, we would have to use two different queries, but with UNION, we can put all together and receive all the list.

```
MATCH (p:Professor)
RETURN p.p_name AS names
UNION
MATCH (c:Course)
RETURN c.c_name AS names
```

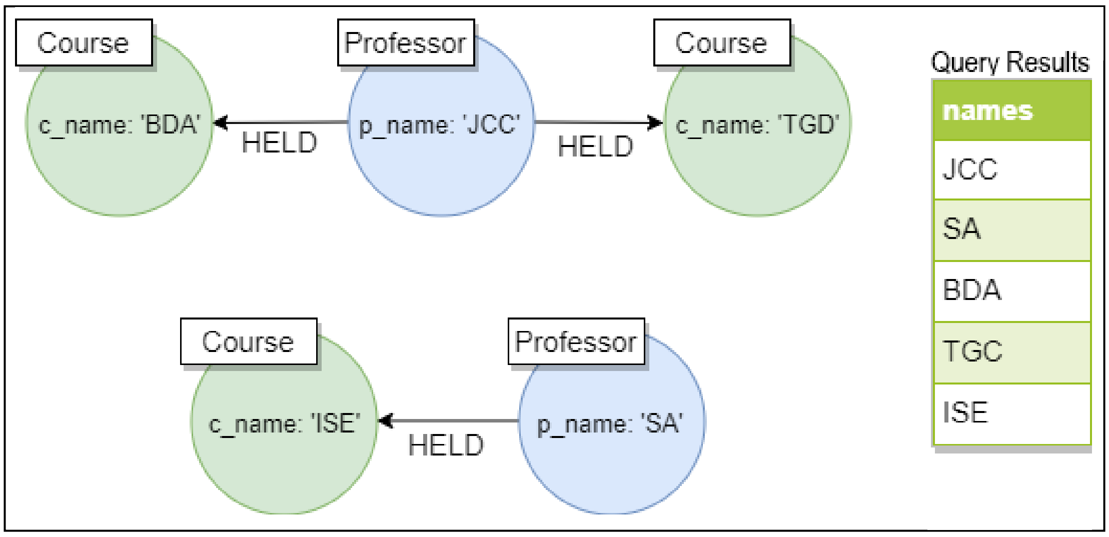


Figure 5.9: An university courses graph and its results of the previous query

- **FOREACH**[110]: like mostly imperative languages, is used to update the elements in a set of entities sequentially. Considering the previous example of friendships, this clause can update all nodes which have a particular property:

```
MATCH list = (p:Person)
WHERE p.age < 18
FOREACH (el in nodes(list) | SET el.underage = TRUE )
```

[109] https://neo4j.com/docs/developer-manual/current/cypher/clauses/union/
[110] https://neo4j.com/docs/developer-manual/current/cypher/clauses/foreach/

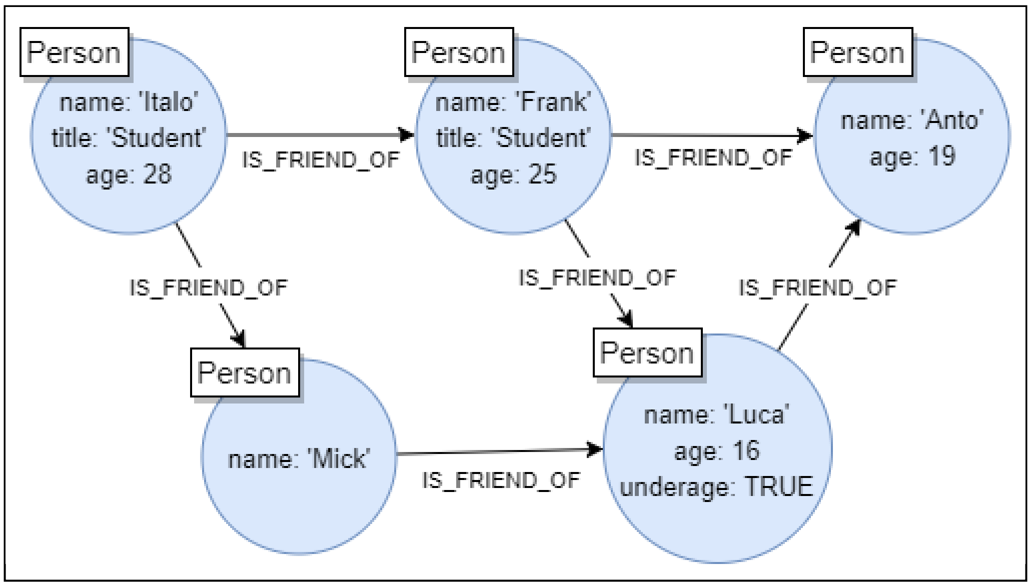


Figure 5.10: Friendship graph with new property for specific nodes

The query above, gets with the MATCH clause all Person nodes of the graph, filtering them by the property *age*, choosing only the youngest people above 18 years old, then, for each of them, it sets a new boolean property called *underage* to *TRUE (Figure 5.10)*.
As we can see, in the MATCH clause we create a sort of node list with all the underaged Person nodes, and then, in the FOREACH clause, we use nodes(list), which permits to receive each node of the list and process it.

- **CREATE/DROP INDEX**[111]: An index is a redundant copy of the information that making look-up operations faster.

  They need extra memory space, but there are specific situations that requires to pick out specific nodes directly, rather than discover them by the traversal walk.

  To support indexes, Cypher allows the creation of them for labels and property combinations

```
CREATE INDEX ON :Person(name)
```

  To remove them:

```
DROP INDEX ON :Person(name)
```

[111] https://goo.gl/2zXtzv

MATCH clauses can work both with or without indexes, but using them will help improve performance. The query syntax maintains the same style because in MATCH clauses it is not required the specification of the index.

- **CREATE/DROP CONSTRAINTS[112]**: In some systems, duplicates values would be classified as an integrity violation, and Neo4j with constraints[113] helps keep property values unique.
  For example, unique constraints are useful when working with usernames that are required to be unique.
  When a constraint is created, Neo4j creates *automatically* an index for the properties that are required to be unique, helping to keep track of the existing values.
  In this scenario, a CREATE operation will produce an error if it tries to create a node which violates constraints.
  The query to create a constraint and at the same time the index on the specific property is:

```
CREATE CONSTRAINT ON (p:Person)
       ASSERT p.username IS UNIQUE
```

  To remove both the constraint and the index:

```
DROP CONSTRAINT ON (p:Person)
       ASSERT p.username IS UNIQUE
```

The first four clauses, together with MATCH and RETURN are the basic **CRUD operations**[114] (Create, Read, Update and Delete) that can be used in Neo4j.

5.3) Key functions

Cypher has many functions that can change the query execution, to receive needed results. The most important are:

- **COUNT:** is a function used to count all the resulting rows from the queries RETURN clause. Two are the typical uses, *count(*)* when we want to count any returned nodes, or *count(element)* when we know what we want to count.

[112] https://goo.gl/jGRzZz
[113] http://neo4j.com/docs/developer-manual/current/cypher/schema/constraints/
[114] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

For example, to count how many Person we have in our graph:

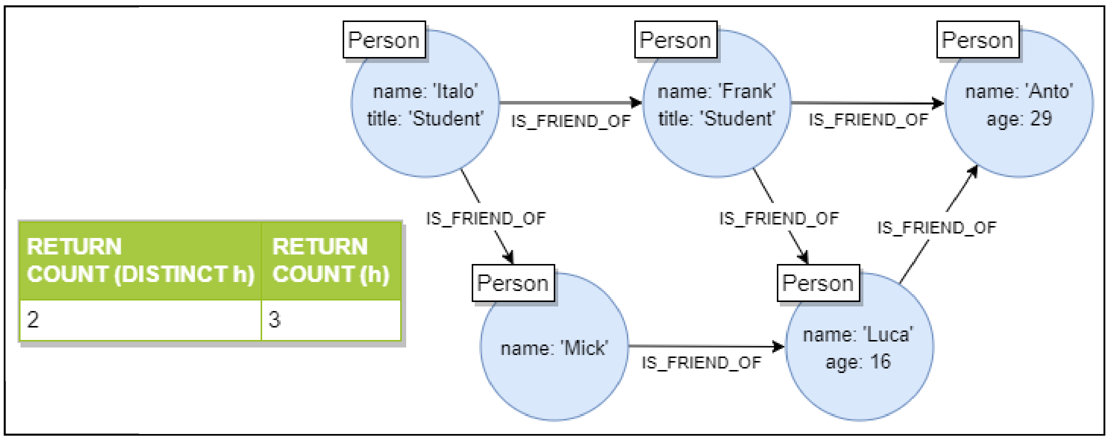
```
MATCH (n:Person) RETURN COUNT(n)
```

There are some cases when elements can be counted several times. To avoid the count of duplicates elements is possible to use the DISTINCT function.

- **DISTINCT**: can be used in combination with other functions and it is used to return unique values, without duplicates.

```
MATCH (n:Person {name : 'Italo'}) -
      [:IS_FRIEND_OF] -> (f) - [:IS_FRIEND_OF] -> (h)
RETURN COUNT(DISTINCT h)
```

This query counts the number of friends of friends, excluding duplicate nodes, like for example the node with the property name 'Luca', which is counted only one time *(Figure 5.11)*.



RETURN COUNT (DISTINCT h)	RETURN COUNT (h)
2	3

Figure 5.11: Different results with and without the DISTINCT function

- **NODES - RELATIONSHIPS:** Both functions are similar because they require a path to return nodes or relationships on it.
  For example, creating a list of paths using the MATCH clause, we can get the information that we need *(Figure 5.12)*:

```
MATCH p = (a:Person {name:'Italo'}) -
          [:IS_FRIEND_OF] -> (b) -
             [:IS_FRIEND_OF] -> (c:Person {name:'Luca'})
RETURN NODES(p)
```

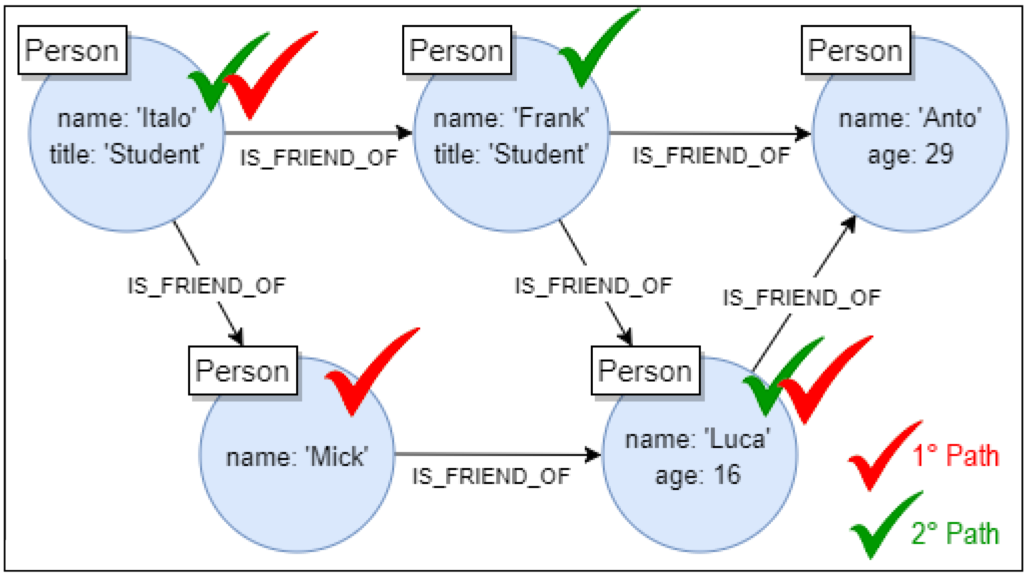


Figure 5.12: Different path from two nodes

- **LENGTH[115]:** this function, similar to count, is a counter for paths or collections and returns the numbers of path hops or the numbers of items within a collection.

```
MATCH p = (a:Person {name:'Italo'}) -
         [:IS_FRIEND_OF] -> (b) -
            [:IS_FRIEND_OF] -> (c:Person {name:'Luca'})
RETURN LENGTH(p)
```

This query, considering the same graph of the previous example, returns *2* and *2*, because there are two possible paths from the first node to the second and both paths have two hops.

- **COLLECT:** allows the aggregation of data, collapsing many rows into only one. It is useful to obtain one particular property from a collection of nodes, without having to process each element one by one.

```
MATCH (p:Person)
RETURN COLLECT(p.name)
```

The result value, considering the previous graph, would be:
["Italo", "Frank", "Anto", "Mick", "Luca"]

[115] https://goo.gl/m8NkTm

- **LABELS[116]:** returns all the labels associated with a specific node, in form of array *(Figure 5.13)*.

```
MATCH (p:Person {name:'Italo'})
RETURN LABELS(p)
```

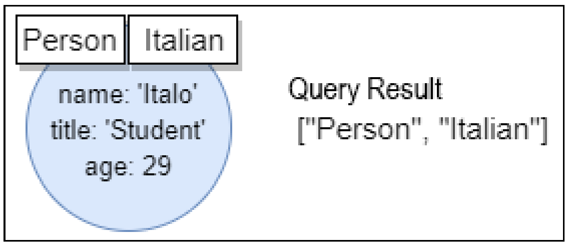


Figure 5.13: LABEL function

- **ID[117]:** returns the actual ID for the node or relationship within the database. This numerical ID is created automatically and cannot be set by a user. Usually, this ID is auto incremental, but when a node is deleted, that number becomes available and can be used for a new element.
  Nodes and relationships have two different lists of IDs, to avoid any problem.
  The simple query to get the ID is:

```
MATCH (p:Person {name: 'Marco Polo'})
RETURN ID(p)
```

It is possible to use the WHERE clause to obtain an element, knowing only its ID.

```
MATCH (n)
WHERE ID(n) = x
RETURN n
```

- **TIMESTAMP[118]:** returns the milliseconds between the current date and 1st January 1970. It can be used to update properties or to check the server timezone.

```
MATCH (p:Person {name : 'Anto'})
SET p.last_access = TIMESTAMP()
```

[116] https://goo.gl/77Nxra
[117] https://goo.gl/W4xkRx
[118] https://goo.gl/PA3wZN

- **TYPE[119]:** in every example during this dissertation, each query had specified the name of relationships which connect nodes. There are some cases when the name is not essential, or it is unknown. The function TYPE helps to explicit the type of a relationship supplied, that, in other words, is the name associated with it.

  ```
  MATCH (n:Person {name: 'Italo'}) - [r] -> ()
  RETURN TYPE(r)
  ```

  This query, with anonymous elements[120], will find any relationships that the node has, and then, return all the types, duplicates included.

5.4) Query optimisations

Although Neo4j offers high performance using its native graph storage, and Cypher is highly understandable, different are the techniques which can be used to improve response times or the readability of our queries:

- **Anonymous elements**: Both nodes and relationships can be associated with identifiers, used to reference the related elements during the query execution.
  The main rule about identifiers is to specify them only when the elements will be used later, because, avoiding identifiers will make the query easier to understand.
  We have already seen some unnamed (anonymous) relationships like *[:IS_FRIEND_OF]*, where the relationships have no associated identifiers, but, in some circumstances, like the query used for the *TYPE* function, we do not know even the name of the relationship. In these cases, we can use the unnamed relationship **[ ]**, or if we need an identifier, we can put it inside the square brackets **[r]** without expressing the relationship type.
  Nodes, follow the same rules, so our queries can have unnamed nodes or relationships causing no problems to the execution.
  Some examples of relationships:

`MATCH (p) - [r:HELD] -> (c)`	Fully specified relationship
`MATCH (p) - [:HELD] -> (c)`	Unnamed relationship

[119] https://goo.gl/BmQJ4Q
[120] Covered in the next section

`MATCH (p) - [ ] -> (c)`	Unnamed relationship with unknown type
`MATCH (p) - -> (c)`	Same as before

If we do not know the relationship direction, or is not essential to specify, the symbols < or > can be omitted.

About nodes:

`MATCH (p:Person)`	Fully specified node
`MATCH (p)`	Named node
`MATCH (:Person)`	Unnamed node
`MATCH ()`	Unnamed node with unknown label

- **Indexes and constraints:** searches in the graph space are optimised using indexes which allow the traversal avoid redundant matches, going directly to the correct element location. Indexes can be created on nodes and unique property values defined by a constraint.
  Searches in the graph space are optimised using indexes which allow the traversal avoid redundant matches, going directly to the correct element location. Indexes can be created on nodes and unique property values defined by a constraint.
  Usually, it is not necessary to specify whether or not and which indexes to use. However, to make sure which indexes should be applied is used the ***USING INDEX*** clause. For example, the following query will match all users using the index on the email property.

```
MATCH (u:User)
USING INDEX u:User(email)
WHERE u.email = *email*
RETURN u
```

This query makes sure that the index on the email property is used for searching. If the index is not present or cannot be used in the query as is, an error will be returned.
The USING clause must be specified before the WHERE clause.

There are some situations where using an index could decrease performance [121]. This is the case of queries that match large parts of indexes in which it might be faster to scan the label and filter out nodes that do not match, rather than using an index.
To avoid the use of indexes, it is possible to use ***USING SCAN*** after the applicable MATCH clause, and this will force Cypher to do a complete label scan.
Considering the following query, Cypher will ignore possible indexes on the label Person, performing a scan of all nodes:

```
MATCH (p:Person)
USING SCAN p:Person
WHERE p.born < 1988
RETURN p.name
```

- **Patterns in the MATCH clause:** As mentioned previously, the *WHERE* clause is not meant for pattern matching, because it is better used to filter the results when used with *START* and *WITH*. However, when used with *MATCH*, it implements constraints to the patterns described. Thus, the pattern matching is faster when we use the pattern in the MATCH section.
  As Cypher is declarative, it can change the order of the operations, and in these cases, *WHERE* clauses can be evaluated before, during, or after pattern matching.
- **Use labels to optimise searches and avoid global data scans:** the use of labels in queries can help to optimise the search process for the pattern because they help to shrink the domain.
  In general, queries without a specific pattern, which necessitate scanning the entire graph are not recommended.
- **Split MATCH clause and avoid Cartesian Products generation:** Rather than using multiple match patterns in the same *MATCH* statement, separated by a comma, is better to split the patterns into several statements.
  This process decreases the query time execution because every *MATCH* is performed on a smaller portion of data, for this reason, the first statement, should contain the pattern which has the "smallest cardinality".

[121] https://goo.gl/7dQp9s

Putting all patterns together in a single *MATCH* statement implies calculating all possible combination of the elements. For example:

```
MATCH (p:Professor), (c:Course), (u:University)
RETURN COUNT(p) as professors,
       COUNT(c) as courses,
       COUNT(u) as universities
```

The query implies the mapping of all possible triplets of the elements and then the filtering of the results.
An optimised query, with successive counting, could be, for example:

```
MATCH (p:Professor)
WITH COUNT(p) AS professors
MATCH (c:Course)
WITH COUNT(c) AS courses, professors
MATCH (u:University)
RETURN COUNT(u) AS universities, courses, professors
```

- **Avoid returning entire nodes:** Most of the queries return elements obtained from the data with the *RETURN* clause, but, when are required only some property values, is better to return them directly, and not the entire node, because doing that, can cause degradation of performance.
- **Profiling queries:** Despite the use of optimisation techniques, if queries are still slow, the next step is the profiling, which identifies the cause of poor performance during the execution. Using the ***PROFILE***[122] command before the query, Neo4j will run the statement and show which operators are doing most of the work.
  By running the following query on a graph with only three nodes, the result is the one in *Figure 5.14*.

```
PROFILE MATCH(n:Person) RETURN n
```

Neo4j will execute the statement and keep track of how many rows pass through each operator, and how much each operator needs to interact with the storage layer to retrieve the necessary data.

[122] https://goo.gl/6CkpZR

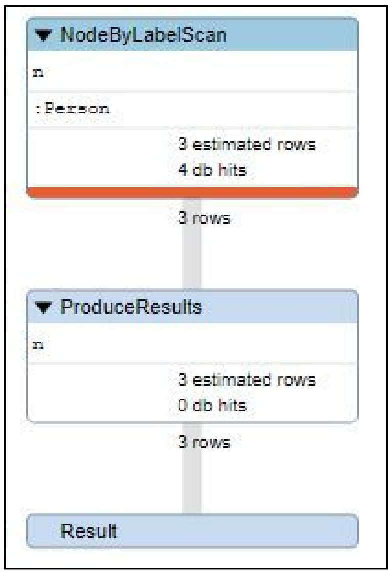


Figure 5.14: Result of a PROFILED query

The ***EXPLAIN*** command (Figure 5.15) is similar to the *PROFILE,* but it will show only the execution plan, without running the statement. The statement will always return an empty result and make no changes to the database.

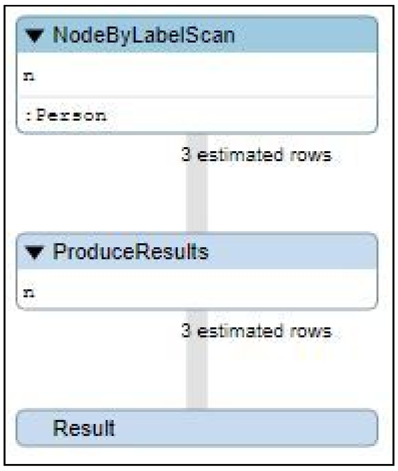


Figure 5.15: Result of a EXPLAINED query

By using one of these commands, each query execution is decomposed into operators, each of which implements a single unit of work. The operators are combined into a structure called an execution plan[123].
Each operator is annotated with statistics:

1. **Rows**: the number of rows that the operator PROFILE produced.

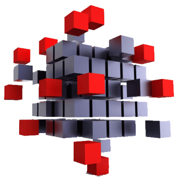
[123] https://neo4j.com/docs/developer-manual/current/cypher/execution-plans/

2. **EstimatedRows**: the estimated number of rows that will be produced by the operator. The compiler uses this estimate to choose a suitable execution plan.
3. **DbHits**: each operator will ask the Neo4j Storage Engine to do work such as retrieving or updating data. A database hit is an abstract unit of this storage engine work. The actions triggering a database hit are listed here.

● As we have already seen, the keyword USING can influence the decisions of the planner when building an execution plan for a query.
The clause ***USING JOIN*** [124], the most advanced type of hint, is used to enforce that joins are made at specified points in the graph.
The using of this clause requires extensive knowledge of the relationships in the graph, to improve performance, reducing the number of visited nodes. By forcing the join to occur on a specific node, if there are statistical reasons to do that, performance is improved.
In specific, a query that makes use of the USING JOIN clause, using more than one starting point (leaf), tries to join the two branches ascending from these leaves on specified nodes written in the clause. This will force the planner to look for additional starting points, and in the case where there are no more good ones, potentially pick an awful starting point. This will negatively affect query performance. In other cases, the hint might force the planner to pick an apparently bad starting point, which in reality proves to be an excellent one.

[124] https://goo.gl/He9pb3

Chapter VI



6) Data Modeling

Modeling is an abstracting activity motivated by the complexity of our World, in according to specific goals. We model to abstract information into a form which can be structured and manipulated, to apply techniques created to satisfy our needs. Usually, transforming and abstracting data can create semantic dissonance between how we conceptualise the real World and the data storage structure. [1][2]

Graph databases can help to decrease the difference between both elements because they are considered versatile tools that can be used to model various domains and problems.

A **data model** shows how the logical structure of a database is modelled, defining connections between data and how data is processed inside the system.

Neo4j uses a connected graph data models, based on relationships which connect different entities, making data access faster than other data models.

As mentioned before, Neo4j is a schemaless database where nodes can have as many properties and relationships as needed. The data model is implicit in the data contained in the database, and, it is a description of the reality that we want to store. For its descriptive nature, it is easy to make changes to expands or alters the data.

Reviewing past arguments, Neo4j offers four fundamental building blocks to structure and store data: *nodes*, *relationships*, *properties* and *labels*; extending the definition of graphs to **labeled graphs** *(Figure 6.1)*, where their elements can have labels to group nodes together or indicate their roles within data.

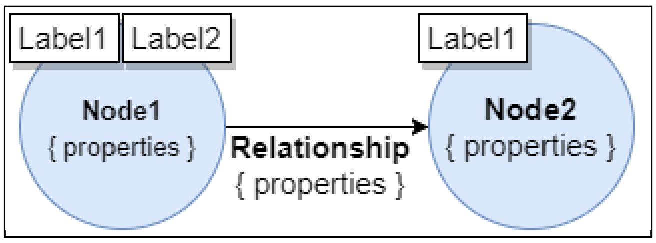


Figure 6.1: Labeled graph

6.1) Data Modeling with Graphs and Neo4j

These are the steps for a correct modeling phase:

1. During the initial stage, the first move is to understand every element in the domain, how they are related, and the rules that direct their state transitions. Graphs and sketches can help during this part.
2. Understand the end-user goals that motivate the creation of a database and create a sort of uses cases list where are presented all clients' needs in the form of sentences.
3. Identify nodes and relationships that appear in the sentences.
4. Identify properties and labels for every domain element.
5. Test and refine the model.

Agile methodologies, using **user stories**[125], provide a compact means for expressing clients' needs in an *user-centred point of view*. The classic template for user stories is:

```
AS A < type of user >,
I WANT < some goal >,
SO THAT < some reason >
```

The clause "AS A" establishes the type of user, the "I WANT" clause poses a question and "SO THAT" specifies the purpose of this story.

An example could be the one where a user wants to find interesting books based on his and other users' preferences:

```
AS A reader of books,
I WANT a list of books based on my preferences
       and similar users like me,
SO THAT I can find other books to read
```

Analysing the user story, a possible graph *(Figure 6.2)* could contain two kind of nodes: *READER* and *BOOK*, connected together by a relationship "*LIKES*". Other element, like specific properties about books, are not relevant to be specified for this story.

[125] https://goo.gl/EXm2CP

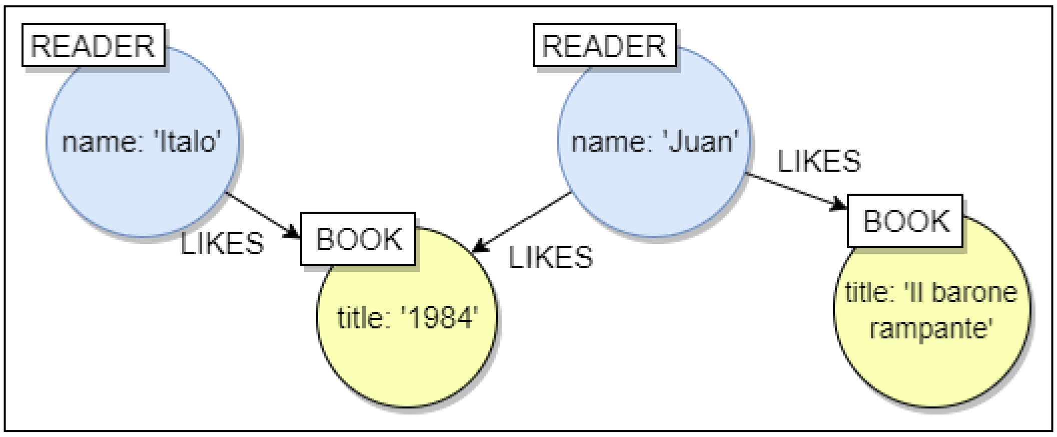


Figure 6.2: Graph for book reviews

A possible Cypher query to answer the user question could be:

```
MATCH (me:Reader {name: 'Italo'}) - [:LIKES] ->
            (:Book {title: '1984'} ) <- [:LIKES]
                 - (other:Reader) - [:LIKES] -> (books:Book)
RETURN books.title
```

6.1.1) Best practices and pitfalls

In graphs modeling, most relevant problems are concerning the choosing and treatment of nodes and relationships. [1][5]

Unfortunately, there is no one perfect way to model in a graph database, but follow some guidelines can help during the process:

- **Node or relationship?**

In a property graph, used in Neo4j, a relationship is a connection between two, and only two nodes. If we need more, it is not a relationship, and we must change it into a node.

A generalised graph model in which a relationship can connect any number of nodes is called **Hypergraphs**[126] [127].

Therefore, the hypergraph model allows any number of nodes at either end of a relationship. However, Neo4j only works with property graph, for this reason, some changes are required.

[126] https://en.wikipedia.org/wiki/Hypergraph
[127] https://neo4j.com/blog/other-graph-database-technologies/

If considering the following hypergraph (*Figure 6.3*), the *OWNS* relationship connects different people with their cars.

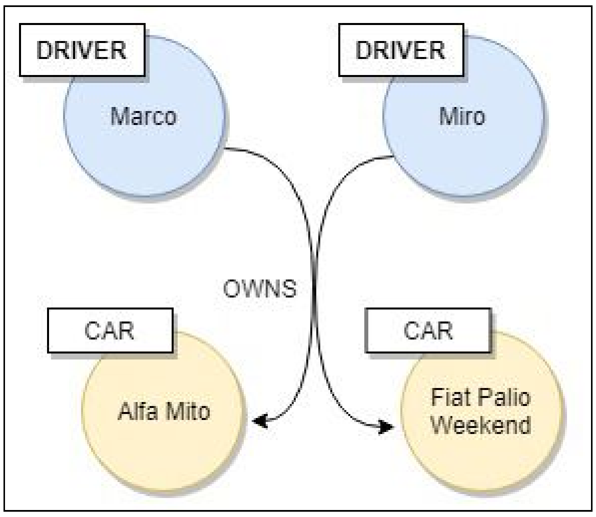


Figure 6.3: Cars Hypergraph

A possible solution for this kind of problem could be the splitting of all *OWNS* relationships into several, where each relationship has only two nodes (*Figure 6.4*).

In a property graph, are needed more relationships to express the same concept, because several *OWNS* relationships are required to express what the hypergraph captured with just one hyperedge.

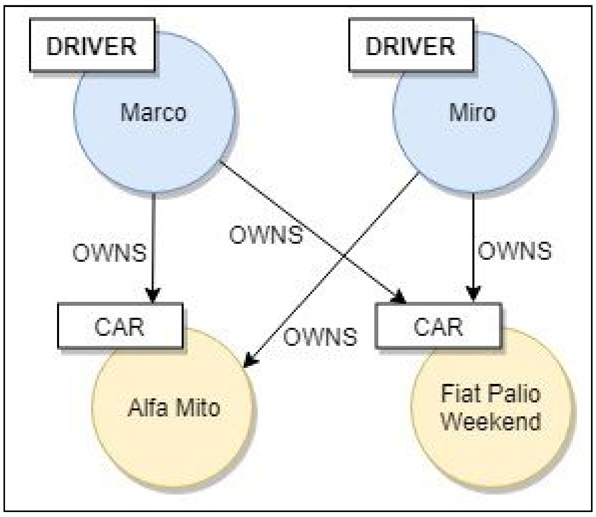


Figure 6.4: Cars Property Graph

In theory, hypergraphs should produce accurate, information-rich data models. However, in practice, it is easy to miss or limit some details.

Two are the advantages of using different relationships instead of one:

1. The data modeling techniques used to identify a property graph are more familiar and result less confusional for a development team.
2. In contrast to hypergraphs, in property graphs, every relationship could be extended in order to create new properties (such as "*primary driver*" for insurance purposes). Because hyperedges are multidimensional, hypergraph models are more generalised than property graphs.

But not everything has to be a node because properties can be used to express relevant values.

- **Node or property?**

Most common errors during the first creation of graphs concern properties and nodes. An example could be the graph below *(Figure 6.5)* which represents a user and some associated properties connected by the relationship "has_property".

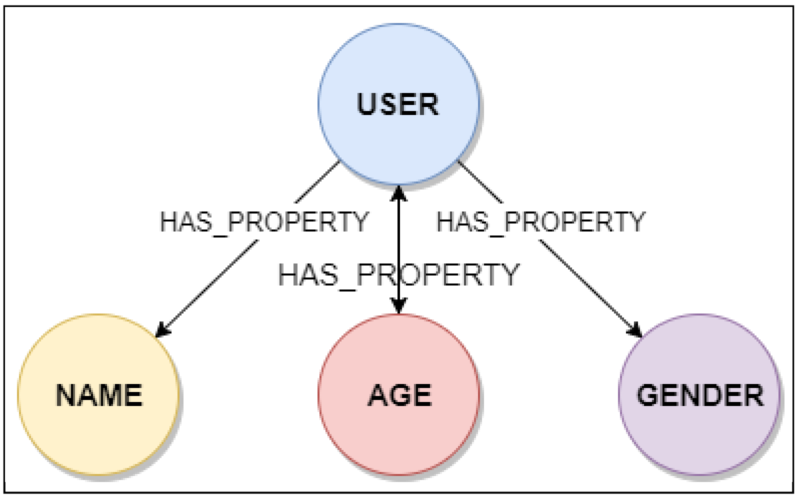


Figure 6.5: A USER with some properties connected by relationships

This graph can be reduced to a single node *(Figure 6.6)* because all other elements can be converted to properties:

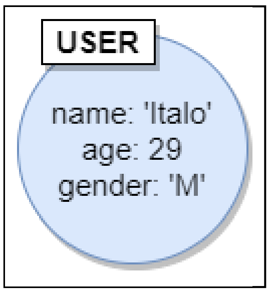


Figure 6.6: A node USER with some properties

However, on some occasion, a node for a property can be useful to share information and avoid redundancy. For example, to connect users with the same Address, indicating which one is current, a good practice is to use a node instead of a property. There are at least three ways to represent that information:

1. Create a node for the address using different relationship names *(Figure 6.7)*:

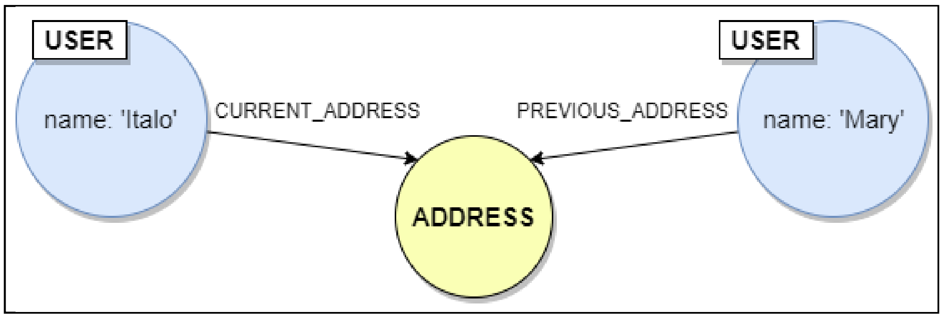


Figure 6.7: Two users sharing the same address with different relationships

2. Use the same relationship name but indicate which one is current with a property *(Figure 6.8)*:

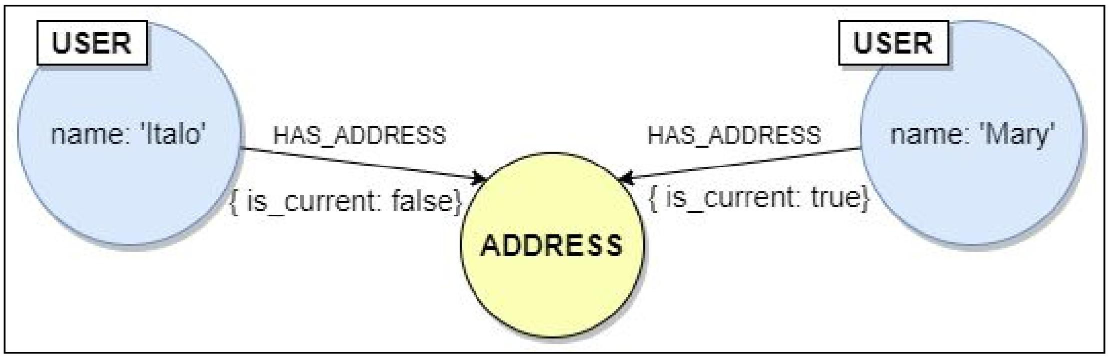


Figure 6.8: Two users sharing the same address with the same relationship but different properties

3. Use a technique called **reification**[128] where a relationship between two nodes is broken into two relationships connected by an intermediate node that represents the original relationship *(Figure 6.9)*.

[128] https://en.wikipedia.org/wiki/Reification_(computer_science)

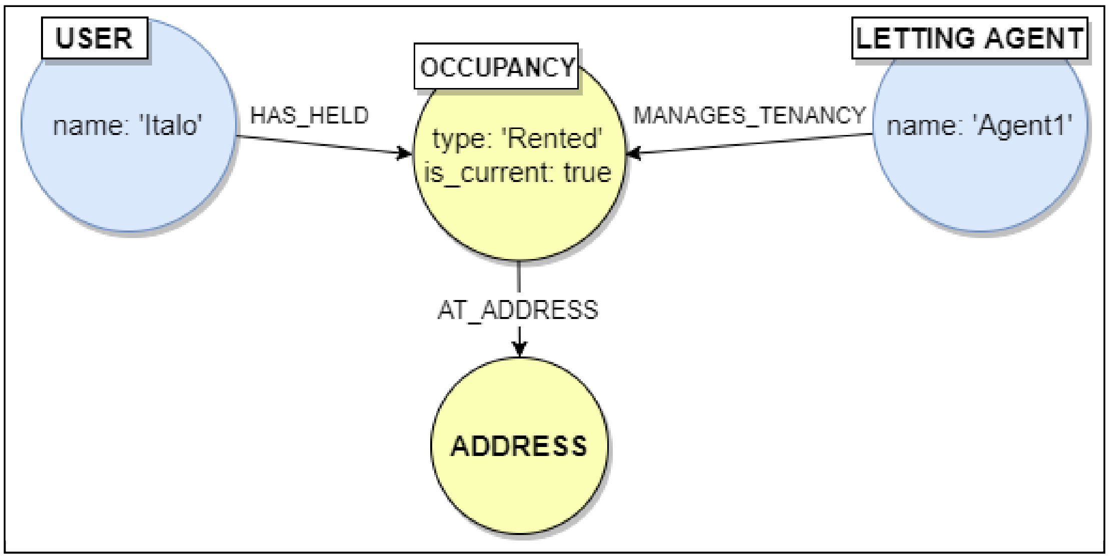


Figure 6.9: Reifying the relationship between USER and ADDRESS

Each way can be useful in a particular domain, and it is the designers' responsibility to choose the best one.

- **"Rich" properties**

Problems can occur when nodes have many properties *(Figure 6.10)*. Usually, it is not a good idea to split a node into others where are presented other properties.

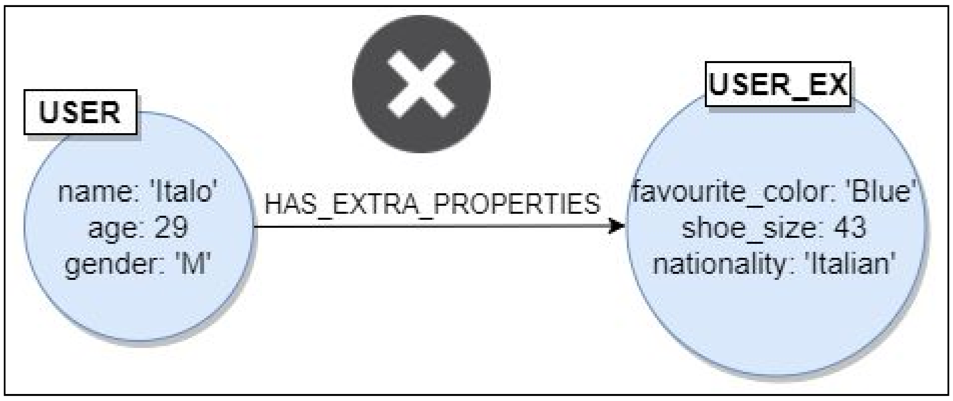


Figure 6.10: Extra properties stored in another node

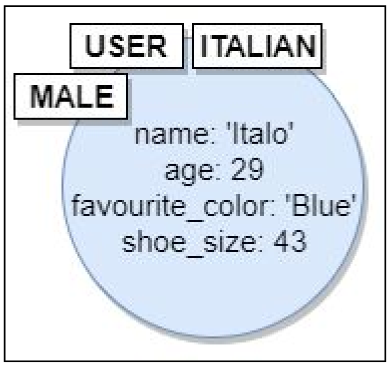


Figure 6.11: All properties together in a single node

A good practice is to put all together using properties combined with labels, to increase query simplicity and performance, reducing the number of nodes considered *(Figure 6.11)*. Choosing between label or property depends on the type of queries that they will be executed on the graph database.

- **Node with multiple concepts**

The previous concept is correct only when properties are related to the node concept, but when a node and its properties are two separate concepts, it is better to split them using a relationship to connect them *(Figure 6.12 and Figure 6.13)*.

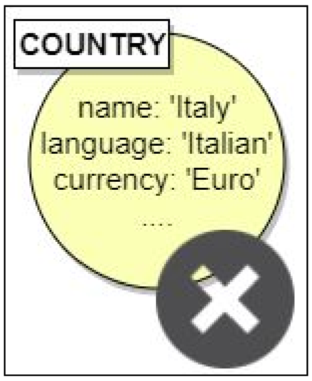


Figure 6.12: Wrong COUNTRY node

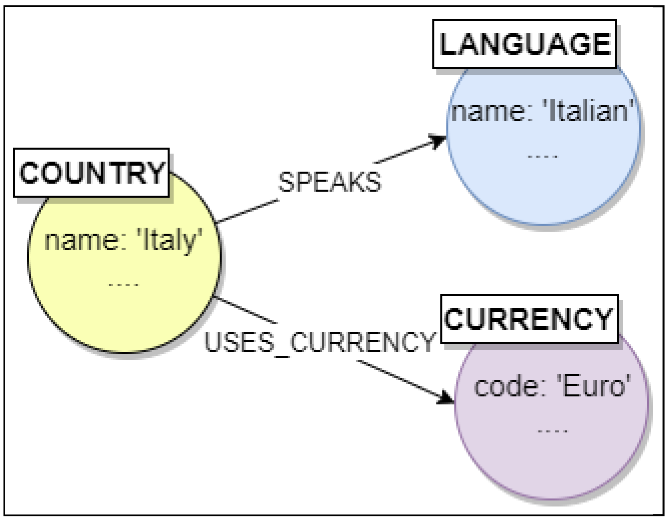


Figure 6.13: COUNTRY node split into three nodes

- **Unconnected graphs**

A significant anti-pattern to avoid during the modeling phase is the unconnected graphs. Neo4j based its success on its traversal speed which follows relationships to get relevant information.

A graph where nodes are not connected leaves a wealth of opportunities underutilised. In these cases, other kinds of databases would be better to store the data.

- **Dense node**

As specified before, the power of Graph Databases is the traversal which avoids complex join operations to get the required data. The traversal speed is correlated to the number of relationships that nodes have.

A problem can occur when some nodes are all connected to the same node. This node, called *dense node*, becomes problematic for the traversal, due to its high number of connections.

Instead of having direct connections from different nodes to this dense node, to avoid worsening performance, it is possible to create a metanode which groups all other nodes.

An example could be a fan-like graph database *(Figure 6.14)*, where fans are not directly connected to the Artist but are connected by metanodes:

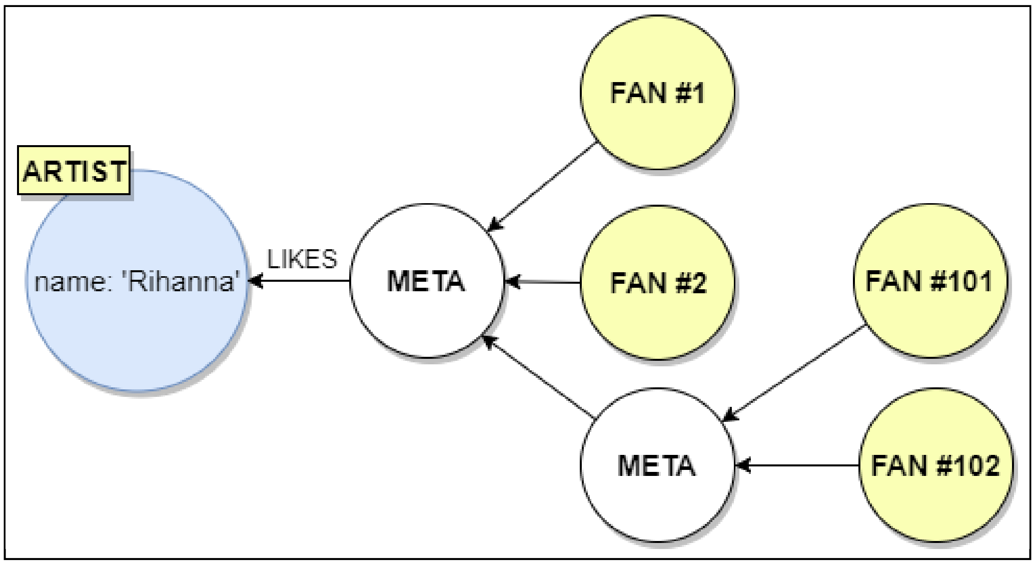


Figure 6.14: Strategy for dense nodes

- **Bidirectional relationships**

A relationship is an oriented connection between two nodes, but, in many cases, relationships go both ways.

The typical example is a family graph *(Figure 6.15)*, where are shown all related people in a family. The relationship between two brothers is bidirectional, but it cannot be represented by only one connection. In these cases, two similar relationships are required to express that information.

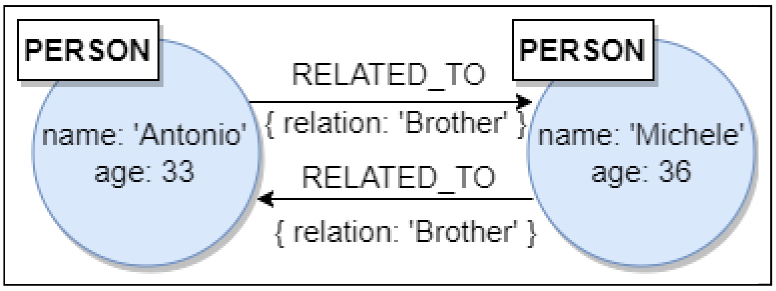


Figure 6.15: Family graph with two relationships RELATED_TO

The problem of this solution is the complexity of our graph because we are extending it creating more relationships, worsening the readability.

Thanks to Cypher, the relationship direction is not required, this means that instead of having two relationships, it is possible to use only one, changing all queries, without including the direction.

In this cases an example query could be:

```
MATCH (p1:Person {name: 'Antonio'}) -
          [r:RELATED_TO {relation: 'Brother'}]
               - (p2:Person)
RETURN p1.name, p2.name
```

Using this method requires fewer data and keeps queries cleaner.

- **Generic vs Specific Relationships**

Defining which type of relationships will be used in our graph is always a difficult choice, but it is always influenced by the queries that will be executed.

If considering the previous example of the family graph, the relationship *RELATED_TO* is *generic* because it can be used with all members of the family, only changing the value associated with the property. In this case, for example, a possible graph including other family members could be *(Figure 6.16)*:

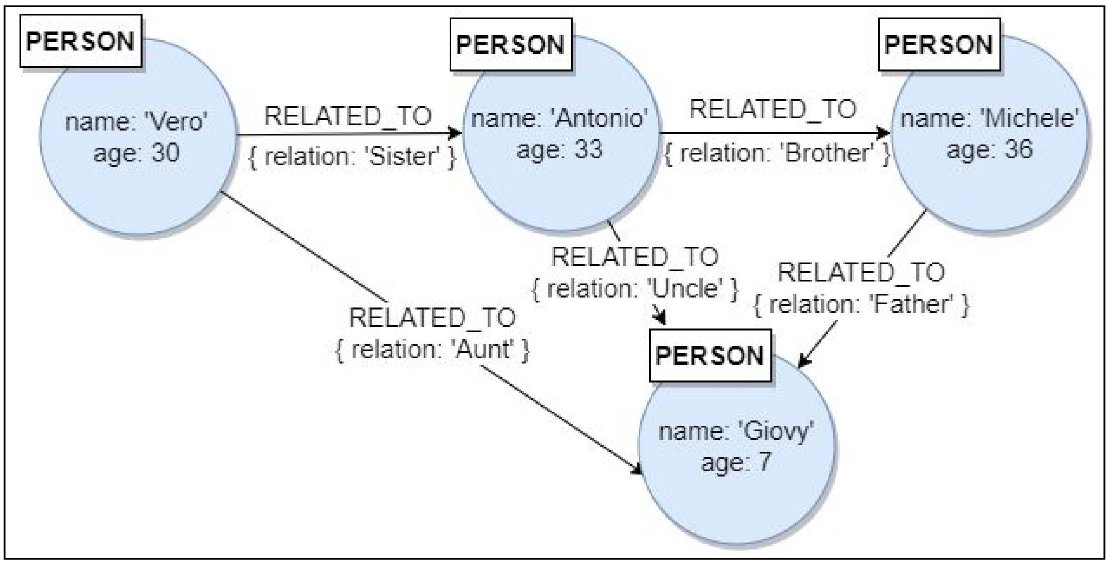


Figure 6.16: Extended family graph with the RELATED_TO relationship

Using this kind of relationship is enough when most of our queries will ignore property values presented in the connection, but, when the values are necessary, specific relationships are the best way to improve graph traversals *(Figure 6.17)*.

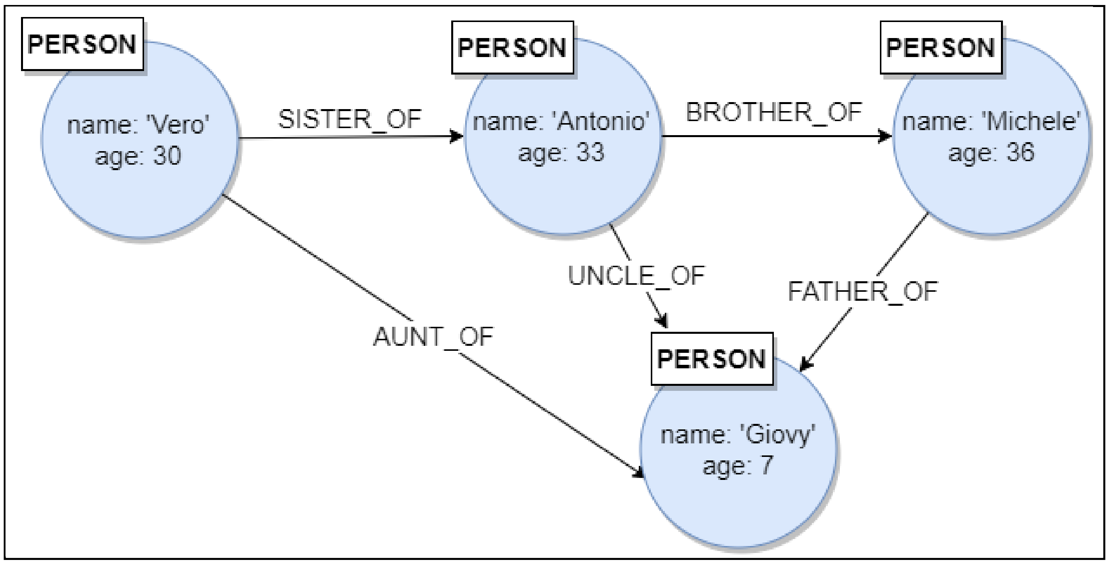


Figure 6.17: Extended family graph with the specific relationships

Adopting this strategy like the graph above, improves performance reducing the numbers of I/O operations, reading only the relationship type for every hop.

- **Property refactoring**

Following earlier definitions, MATCH and WHERE clauses consume most of the query execution time. A powerful mean to improve performance is the refactoring of some properties.

For example, considering the following query, the WHERE clause filters the results comparing the release date with a defined interval of date:

```
MATCH (b:Book)
WHERE b.releaseDate >= '01/01/2016'
          AND b.releaseDate < '01/01/2017'
RETURN b.title
```

This query returns the titles of all books released during 2016. A proper modification would be splitting of the date, putting the year in a single property.

```
MATCH (b:Book)
WHERE b.releaseYear = 2016
RETURN b.title
```

That gives a marked improvement in the query performance regarding its execution time.

6.1.2) Architecture

At this point of the graph database creation, there are several architectural decisions to be made, depending on the product in realization. Every choice is led by the clients' need, such as the type of APIs that will be used later, and the hardware specifications.

The primary choice is about Embedded vs Server Neo4j instance:

- **Embedded**: the database runs in the same process as the application and is ideal for hardware devices or desktop applications. It offers several advantages such as low latency because the application does not use the network to access the database. The main disadvantages are that the application is responsible for the database life cycle, which includes starting and closing it safely, and other problems with the *Garbage Collection* which can affect query performances.
- **Server mode**: the database resides on an external server independent both from the client platform and the application, because, they can access the database from any platform, only using an *HTTP client library* to manage

REST APIs. The main problems of this approach are about availability and network overhead.

Other choices are related to the hardware configurations, clustering, replication, use of cache sharding, load balancing and so on[129].

All these activities are part of the **capacity planning[130]**, essential both for budgeting and performances purposes. [1]

The ability to estimate the production needs depends on many factors.
More information about the number of expected users, graph size and query performances required we have, the better our estimation will be accurate.

Severals are the optimisation choices which we will be faced with the planning phase. Specifically, we can optimise for:

- **Cost**: using the minimum hardware necessary to offer the required service.
- **Performance**: by procuring the fastest solution which reduces query times. Performance can be improved, for example, using fast hard drives (Solid-State Drives[131] or Enterprise Flash Hardware[132]) and increasing the cache size.
- **Redundancy**: uses when the database availability is crucial, adding new database clusters to sustain a certain number of machine failures. For example, using Neo4j, redundancy of one can be achieved with three or four instances.
- **Load**: by scaling horizontally for reading load or vertically for writing load to serve all requests even in a peak load.

6.1.3) Testing

Once the domain model is created, the next step is to test it, in according to the uses cases list. Two are the techniques which can be applied during testing: [1]

A. **The simplest**: pick a start node and follow every relationship, reading each element and verify if it makes sense.
B. **Adopt a design for queryability**: every sentence in our list is translated into high-level queries, and then these queries are executed on the graph, proving the correctness of the domain model and implicitly improving the graph model. This method is typical of a *test-driven approach*.

[129] More information in the chapter IV related to Neo4j.
[130] *"The Art of Capacity Planning"* (O'Reilly Media). ISBN-13: 978-0596518578
[131] https://en.wikipedia.org/wiki/Solid-state_drive
[132] https://en.wikipedia.org/wiki/IBM_FlashSystem

For any extension, mistake or lack, going back to refine the model is lighter than modify the database during the execution time, but, at the same time, graphs are very adaptive, allowing modification every time easily than a traditional relational database system.

6.1.4) Performance testing

Usually, tests are made on a small portion of the entire graph, but what works well and fast on a piece of data, might not work so well on a higher graph with representative data. For this reason, **query performance tests**, created following guidelines, are required to evaluate the system.

After defining a set of queries with random starting nodes each time, recording of performances is essential to understand how they change modifying some characteristics, like graph size, hardware or configurations.

Performances are related both to the database and the application. This implies the execution of some **application performance tests**, to prove how the application responds to representative production usage scenarios.

Performance tests serve two purposes: they demonstrate the correctness of the system, and they help the identification of incorrect behaviours and bugs.

6.2) Evolving the domain

Extending an existing relational database is usually something challenging, which implies **migrations**[133] to change the model.

This technique provides a structured approach by applying a set of database refactorings which creates a new database that satisfies the changing needs. Database refactoring usually requires the changing of the data structure without losing any information, for this reason, is slow, risky and expensive.

Graph databases simplify the domain extension, reducing the necessity of a full migration.

Usually, the approach is to add new nodes and relationships without changing the original model, to support new requirements.

[133] https://en.wikipedia.org/wiki/Schema_migration

Adding new kinds of relationships, fortunately, does not affect any existing queries and is perfectly safe, but the changing of other elements, like relationship types or properties, might affect the system.

Run a representative set of queries is mandatory to maintain confidence that the graph is still working well.

The evolution of the domain is something usual during the system life. Modeling is a balancing act between the present and the future which implies compromises on some parameters while optimising for others.

It is essential to understand that there is no correct model for every problem, but only better and worse solutions. It is up to the designers to understand which one is the best.

Chapter VII

7) Flights and Cities, a Real World Example

The travel domain is interesting regarding data modeling challenges, for this reason, will be presented a modeling phase for a *flights and cities graph database*. [10]

7.1) Modeling Phase

7.1.1) Introduction and understanding of the domain

The purpose of this project is to create a graph database that can be used for planning flight travel. The two most relevant elements in the graph are *cities* and *flights*. Each element has some properties that are required to process and return the information expected from travellers. Their goal is to look at all the possible options for an itinerary, choosing the interesting cities.

A first phase could be the drawing of cities as nodes, creating a connection between them only if there are two or more direct flights between them *(Figure 7.1.1)*.

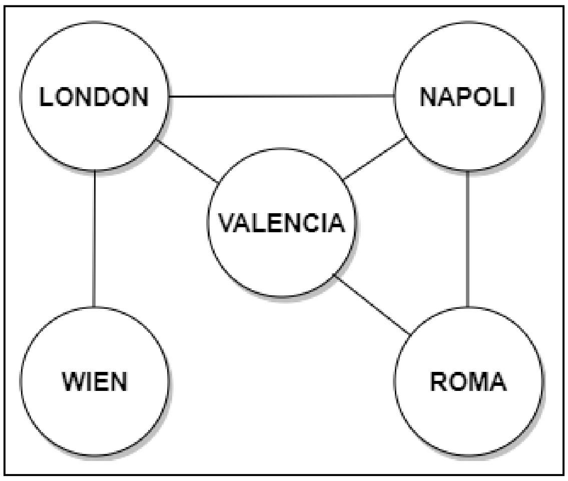


Figure 7.1.1: Cities and flight routes

In this high-level abstract figure, relationships are undirected, but it is implicit that there are at least two direct flights between that connected cities, one from the first to the second and vice-versa.

7.1.2) First possible solution

If looking at the figure below *(Figure 7.1.2)*, the first solution might come in the designers' mind could be the one where cities are nodes and flights are relationships between them.

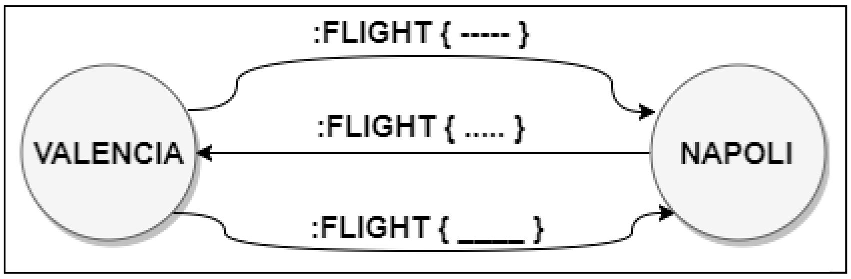


Figure 7.1.2: A graph of a first possible solution

There are few problems with this approach. Modeling flights as relationships works well only if an extension of the model in the future is not required. To support other functionality, like flight bookings, it is better to change the graph design.

Modeling entities as relationships is something not correct during the modeling phase, because, relationships cannot have other connections linked to them, for this reason, a different approach is required to solve this problem.

7.1.3) Identifying the entities

As mentioned previously, two are the relevant entities in our model:

- **City**: which has a name and a country. Usually, only the name is not adequate to identify a city uniquely, but in this simplification, we can use it as an identifier.
- **Flight**: uniquely identified by its code, has properties like duration, carrier, information about the airport and so on.

These two elements will be modelled as nodes *(Figure 7.1.3)*.

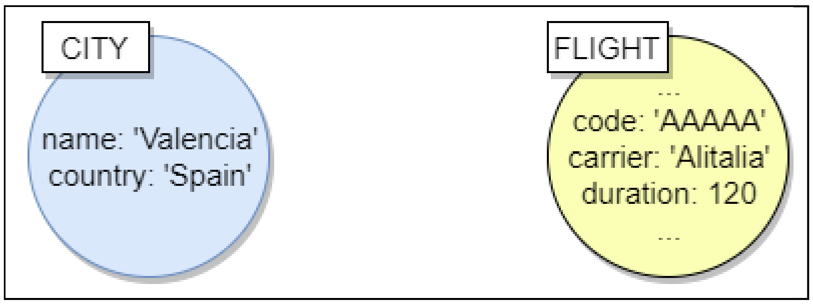


Figure 7.1.3: Cities and flights modeled as nodes

7.1.4) Identifying the relationships

As specified in the first possible solution, our graph will be created supporting possible future extensions, consequently, there will be two different relationships which connect cities and flights *(Figure 7.1.4)*:

- **HAS_FLIGHT**: connects the origin city to the flight;
- **FLYING_TO**: connects the flight to the destination city.

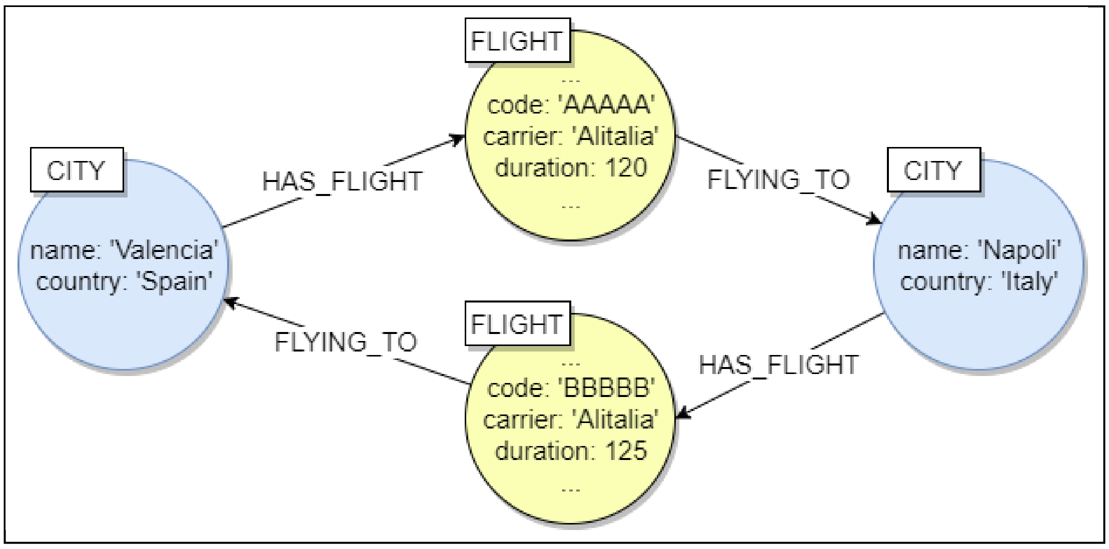


Figure 7.1.4: A flights and cities graph

7.2) Cypher queries

7.2.1) Cities

As defined before, cities will be modelled as nodes, with a unique city's name. To define this constraint, which ensures the uniqueness of the value:

```
CREATE CONSTRAINT ON (city:City) ASSERT city.name IS UNIQUE
```

It is a good practice to add uniqueness constraints before the creation of nodes because it ensures that no nodes will have the same property value.

Using the city name in this example is enough to ensure uniqueness, but usually, it is required other values to achieve that, like name and country together or ID values.

After defining the first constraint, the next step is the creation of cities:

```
CREATE (city:City { name: "Valencia", country: "Spain" })
CREATE (city:City { name: "Napoli", country: "Italy" })
CREATE (city:City { name: "Roma", country: "Italy" })
CREATE (city:City { name: "London", country: "England" })
CREATE (city:City { name: "Wien", country: "Austria" })
```

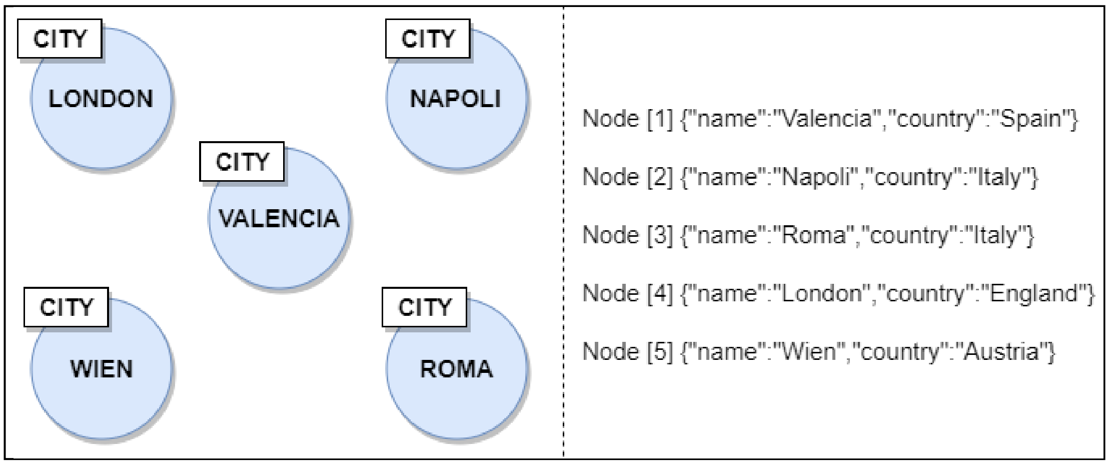
Executing twice a query will provoke a *Constraint Validation error*:

```
Node(x) already exists with label `City`
                    and property `name` = 'xxx'
```

To show the actual graph in the database can be used the following query:

```
MATCH (n) RETURN n
```

The result will be similar to the image below *(Figure 7.2.1)*, or it will be only text:



```
Node [1] {"name":"Valencia","country":"Spain"}
Node [2] {"name":"Napoli","country":"Italy"}
Node [3] {"name":"Roma","country":"Italy"}
Node [4] {"name":"London","country":"England"}
Node [5] {"name":"Wien","country":"Austria"}
```

Figure 7.2.1: All the cities in our graph

7.2.2) Flights

Since we have identified flights as entities, the procedure is the same as before.

First, we define a uniqueness constraint on the property *code* for every flight:

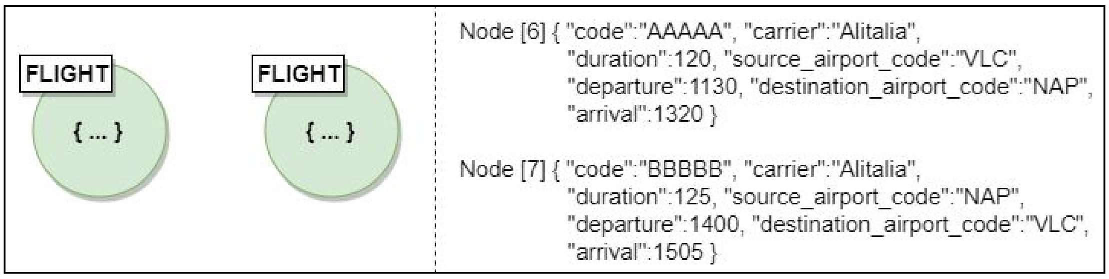
```
CREATE CONSTRAINT ON (flight:Flight)
                ASSERT flight.code IS UNIQUE
```

Then, we proceed with the flights' creations:

```
CREATE (flight:Flight { code: "AAAAA", carrier: "Alitalia",
        duration: 120, source_airport_code: "VLC",
        departure: 1130,
        destination_airport_code: "NAP", arrival: 1320 })

CREATE (flight:Flight { code: "BBBBB", carrier: "Alitalia",
        duration: 125, source_airport_code: "NAP",
        departure: 1400,
        destination_airport_code: "VLC", arrival: 1505 })
```

The actual graph situation, referring to the flights, is *(Figure 7.2.2)*:



```
Node [6] { "code":"AAAAA", "carrier":"Alitalia",
           "duration":120, "source_airport_code":"VLC",
           "departure":1130, "destination_airport_code":"NAP",
           "arrival":1320 }

Node [7] { "code":"BBBBB", "carrier":"Alitalia",
           "duration":125, "source_airport_code":"NAP",
           "departure":1400, "destination_airport_code":"VLC",
           "arrival":1505 }
```

Figure 7.2.2: All the flights in our graph

7.2.3) Relationships

Flights can now be connected to the source and destination cities by relationships. As defined before, it will be used two relationships:

```
MATCH (source:City {name: "Valencia"}),
            (destination:City {name: "Napoli"}),
                  (flight:Flight {code: "AAAAA"})
CREATE (source) - [:HAS_FLIGHT] -> (flight)
                       - [:FLYING_TO] -> (destination)
```

```
MATCH (source:City {name: "Napoli"}),
            (destination:City {name: "Valencia"}),
                  (flight:Flight {code: "BBBBB"})
CREATE (source) - [:HAS_FLIGHT] -> (flight)
                     - [:FLYING_TO] -> (destination)
```

The graph after the executions of the queries is *(Figure 7.2.3)*:

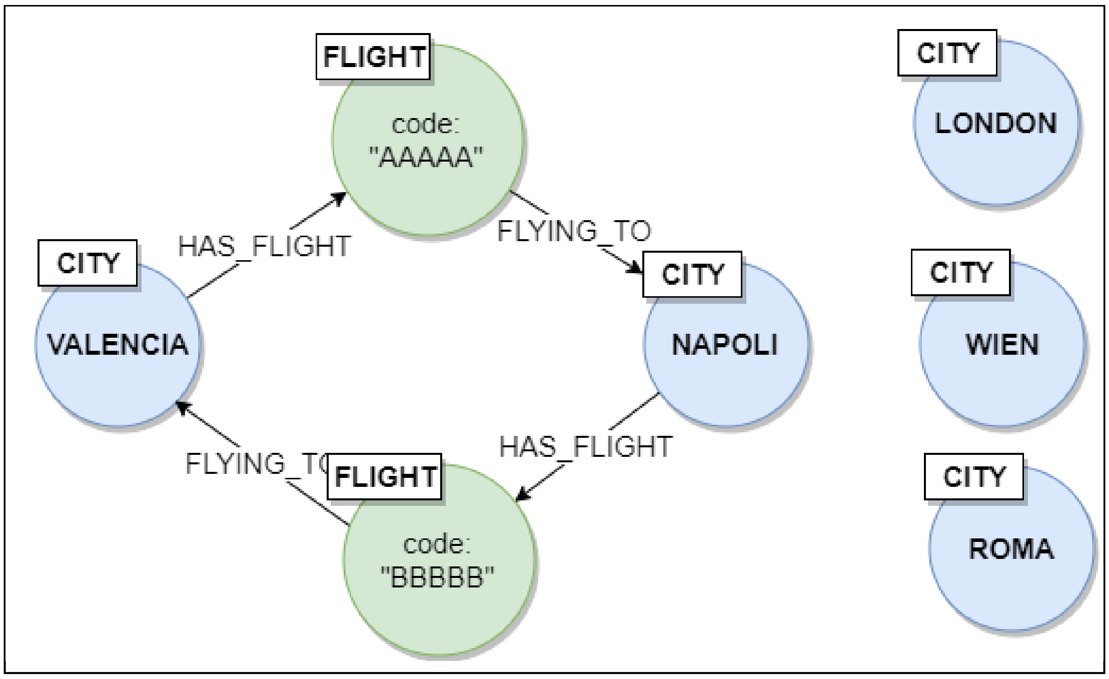


Figure 7.2.3: Flights and cities graph

7.2.3) Indexes

Searching elements without indexes is inefficient.

Neo4j creates indexes on a constraint property automatically, just after the creation of the constraint itself, for this reason, it is a good practice to add indexes for other relevant properties manually:

```
CREATE INDEX on :City (country)
CREATE INDEX on :Flight (carrier)        ............
```

Usually, indexes should be built on the properties that will be used in future during the query execution.

The creation of indexes for labels is not required because it is automatic by the system.

7.2.4) First traversal query

In the Real World, this graph would be extended than now. However, at the moment, it is possible the execution of some queries, to get the information that we need.

The first query could be the research of flights knowing only the source and destination cities.

```
MATCH (source: City {name: "Wien"}) - [:HAS_FLIGHT]
        -> (f:Flight) - [:FLYING_TO]
              -> (destination:City {name: "Valencia"})
RETURN f.code as FlightCode, f.carrier as Carrier
```

This is a classic query which will generate a list of all possible direct flights between the designated cities.

7.2.5) Planning itinerary

Our primary focus using this graph database is the creation of an itinerary between two cities connected by flights. Direct flights are always preferable, but if there is no one, most people are comfortable with some flights changes.

In a graph terminology, an itinerary, which includes a start node, intermediate nodes connected by relationships and an end node is called **path**.

The query to find a flight path is a mild extension of the previous query:

```
MATCH path = (source:City {name: 'Valencia'}) -
                   [:HAS_FLIGHT|FLYING_TO*0..3]
                       -> (destination:City {name: 'Wien'})
RETURN path
```

The flight node has been omitted and has been specified a pattern of relationships to match. The pipe symbol (|), which separate the relationship types, is used to specify multiple relationships that might be matched by the query.
The asterisk (*) and the range (**0..3**) specify the maximum hop numbers that the query should be traversing. Paths with more hops will be excluded from the results.

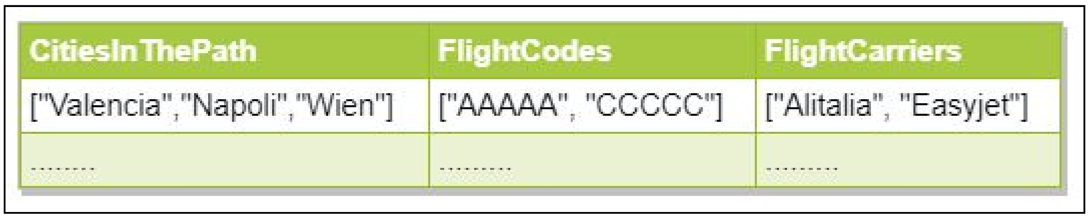
The query result is a set of nodes which formed paths from the source node to the destination, but, usually, travellers need visual information to understand which option could be better for them.

Two Cypher clauses could be used in combination with the "WITH" clause, to extract information from the result paths:

- **FILTER**: used to separate and group similar items, like nodes with the same label;
- **EXTRACT**: extracts elements, path by path, from a collection created with the previous clause, to display them to the user.

```
MATCH path = (source:City {name: 'Valencia'})
                - [:HAS_FLIGHT|FLYING_TO*0..6]
                     -> (destination:City {name: 'Wien'})
WITH
FILTER (f in nodes(path)
                WHERE "Flight" in labels(f)) AS flights,
FILTER (city in nodes(path)
                WHERE "City" IN labels(city)) AS cities
RETURN
EXTRACT (city IN cities | city.name) AS CitiesInThePath,
EXTRACT (flight IN flights | flight.code) AS FlightCodes,
EXTRACT (flight IN flights | flight.carrier)
                                        AS FlightCarriers
```

If considering only a few flights between our cities, a possible query result could be *(Figure 7.2.4)*:



CitiesInThePath	FlightCodes	FlightCarriers
["Valencia","Napoli","Wien"]	["AAAAA", "CCCCC"]	["Alitalia", "Easyjet"]
....	....	....

Figure 7.2.4: Different paths between two cities

The result shows a first possible itinerary from Valencia to Wien, which goes to Napoles with the flight *AAAAA*, and then to Wien with the flight *CCCCC*, both with different carriers.

Other properties could be shown putting other *EXTRACT* clauses.

In conclusion, the creation of a flight and cities graph database is manageable, and performances are incredibly high compared to the relational database systems where complex join operations must be necessary to obtain the same results.

Chapter VIII

8) Inside Neo4j

8.1) Index-free adjacency

Neo4j is an *open source* graph database with **native graph storage**. Its engine utilises **index-free adjacency** where each node maintains direct references to its adjacent nodes. This technique, avoiding the use of global indexes, is much cheaper and reduces query times because they are independent of the total size of the graph.

If we compare a *non-native* graph database with a *native*, depending on the implementation, index lookups could have a logarithmic execution time (using the **Big O notation**[134] is *O(log n)* where *n* is the number of nodes in the graph), versus the constant time (*O(1)*) for an index-free database.
In this case, to traverse *m* nodes, the indexes approach has an *O(m log n)* cost, compared with the *O(m)* of the native storage.

Figure 8.1 represents two graphs, the first in a non-native approach, which uses indexing to traverse nodes, and the other with index-free adjacency:

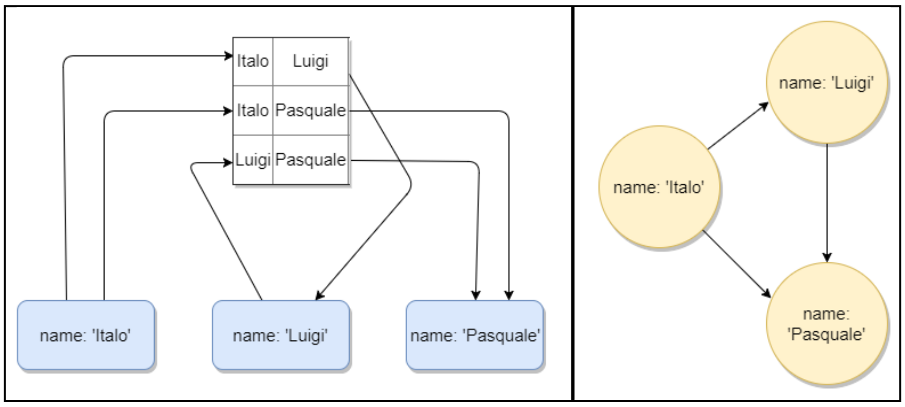


Figure 8.1: A non-native graph vs. index-free adjacency graph processing engine

[134] Bachmann, Paul; Analytic Number Theory. (1894)

Considering the *Figure 8.1*, to find *Italo*'s friends, the index lookup costs *O(log n)*, that would seem good for small values of *n*. However, reversing the direction of the traversal, asking who is friends with Italo, we have to perform multiple index lookups for every potential friend, and this implies an exponential increase in the cost (*O(m log n)*).

In a graph database with native graph storage, relationships can be traversed with the same cost in either direction. The cost to find *Italo*'s friends is *O(1)* for each friend, and the same cost is for the other question, following all the incoming relationships to the node *Italo*.

In conclusion, index lookups can work for small graphs, but index-free adjacency ensures high-performance independently of the graph size.

8.2) High-level Neo4j architecture

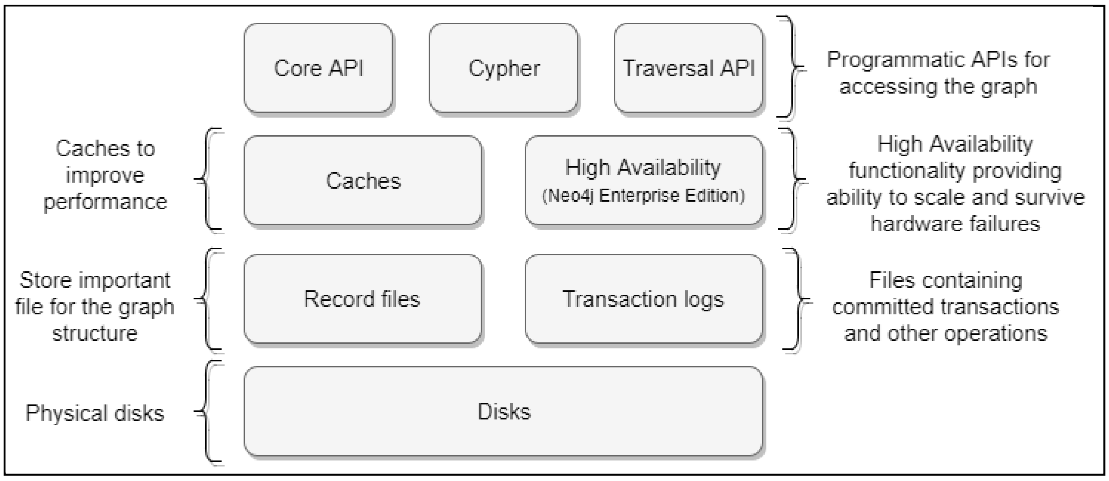

Core API
Cypher
Traversal API
Programmatic APIs for accessing the graph
Caches to improve performance
Caches
High Availability (Neo4j Enterprise Edition)
High Availability functionality providing ability to scale and survive hardware failures
Store important file for the graph structure
Record files
Transaction logs
Files containing committed transactions and other operations
Physical disks
Disks

Figure 8.2.1: Neo4j architecture

The figure above *(Figure 8.2.1)* presents the high-level architecture of Neo4j, where each piece fits together to offer one of the most reliable graph databases at the moment.

Without entering in low-level details, this work will analyse the structure with a bottom-up strategy, from the files on disk to the programmatic APIs. [1][2]

8.2.1) Disks

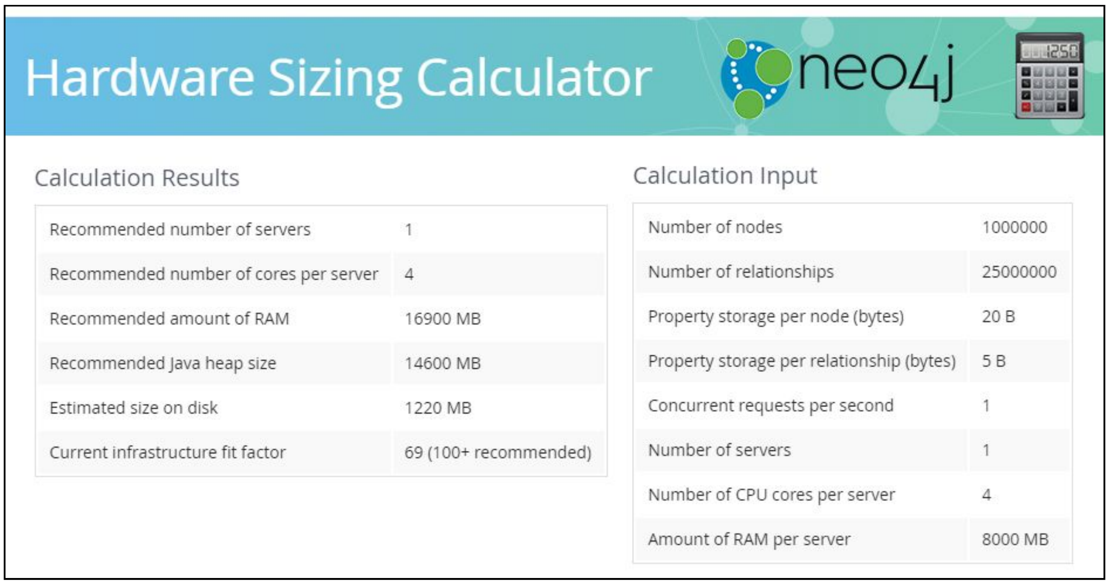
Physical disks are used to store the graph data. Several are the questions about storage, concerning performance and space.

In general, making use of disks that provide lower seek times, like Solid-State Drives, is better than traditional spinning disks[135].

Concerning the amount of space required, it depends on how much data will be stored in the database. All Neo4j data is placed under a single directory[136], and the main store files have fixed record sizes, which can help during the estimation of the required space.

Neo Technology, Inc. provides an **online hardware sizing calculator**[137] useful to identify the minimum hardware configuration required to run the system.

This calculator *(Figure 8.2.2)*, in its estimation, provides information about disk storage capacity, RAM and clustering setup recommendations.



Calculation Results

Recommended number of servers	1
Recommended number of cores per server	4
Recommended amount of RAM	16900 MB
Recommended Java heap size	14600 MB
Estimated size on disk	1220 MB
Current infrastructure fit factor	69 (100+ recommended)

Calculation Input

Number of nodes	1000000
Number of relationships	25000000
Property storage per node (bytes)	20 B
Property storage per relationship (bytes)	5 B
Concurrent requests per second	1
Number of servers	1
Number of CPU cores per server	4
Amount of RAM per server	8000 MB

Figure 8.2.2: An example estimation using the Hardware Sizing Calculator

[135] https://en.wikipedia.org/wiki/Hard_disk_drive
[136] By default "data/graph.db/" in a server-based setup
[137] https://neo4j.com/hardware-sizing/

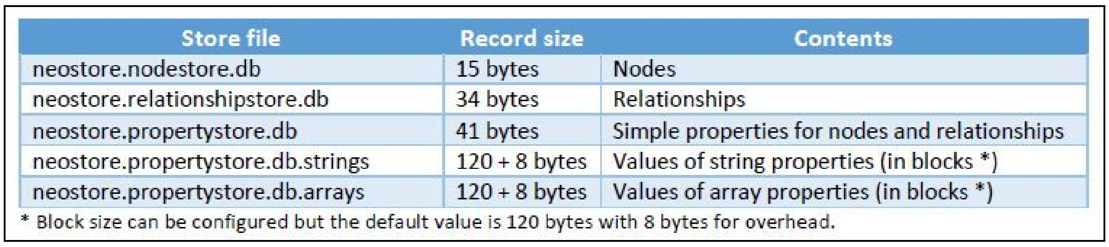
8.2.2) Store files

The necessary files for the graph structure in Neo4j are called **store files**. [1]

These store files are divided by record type, in consequence, there are separate files for every element like nodes, relationships and so on.

The layout of each file has been designed to optimize performance and storage. Files are all located under the main directory *data/graph.db/* and are prefixed by the word *neostore*.

The main store files used by Neo4j from the 2.1 version are *(Figure 8.2.3)*:



Store file	Record size	Contents
neostore.nodestore.db	15 bytes	Nodes
neostore.relationshipstore.db	34 bytes	Relationships
neostore.propertystore.db	41 bytes	Simple properties for nodes and relationships
neostore.propertystore.db.strings	120 + 8 bytes	Values of string properties (in blocks *)
neostore.propertystore.db.arrays	120 + 8 bytes	Values of array properties (in blocks *)

* Block size can be configured but the default value is 120 bytes with 8 bytes for overhead.

Figure 8.2.3: Primary store files

The uniform record size has been defined to enable fast lookups and traversals, because, lookups on IDs do not require any searching through the store file itself. Given an ID, the starting point for where the data is located can be directly computed because there is a fixed relation between the ID and the location within the store file.

For example, we know that the node with ID *1* will be the first record in the corresponding file. If we are looking for the node with ID *100*, this data will start from the byte *15,000* [(ID) *100 * 15* (bytes for each node record)]. This operation is calculated in a short constant time *O(1)* increasing query performance incredibly.

8.2.2.1) Node store file

The node store file (*neostore.nodestore.db*) stores node records. [1]
In the *Figure 8.2.4* is represented a node record and the meaning of its bytes.

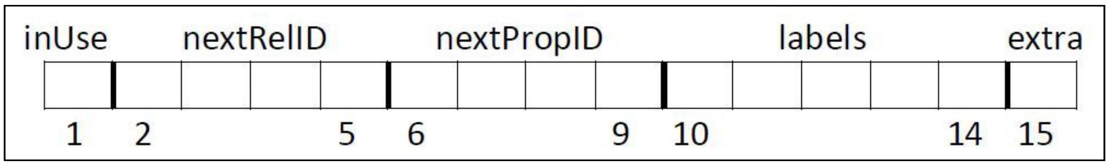


Figure 8.2.4: A node record

The first byte is the *inUse* flag and informs the database if the record is being used at the moment or it can be reclaimed for a new node.

From the second byte to the fifth included, is represented the ID of the first relationship connected to the node (*nextRelID*), and from the sixth to the ninth the ID of the first property for the node (*nextPropID*).

The following five bytes (*labels*) point to the label store for this node, and the final byte (*extra*) is reserved for flags to identify densely connected nodes (aka super-nodes[138] [139] [140]).

8.2.2.2) Relationship store file

The relationship store file (*neostore.relationshipstore.db*) stores relationship records *(Figure 8.2.5)*. [1]

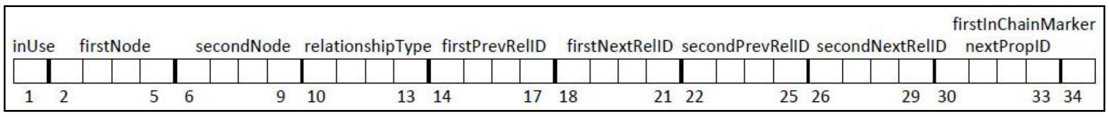


Figure 7.2.5: A relationship record

The first byte is the *inUse* flag and informs the database if the record is being used at the moment or it can be reclaimed for a new relationship.

firstNode and *secondNode* are the IDs of the start and end nodes of the relationship. In the record are presented both nodes because a relationship always belongs to both.

relationshipType is a pointer to the relationship type stored in the corresponding file.

The following four groups of four bytes are pointers for the previous and next relationship records for both the start and end nodes. The presence of these lists enables us to rapidly iterate through them in either direction, inserting and deleting relationships efficiently.

From the twenty-ninth to the thirty-third bytes, *nextPropID* references the ID of the first property for the relationship.

The last byte (*firstInChainMarker*) is a flag indicating if the current record is the first of a relationship chain.

[138] https://en.wikipedia.org/wiki/Dense_graph
[139] https://opencredo.com/neo4j-super-nodes-and-indexed-relationships-part-i/
[140] https://neo4j.com/blog/the-neo4j-2-1-0-milestone-1-release-import-and-dense-nodes/

8.2.2.3) Property store file

The property store file (*neostore.propertystore.db*) stores property records where are located the users' data in key-value pairs. [1][9]

Property can be attached to both nodes and relationships.

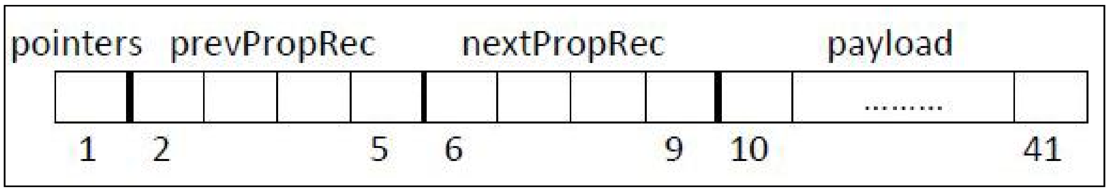


Figure 8.2.6: A property record

In the previous version of Neo4j[141], the property record was different from now because it had only 8 bytes for a property value. In the newer version, the record has been optimised to emulate a container, to incorporate many properties of variable lengths *(Figure 8.2.6)*. The *inUse* flag has been removed to optimize the *payload*.

The first byte has four high bits of the previous pointer and four high bits of the next pointer.

From the second byte to the fifth included, there is a pointer to the previous property record and from the sixth to the ninth the ID of the next.

There are four blocks of 8 bytes in the *payload*. Each of them is used in a different way depending on the data type stored in the property. The maximum number of properties which can be stored in a record is four because each property occupies between one and four property blocks.

If considering a block of 8 bytes, the type and index of the property are always necessary and occupy the first 3 bytes and a half of the fourth byte (1 byte + 1 byte + 1 byte + 4 bits).

The pointer to the property index file (*neostore.propertystore.db.index*) is used to store the property name and consequently, is always required.

Neo4j optimise automatically the property names allowing all properties with the same name to share a single record, reducing space wastage and I/O operations.

[141] Before the version 2

For each property value, the record contains either a pointer to a dynamic store record or an inlined value:

- The inlined value of the property is dependent on the data type being stored:
    - If the value is a JVM primitive that can be stored in 4 bytes, the other 4 bits of the fourth byte are skipped, and the property is stored into the remainder bytes.
    - If the value is an array or a long string using *DynamicStore*, all the 36 free bits in the block are used to store the property value.
    - If the value is a double or a long, the remaining 36 bits are skipped, and the next block is used to store the value. This option can cause space wastage, but it is better than the old manner to store properties, and it is rare.
- There are two dynamic stores used to store large property values: a dynamic string store and a dynamic array store (*neostore.propertystore.db.strings* and *neostore.propertystore.db.arrays*). For very wide strings or arrays, it is possible to use more than one dynamic record.

8.2.2.4) Physical store and traversal

Considering the following graph *(Figure 8.2.7)*, on this picture is represented how Neo4j truly stores this graph on disk:

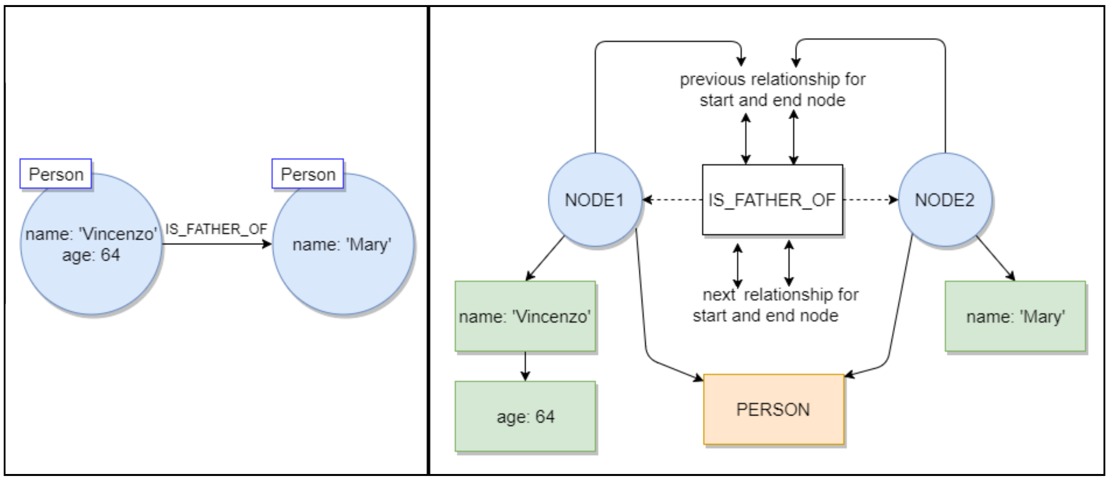


Figure 8.2.7: How the graph is physically stored

Each of the two node records has pointers to their first property and their first relationship in a relationship chain. Consequently, is easy to read all properties, following the singly linked list structure pointed in the record.

To find a relationship for a node, following the pointer for its first relationship, we can obtain all related relationships in a doubly linked list for that particular node and that allows the searching of the needed element.

The **traversal** power is its speed. To traverse a relationship from one node to another it is only necessary several ID computations:

1. From a node record, the database has to locate the first record in the relationship chain by computing its offset into the relationship store. This operation is simple because it is only the ID multiplied by 34, the size in bytes of a relationship record.
2. From the relationship record, to obtain the second node, the system has to get the ID and then multiplied again, but, this time, by 15, to acquire the correct node record in the store.

Another significant help offered by the fixed record size is the possibility to estimate the amount of space required on disk with more precision, knowing the number of the required elements.

An optimized storage layout is helpful to increase performances, but hardware considerations still have a significant impact on them. Neo4j uses in-memory caching to provide low-latency access to the graph, loading portions of the store files into the cache memory.

8.2.3) Neo4j Caches

The using of fast disks and optimized storage layout increases performance, but a latency penalty always happens if we are processing data from disks. This penalty can be reduced using cache memories which decrease the number of I/O operations on disk.

Recalling the previous section regarding cache[142], Neo4j offers two different caching systems: a file buffer cache (sometimes called **filesystem cache**) and an **object cache**.

8.2.3.1) Filesystem cache

The filesystem cache is an area of free RAM not allocated to any process that can be used to increase performance.

[142] Section 3.1.9

When an element is required, the system checks the filesystem cache to see if it has already been loaded there. If not, the related store file is read from disk into this memory area. The file is stored in the same manner which is on disk, with a *1:1 correspondence*. Every future request for the same element can be served directly from the cache, eliminating physical disk I/O. Every future change to the data is also written to the cache rather than the disk.

Every decision about the cache, including the flush of changes to disk, is entrusted to the operating system.

Neo4j also ensures that separate file buffer caches are maintained for each store file in the filesystem cache. This allows the possibility to configure how much space should be assigned to each file, independently from the others. With a default configuration, Neo4j will try to configure the quantity of RAM associated with each file store the best as it can.

8.2.3.2) Object cache

The object cache is an area within the **JVM heap**[143] where Neo4j, a JVM-based application, stores nodes and relationships optimised for fast traversals and quick retrievals.

Rather than only interact with raw files, Neo4j makes use of Java objects to store nodes and relationships. Combining the object cache and the Java objects for nodes and relationships, it is possible another performance boost which decreases enormously execution times.

If speaking of the object cache, there is a necessary clarification to be made. In contrast to the RAM sizes, larger heap sizes can cause problems for a few JVMs.

Java offers the automatic allocation and deallocation of memory for its objects, making use of the garbage collector to clean old ones. Large heap sizes can result in long garbage collector pauses and thrashing which decrease performance dramatically. In these cases, the JVM ends spending more time performing maintenance activities to free objects than the time to perform useful work, causing worsening of waiting times.

Choosing the correct heap size is always tricky, but tests can be useful to identify a good value in correlation to the graph size.

Below an image regarding the use of RAM for caching *(Figure 8.2.8)*.

[143] https://www.yourkit.com/docs/kb/sizes.jsp

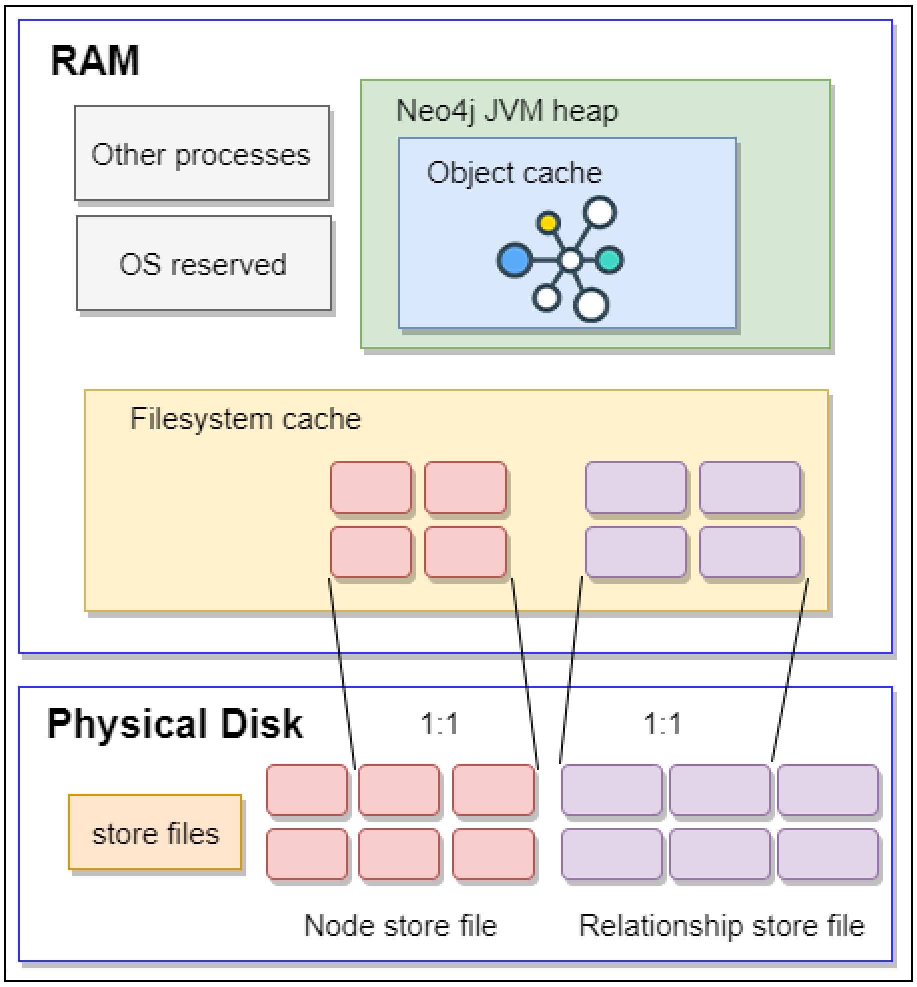


Figure 8.2.8: Use of RAM for caching

8.2.4) Transaction logs

As specified before, data changes are executed in the cache memory, but sometimes, system failures can cause problematic memory loss.

Neo4j offers a full ACID compliance which ensures that all committed transactions will never be lost. It uses a separate and durable transaction log where every change is flushed to disk upon every commit. This mechanism is called **write-ahead log** (**WAL**[144]) and provides atomicity and durability of the ACID properties.

Even if the committed transaction has been executed only in the cache memory, the transaction log is up-to-date on disk. Consequently, it can be used to

[144] https://www.youtube.com/watch?v=dEbw211njcc

recover and restore the system after a failure, recreating all operations until the last committed operation.

All transaction log files can be found in the main Neo4j directory, and they follow the naming format *nioneo_logical.log.**.

The transaction log is also useful for the high availability (HA) functionality allowing Neo4j to run in a clustered setup.

8.2.5) Neo4j High Availability

Allowed only in the Enterprise Edition, Neo4j offers a High Availability component that provides the capability to run a graph database in a clustered setup, allowing the distribution of data across multiple machines.

All this topic was already covered in chapter *IV*, in the section *4.1.3*, where are presented all the relevant information.

8.2.6) Programmatic APIs

At the top of the Neo4j architecture, there are the three primary APIs used to access and manipulate data. Each of these APIs can be used both individually or together depending on what operation is required. [2]

Excluding Cypher which has been covered previously in chapter *V*, the other two APIs are:

- **Core API**: it is an *imperative Java API* that can be used to access and manage all the graph.
  The readings are lazily evaluated[145] to improve performance, for this reason, the relationships are traversed only when the next node is required.
  For writes, are provided transaction management capabilities to ensure the ACID properties.
  The following portion of imperative code is checking every friend of a defined person, to find who has a dog. Friends are connected by the relationship *IS_FRIEND_OF*, and every person can have several *HAS_A* relationships to indicate possession of an object or animal.

[145] https://en.wikipedia.org/wiki/Lazy_evaluation

```
[...]
Iterable<Relationship> relationships =
        me.getRelationships ( Direction.INCOMING, IS_FRIEND_OF ):
for( Relationship rel : relationships )
{
   Node companionNode = rel.getStartNode();
   if ( companionNode.hasRelationship (
                               Direction.OUTGOING, HAS_A) )
   {
      Relationship singleRelationship =
              companionNode.getSingleRelationship (
                               HAS_A, Direction.OUTGOING);
      Node endNode = singleRelationship.getEndNode();
      if( endNode.equals ( dog ) )
      {
         //Do something because the node has a dog
      }
    }
}
[...]
```

- **Traversal API**: The Traversal Framework is a *declarative Java API* which enables the user to specify constraints that limit the visit of the graph during the traversal.
  With these constraints, it is possible to specify which relationship types follow, the direction and which type of traversal we want to perform (for example *breadth-first*[146] or *depth-first*[147] covered in the next section).
  In the following portion of code, we combine the Traversal API and the Core API. The first is used to declare which relationship we want to follow and in which direction, with a breadth-first strategy. The second is used to identify, given the path to the current node, if further hops through the graph are necessary or not.

```
Traversal.description()
   .relationships ( relationships.IS_FRIEND_OF,
                                    Direction.INCOMING)
   .breadthFirst()
   .evaluator ( new Evaluator()
```

[146] https://www.youtube.com/watch?v=s-CYnVz-uh4
[147] https://www.youtube.com/watch?v=AfSk24UTFS8

```
{
  public Evaluation evaluate ( Path path )
  {
     if ( path.endNode().hasRelationship(
                                relationships.HAS_A ) )
     {
          return Evaluation.INCLUDE_AND_CONTINUE;
     }
     else
     {
          return Evaluation.EXCLUDE_AND_CONTINUE;
     }
  }
} );
```

In conclusion, if performance is the most relevant requirement, using the Core API provides a fast and flexible control over the interaction with the graph, but it requires the explicit knowledge of how the graph data is laid to interact with it.

The Traversal API provides more abstraction that the Core API and it is useful to create specific functions like the "depth-first search to a specific depth". Due to its abstraction, it might not perform as optimally as the Core API.

Cypher is the most friendly because it is easy to use and understand, but at the moment performances cannot be compared with those of the other two APIs. Some structural changes to the graph can cause problems with the other two APIs, but Cypher, for its abstraction, is most tolerant.

The future tendency is to improve Cypher performances and offered functions, for this reason, is considered the future of Neo4j.

8.3) Traversal ordering

Although traversals offer fast walking through the graph, efficient queries are essential to improve the performance successfully. As mentioned before, every time a traverser visits a node, it decides which relationship has to be followed to visit the next node. Selecting an optimal path through the graph improves performance and reduce memory waste. [2]

In graph theory, there are two different algorithms which can be used to traverse the graph: depth-first and breadth-first.

For both algorithms will be used the same graph which represents a tree *(Figure 8.3.1)*, a particular graph where there are no cycles, and every two nodes are connected via a single path.

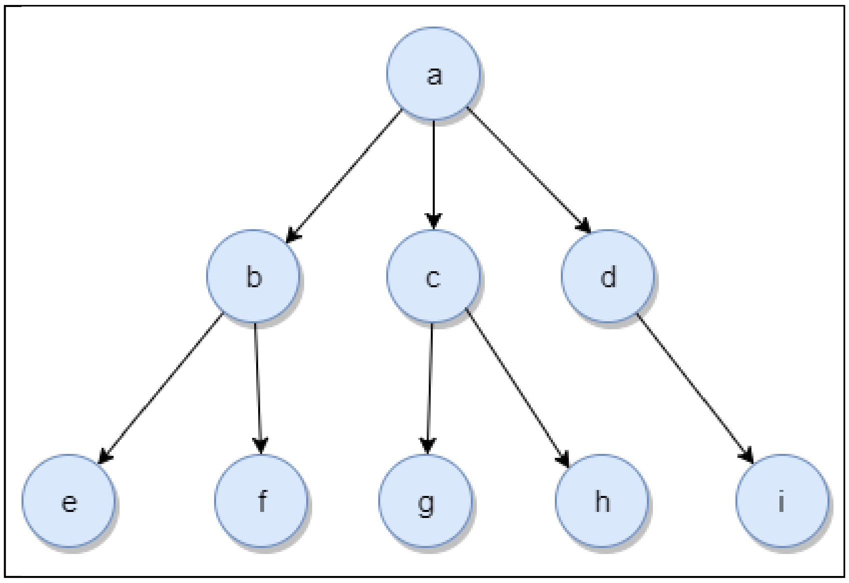


Figure 7.3.1: Ordered graph with a typical tree structure

8.3.1) Depth-first

During the traversal, this algorithm always chooses the first child of the considered node, that has not been visited before. If all descendant have already been visited, the algorithm goes back to the first node that has a not visited child.

Using the graph in the example, a walking using depth-first ordering will start from the node *a*, then to its first child *b* and then again to its first child *e*. In this example *e* has no child, then the algorithm goes back to the node *b* to visit the other child *f* and then, after that, it goes back again to continue the walking through the node *c* and so on.

Using the same rules, the path generated by this algorithm is:

$$a \rightarrow b \rightarrow e \rightarrow f \rightarrow c \rightarrow g \rightarrow h \rightarrow d \rightarrow i$$

which hops all the relationships in this order (*Figure 7.3.2*):

R1, R2, R3, R4, R5, R6, R7, R8.

By using this algorithm, the nodes in the left part of the graph are visited much sooner than the other nodes in the right part *(Figure 8.3.2)*.

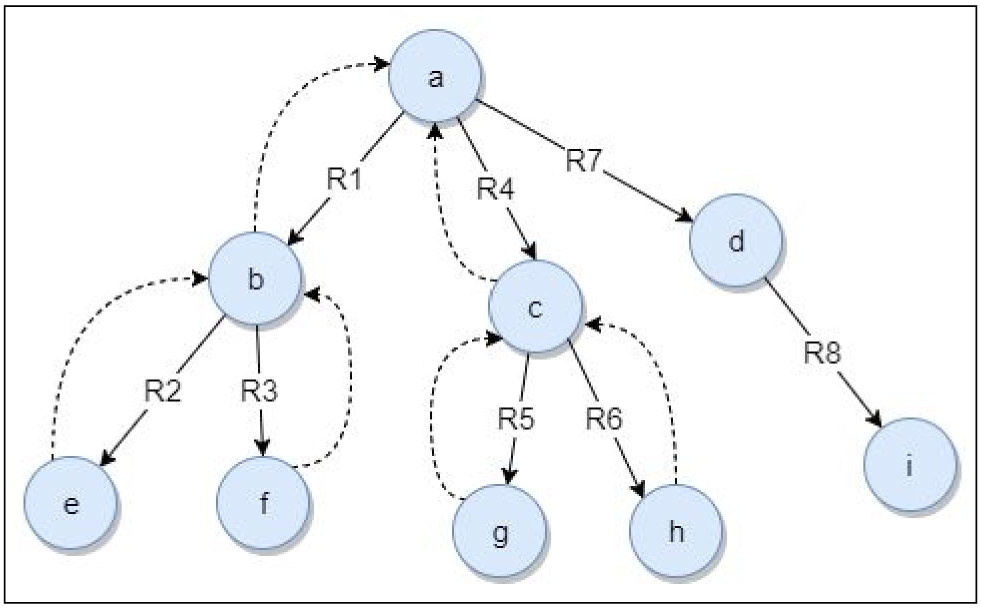


Figure 8.3.2: Walking the graph using the depth-first algorithm

8.3.2) Breadth-first

During the traversal, instead of visiting nodes more distant from the starting point firstly, the algorithm tries to go as wide in the graph as possible, for every level of depth. Particularly, the traversal first visits all siblings (nodes which have the same distance from the root node as the current node) and then it moves onto their children.

Using the graph in the example, if selecting the node *a* as starting point, the traversal will visit first its descendant, the node *b*, then the node *c* and finally the node *d*. In this case, there are no other siblings. Consequently, the algorithm goes back to the node *b*, visiting all its children, then goes back to the node *c* doing the same, repeating this procedure until all the graph has been visited.

By using this algorithm, the path generated is:

$$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i$$

which hops all the relationships in this order (*Figure 8.3.3*):

R1, R2, R3, R4, R5, R6, R7, R8.

In breadth-first traversal, the nodes closer to the root are visited earlier, leaving nodes further away from the root for later *(Figure 8.3.3)*.

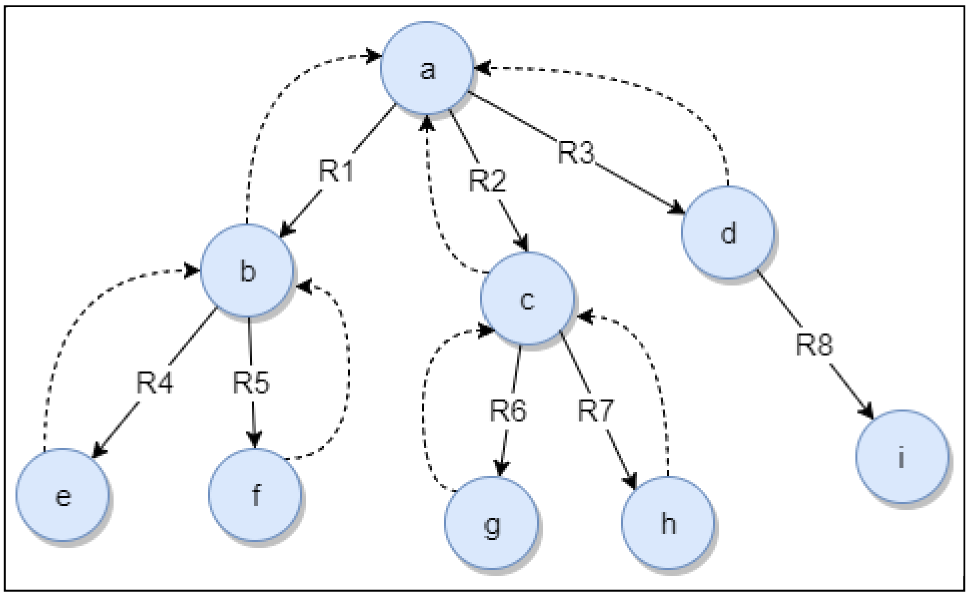


Figure 8.3.3: Walking the graph using the breadth-first algorithm

8.3.3) A comparison between the two algorithms

The traversal speed is directly proportional to the number of visited nodes during its walk on the graph, consequently, choosing the adequate algorithm is essential to improve performance.

The fundamental characteristics of the two algorithms are:

- depth-first traversal increases search performance if the solution is in the left part of the graph.
- breadth-first traversal is faster when the solution is closer to the starting point.

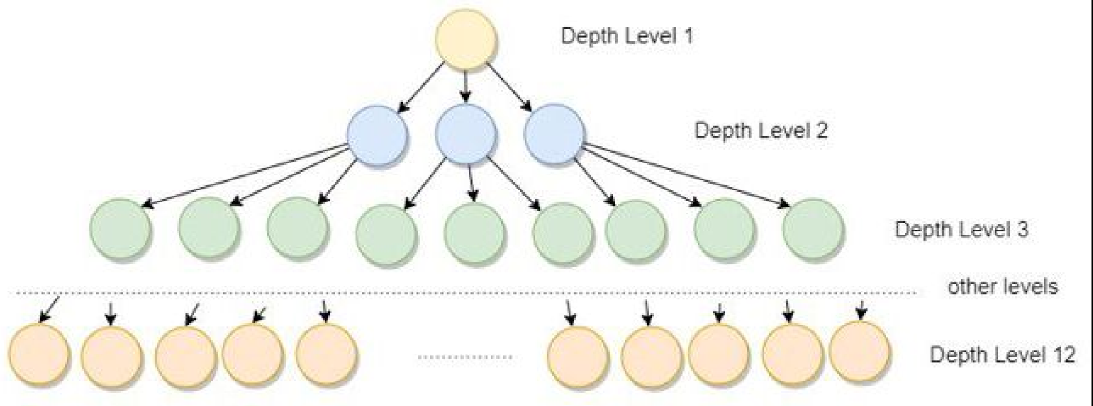
Knowing the location of the needed node is crucial to choose the best search algorithm, but, usually, the location is unknown, and that can cause problems during the traversal.

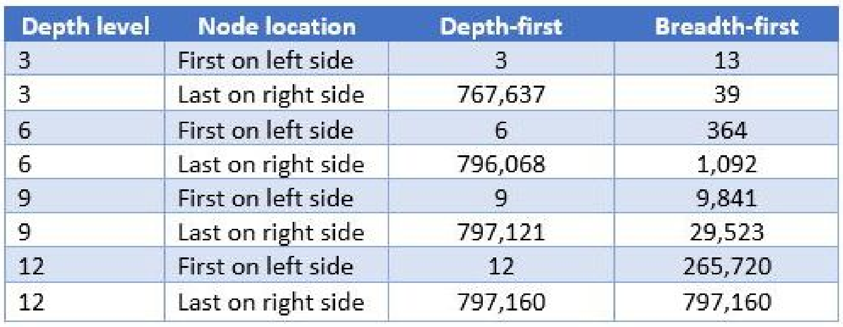
The larger is the graph, and bigger is the impact on the walking.

The next comparison, using a large graph, will illustrate the difference between the two algorithms, depending on the node location.

In the considered graph, each node has three child nodes up to depth level *12*; therefore, there are *797,161* nodes, connected by *797,160* relationships.

In total, eight tests were performed, two for each chosen depth (*3, 6, 9, 12*), the first with the target node location on the left side of the graph, the other on the right side *(Figure 8.3.4)*.





Depth level	Node location	Depth-first	Breadth-first
3	First on left side	3	13
3	Last on right side	767,637	39
6	First on left side	6	364
6	Last on right side	796,068	1,092
9	First on left side	9	9,841
9	Last on right side	797,121	29,523
12	First on left side	12	265,720
12	Last on right side	797,160	797,160

Figure 8.3.4: Traversal performance depending both on the node location and the depth

In conclusion, when the result is close to the starting node, breadth-first ordering is usually better than depth-first, but away from the starting node, depth-first can be extremely efficient, depending on the target node location.

In the worst-case scenario, to find the target, the entire graph needs to be traversed. However, due to the larger required memory of breadth-first traversal, the other algorithm is better.

In these cases, in addition to performance, memory waste is another relevant aspect that has to be evaluated:

- If using depth-first ordering, when the algorithm visits all nodes from the root to the last child in a path, it can go back to other nodes which have not visited children and forget about that visited branch of the graph. That causes less waste of memory.

- When using breadth-first ordering, the algorithm has to remember all the visited nodes and which descendants are not visited yet, causing an increase in the memory requirement.

In reality, the choosing of the algorithm is always complicated and depends both on the graph size and on the position of the target nodes, but usually, the breadth-first algorithm causes too memory waste, to be considered practical.

8.3.4) Bidirectional traversals

In the first versions of Neo4j, the traversals could have only one starting point, but from the version *1.8*, the limitation changed, introducing the concept of bidirectional traversals. [2]

If considering the typical pathfinding problem[148], where the system is looking for a path between two nodes, this concept could increase performance significantly.

In the figure below *(Figure 8.3.5)* is illustrated a graph where nodes, representing people, are connected by relationships.

In order to find a path between the start and end node, the system could start its traversal from the first node, examining all its connections and possibly meet the end node.

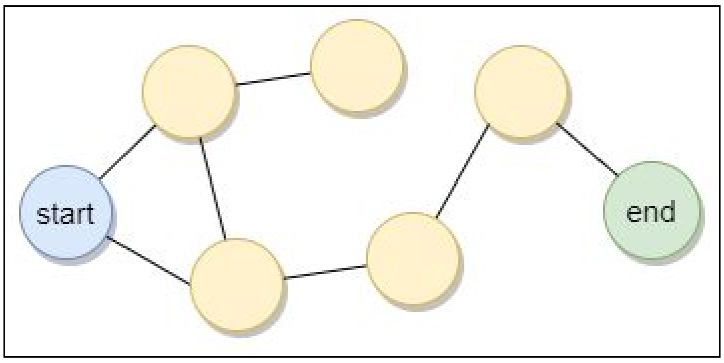


Figure 8.3.5: An example graph for the pathfinding problem

With bidirectional traversals, the system could start two traversals, from the start and end node, looking for any common meeting node, called *collision node*.

These kinds of traversals are possible because the direction of the relationship is not relevant during the execution because it can be hopped from both sides *(Figure 8.3.6)*.

[148] Other information in section 4.1.5

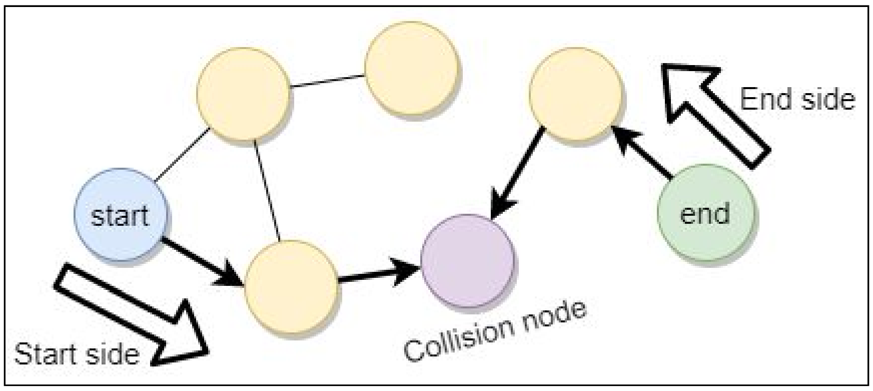


Figure 8.3.6: Bidirectional traversal in a pathfinding problem

In these traversals are used several components which keep information of every visited node, and if a collision occurs, which is the moment when both traversals realised that have visited the same node (even in different moments), the system can directly assert that there is a path between the two nodes.

This technique improves performance because it is necessary only one collision point, to verify the existence of a path.

The single traversal technique, instead, needs to find precisely the end node, starting from the first node, and, in a vast and dense graph, this can cause the examination of the entire database until it finds it.

By using the Traversal API, to find paths between two users connected by the relationship IS_FRIEND_OF, the code to execute a bidirectional traversal is:

```
BidirectionalTraversalDescription description =
    Traversal.bidirectionalTraversal()
           .startSide (
        Traversal.description().relationships(IS_FRIEND_OF)
                   .uniqueness(Uniqueness.NODE_PATH)
           )
           .endSide(
        Traversal.description().relationships(IS_FRIEND_OF)
                   .uniqueness(Uniqueness.NODE_PATH)
           )
           .collisionEvaluator (new Evaluator() {
                public Evaluation evaluate (Path path) {
                    return Evaluation.INCLUDE_AND_CONTINUE;
                 }
            })
           .sideSelector(SideSelectorPolicies.ALTERNATING, 100);
```

```
Traverser traverser = description.traverse(startNode, endNode);
Iterator<Path> iterator = traverser.iterator();
while(iterator.hasNext()) {
     //Shows the path
     System.out.println(iterator.next());
}
```

The code will show all the paths between the start and end node, avoiding duplicate elements, performing a bidirectional traversal.

8.4) Design constraints in Neo4j

Errors and limitations are typical in human nature, and even Neo4j suffers from certain constraints regarding the size of data. [10]

The limitation concerns the size of the address space for all the primary keys used to lookup elements[149].

Each element has a different *address space size* which limits the maximum number of storable elements:

- Nodes - maximum 2^{35} elements, no more than 34 billion;
- Relationships - maximum 2^{35} elements, no more than 34 billion;
- Relationship types - maximum 2^{15} elements, no more than 32,000;
- Properties - from 2^{36} to 2^{74} elements, according to the type of the property, which is no more than 68 billion.

Although those numbers are high, some graphs can reach that size and cause system failures. In these cases, identify and convert relationships into properties can alleviate those problems, but usually, a full database refactoring is essential.

Frequently, the problems are caused when the graph contains large domains together. Modeling each of these domains in a separate graph can solve those problems. An example could be a company graph which contains information about employees and sales. Although each information is related to the company, they represent different domains, and for this reason, they can be modelled in two different graphs.

[149] https://neo4j.com/blog/neo4j-3-0-massive-scale-developer-productivity/

Chapter IX

9) Installation & Configuration

This chapter has been written to guide the reader during the Neo4j installation. To run a graph database properly is required a minimum hardware configuration calculable through the online calculator tool, covered in section *8.2.1*.

9.1) Installing Neo4j Desktop on Windows

In order to install Neo4j on the latest version of Windows, the steps are:

1. Download the latest release of *Neo4j Desktop*[150] and execute it *(Figure 9.1.1)*.

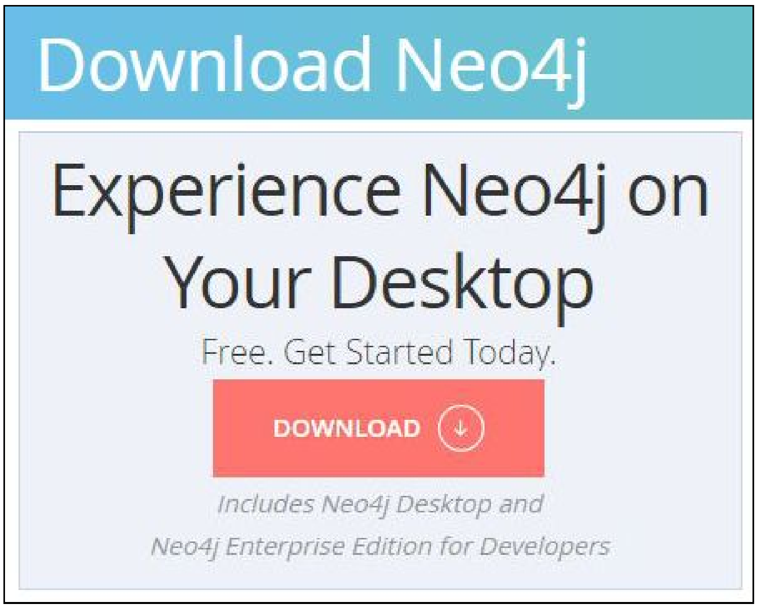

Download Neo4j
Experience Neo4j on Your Desktop
Free. Get Started Today.
DOWNLOAD
Includes Neo4j Desktop and
Neo4j Enterprise Edition for Developers

Figure 9.1.1: Download Neo4j

[150] https://neo4j.com/download/

2. During the installation is required a Neo4j free account. The login is possible with the all common social networks *(Figure 9.1.2)*.

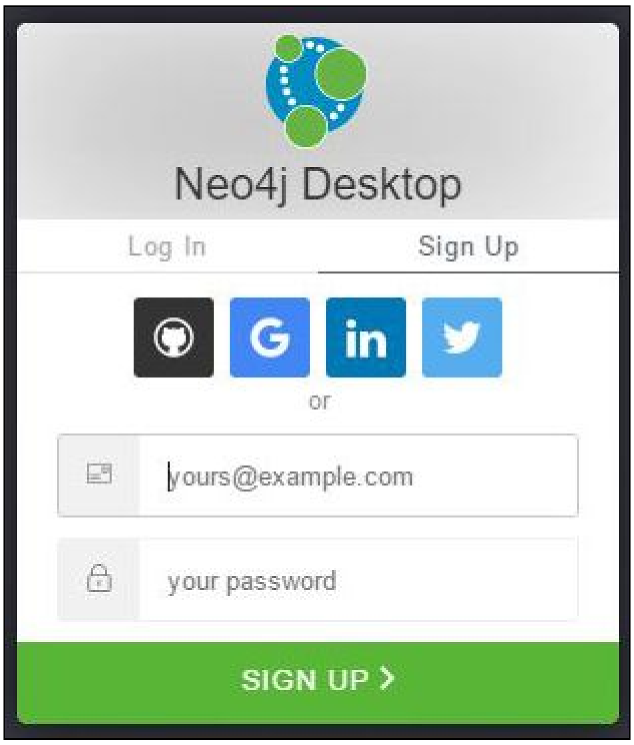


Figure 9.1.2: Login or Sign Up Neo4j Desktop

3. The installation will download and install the latest available version of *Oracle Java*[151] [152] automatically *(Figure 9.1.3)*.

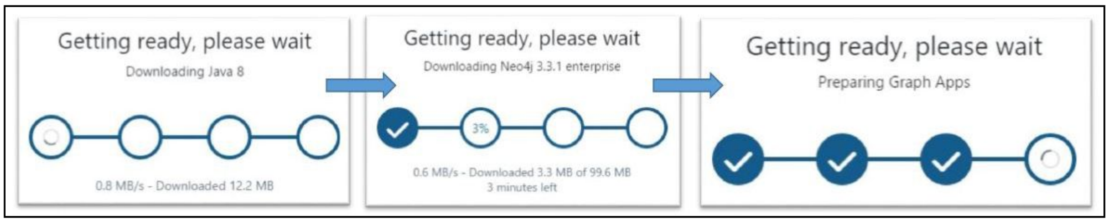

Getting ready, please wait
Downloading Java 8
0.8 MB/s - Downloaded 12.2 MB
Getting ready, please wait
Downloading Neo4j 3.3.1 enterprise
3%
0.6 MB/s - Downloaded 3.3 MB of 99.6 MB
3 minutes left
Getting ready, please wait
Preparing Graph Apps

Figure 9.1.3: Neo4j installation

4. As soon as the installation process is finished, Neo4j Desktop *(Figure 9.1.4)* will be ready to manage our graphs. The installation of other version is similar to this.

[151] https://www.oracle.com/java/index.html
[152] http://www.oracle.com/technetwork/java/javase/downloads/index.html

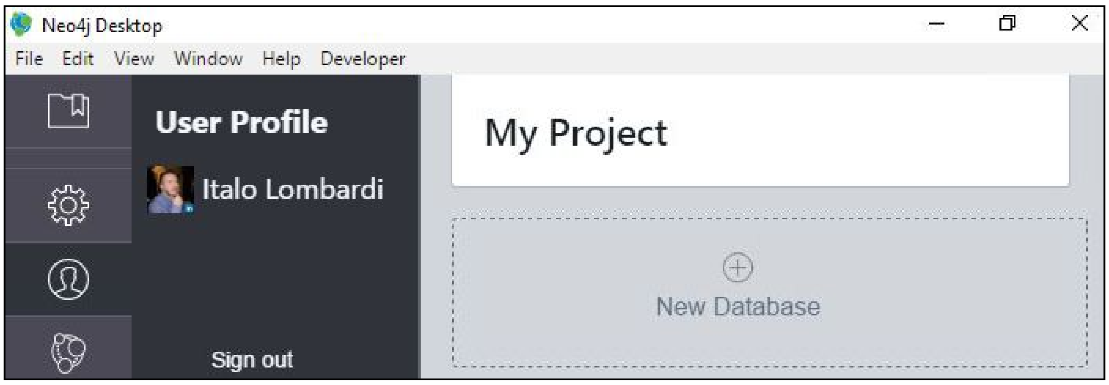
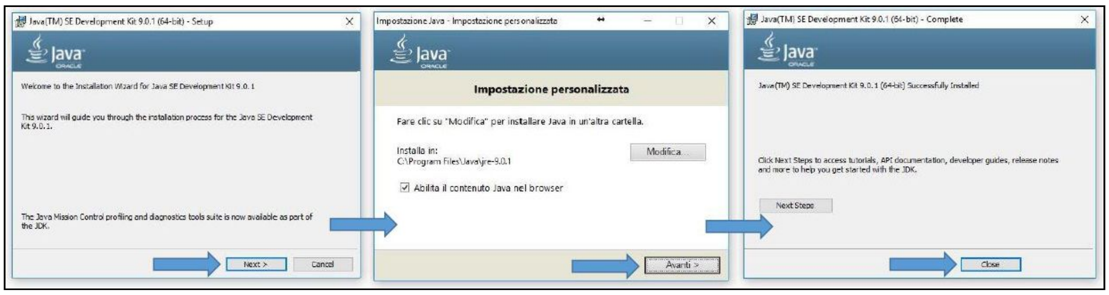


Figure 9.1.4: Neo4j Desktop Edition

For every problem with the JVM, a manual installation can fix everything:

1. Download and install the latest version available (*32 bit* or *64 bit* upon the platform) of *Oracle Java* or *Open JDK*[153] *(Figure 9.1.5)*.



Java(TM) SE Development Kit 9.0.1 (64-bit) - Setup

Welcome to the Installation Wizard for Java SE Development Kit 9.0.1

This wizard will guide you through the installation process for the Java SE Development Kit 9.0.1.

The Java Mission Control profiling and diagnostics tools suite is now available as part of the JDK.

Next > Cancel

Impostazione Java - Impostazione personalizzata

Impostazione personalizzata

Fare clic su "Modifica" per installare Java in un'altra cartella.

Installa in:
C:\Program Files\Java\jre-9.0.1

Modifica...

☑ Abilita il contenuto Java nel browser

Avanti >

Java(TM) SE Development Kit 9.0.1 (64-bit) - Complete

Java(TM) SE Development Kit 9.0.1 (64-bit) Successfully Installed

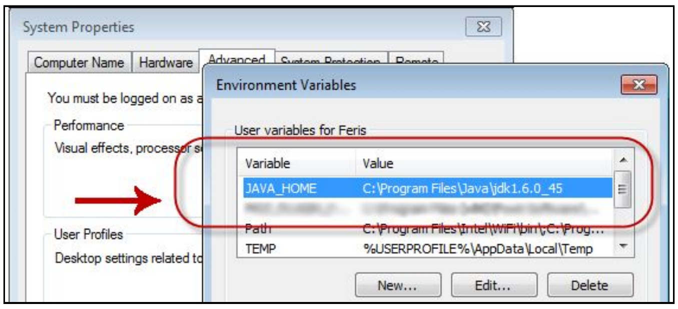
Click Next Steps to access tutorials, API documentation, developer guides, release notes and more to help you get started with the JDK.

Next Steps

Close

Figure 9.1.5: Manual Oracle Java installation

2. Check *Environment Variables* if the *JAVA_HOME* variable is set *(F. 9.1.6)*



System Properties

Computer Name | Hardware | Advanced | System Protection | Remote

You must be logged on as a

Performance
Visual effects, processor s

User Profiles
Desktop settings related to

Environment Variables

User variables for Feris

Variable	Value
JAVA_HOME	C:\Program Files\Java\jdk1.6.0_45
[redacted]	[redacted]
Path	C:\Program Files\Intel\WiFi\bin\;C:\Prog...
TEMP	%USERPROFILE%\AppData\Local\Temp

New... Edit... Delete

Figure 9.1.6: Environmental Variables [Windows]

153 http://openjdk.java.net/

9.2) Installing Neo4j on Linux/Unix

In the previous section, we have installed the Neo4j on Windows, consequently, now we install it on the operating system *Ubuntu*[154].

All the information about the installation of Neo4j, even on other operating systems, can be found on the official manual[155].

Neo4j requires the *Java runtime* which is not included in a clean Ubuntu version. The installation is very simple and almost automatic because the needed code is only:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

Another way to install Java is:

```
sudo apt-get install
            default-jre
```

or

```
sudo apt-get install
                  default-jdk
```

The first command will install the *Java Runtime Environment (JRE)*, the other the *Java Development Kit (JDK)*, which is usually needed to compile Java applications[156].

On Linux/Unix, there are two different ways to install Neo4j. The first and simple way allows the installation of both editions, the other only the Enterprise.
The easier and almost semi-automatic method is:

1. First, is required a new Repository:

```
wget -O - https://debian.neo4j.org/neotechnology.gpg.key |
    sudo apt-key add - echo 'deb http://debian.neo4j.org/repo
      stable/' | sudo tee -a /etc/apt/sources.list.d/neo4j.list
sudo apt-get update
```

2. Later, Neo4j can be installed with these commands where is specified the edition and the version required:

```
sudo apt-get install neo4j=3.x                        or
```

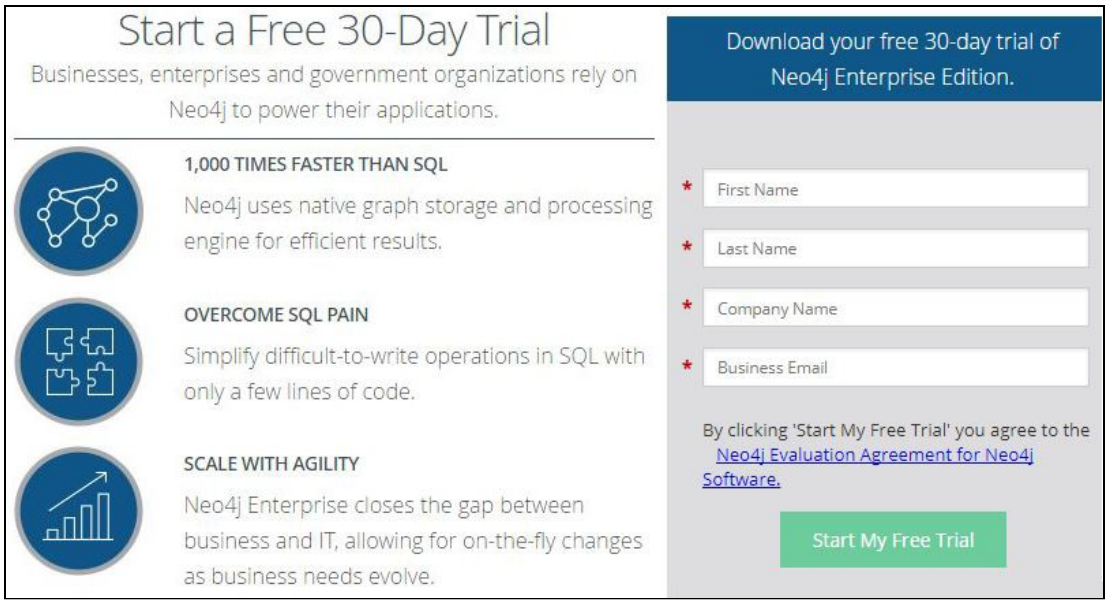
[154] https://www.ubuntu.com/ version 16.04 LTS
[155] http://neo4j.com/docs/operations-manual/current/installation/
[156] https://www.javatpoint.com/difference-between-jdk-jre-and-jvm

```
sudo apt-get install neo4j-enterprise=3.x
```

Make sure to use always the latest and stable version.

The other way to install the Enterprise edition requires more effort:

1. The download needs a Business Subscription[157] which allows exploiting the full potential of Neo4j. As specified before[158], each test showed on this dissertation has been executed on that version with an Educational License obtained for free.
   It is possible to require a 30-day trial licence to try all its potential *(F. 9.2.1)*.



Start a Free 30-Day Trial

Businesses, enterprises and government organizations rely on Neo4j to power their applications.

1,000 TIMES FASTER THAN SQL

Neo4j uses native graph storage and processing engine for efficient results.

OVERCOME SQL PAIN

Simplify difficult-to-write operations in SQL with only a few lines of code.

SCALE WITH AGILITY

Neo4j Enterprise closes the gap between business and IT, allowing for on-the-fly changes as business needs evolve.

Download your free 30-day trial of Neo4j Enterprise Edition.

By clicking 'Start My Free Trial' you agree to the Neo4j Evaluation Agreement for Neo4j Software.

Figure 9.2.1: 30-day Trial Licence

2. Once the installation (*.tar*) has been downloaded, open the terminal/shell[159].
3. Extract the contents of the archive, using:

```
tar -xf neo4j-enterprise-<VERSION>-unix.tar.gz
```

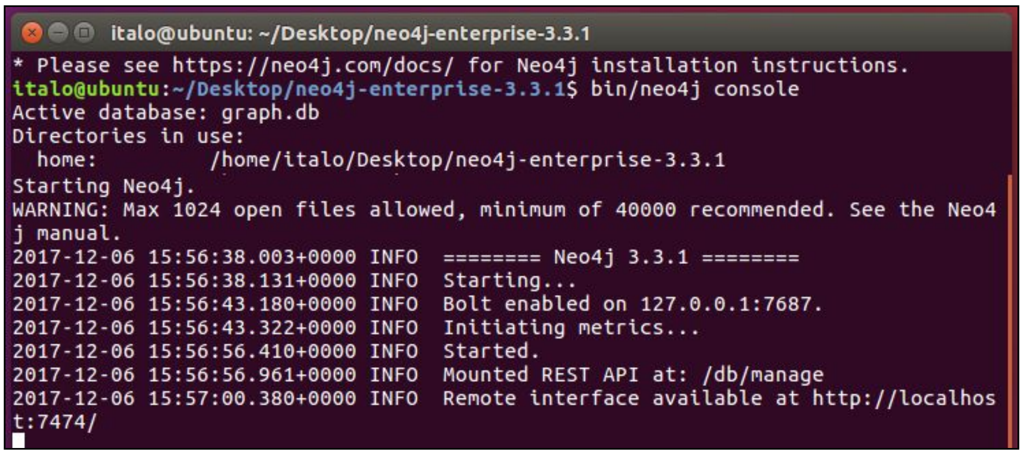
4. The top-level directory is referred with the *NEO4J_HOME* variable.
   Run Neo4j using the commands below to open the console *(Figure 8.2.2)* or to start only the server process in background.:

```
$NEO4J_HOME/bin/neo4j console
```

or

```
$NEO4J_HOME/bin/neo4j
```

[157] https://neo4j.com/business-subscription/
[158] Section 3.1.14
[159] https://distrowatch.com/table.php?distribution=linuxconsole



```
italo@ubuntu: ~/Desktop/neo4j-enterprise-3.3.1
* Please see https://neo4j.com/docs/ for Neo4j installation instructions.
italo@ubuntu:~/Desktop/neo4j-enterprise-3.3.1$ bin/neo4j console
Active database: graph.db
Directories in use:
  home:         /home/italo/Desktop/neo4j-enterprise-3.3.1
Starting Neo4j.
WARNING: Max 1024 open files allowed, minimum of 40000 recommended. See the Neo4
j manual.
2017-12-06 15:56:38.003+0000 INFO  ======== Neo4j 3.3.1 ========
2017-12-06 15:56:38.131+0000 INFO  Starting...
2017-12-06 15:56:43.180+0000 INFO  Bolt enabled on 127.0.0.1:7687.
2017-12-06 15:56:43.322+0000 INFO  Initiating metrics...
2017-12-06 15:56:56.410+0000 INFO  Started.
2017-12-06 15:56:56.961+0000 INFO  Mounted REST API at: /db/manage
2017-12-06 15:57:00.380+0000 INFO  Remote interface available at http://localhos
t:7474/
```

Figure 9.2.2: Neo4j Console starting

5. Visit with a web browser *http://localhost:7474* or *http://127.0.0.1:7474* and connect using the username '*neo4j*' with default password '*neo4j*' *(Figure 9.2.3)*.
   During the first execution, the system will ask for a new password.
   Neo4j is now ready to store data.

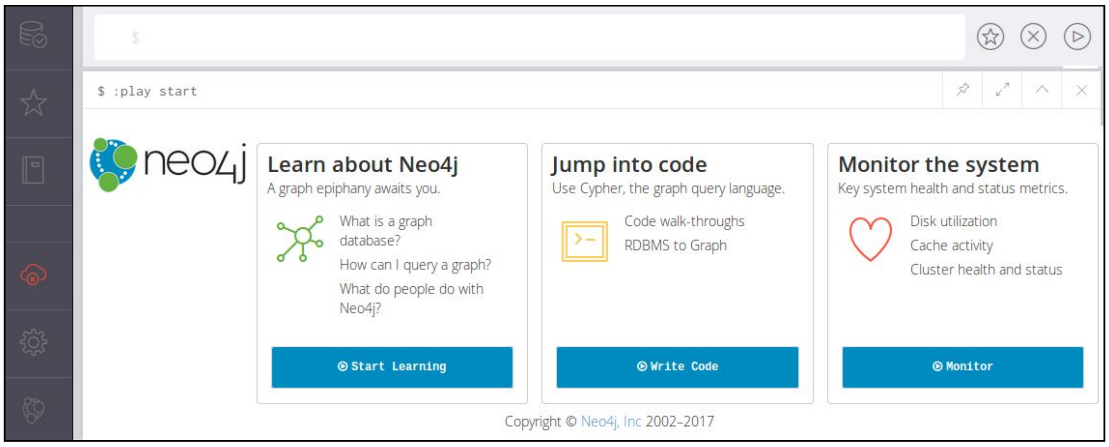

$
$:play start
neo4j
Learn about Neo4j
A graph epiphany awaits you.
What is a graph database?
How can I query a graph?
What do people do with Neo4j?
Start Learning
Jump into code
Use Cypher, the graph query language.
Code walk-throughs
RDBMS to Graph
Write Code
Monitor the system
Key system health and status metrics.
Disk utilization
Cache activity
Cluster health and status
Monitor
Copyright © Neo4j, Inc 2002-2017

Figure 9.2.3: Neo4j HTTP Console

In every moment, to show the current status of the Neo4j Server is possible to execute the command:

```
$NEO4J_HOME/bin/neo4j status
```

9.3) Neo4j with Docker

Docker[160] *(Figure 9.3.1)* is a software technology able to automate the deployment of applications within software containers.

It is a shipping container system[161] for software code that provides an additional layer of abstraction and automation at the level of operating system virtualization on Linux.

In the real world, containers protect their content and allow to carry, all together, what we need to transport. Through Docker, the container maintains source code, database and other relevant elements together, to offer a service on every platform it runs.

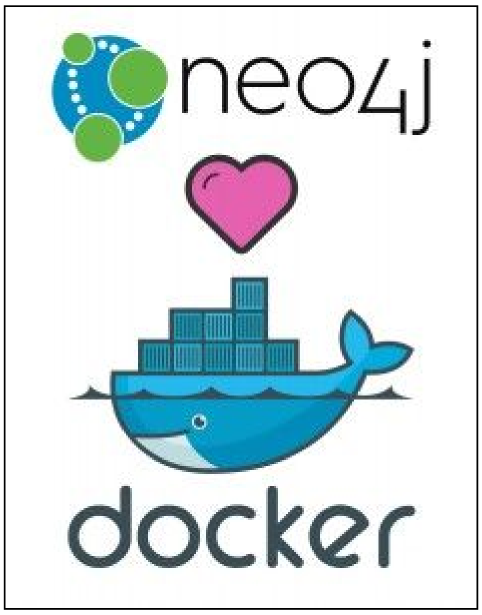


Figure 9.3.1: Neo4j LOVES Docker

There are several guides on the Internet[162], or the official manual[163], to install Docker, with all the needed information. Docker is available for MAC, Windows PC, the most famous Linux distributions and even for Raspberry PI[164].

Docker needs a specific image to run an application or an operating system[165]. Images can be downloaded from several stores, like the Hub Dock Store[166], where is presented the official Neo4j image[167], created by Neo Technology, Inc.[168]

After the installation of Docker, the needed command[169] to install Neo4j is:

```
docker pull neo4j
```

The Docker system will automatically download every needed file.

[160] https://www.docker.com/
[161] https://en.wikipedia.org/wiki/Containerization
[162] https://goo.gl/ZVCyH3
[163] https://docs.docker.com/engine/installation/
[164] https://blog.hypriot.com/getting-started-with-docker-on-your-arm-device/
[165] https://docs.docker.com/get-started/
[166] https://store.docker.com/
[167] https://hub.docker.com/_/neo4j/
[168] https://neo4j.com/developer/docker/
[169] Sometimes the command “sudo” is required to obtain root privileges.

After pulling the image in Docker, it will be ready to run. Neo4j requires two folders to store its data:

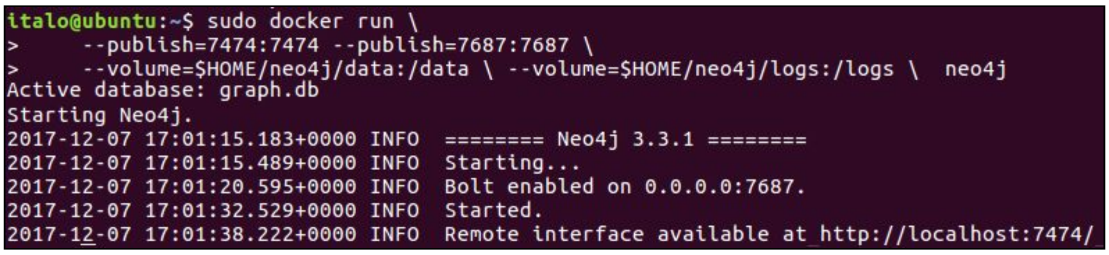
- **/data** to allow the database to be persisted inside or outside its container;
- **/logs** to allow access to Neo4j log files.

The command [153] to create these folders and to run the image is:

```
docker run \
    --publish=7474:7474 --publish=7687:7687 \
    --volume=$HOME/neo4j/data:/data \
    --volume=$HOME/neo4j/logs:/logs \
    neo4j
```

Once Neo4j is started *(Figure 9.3.2)*, pointing the browser at http://localhost:7474 will show the same interface as before.

The Neo4j image is now ready to be moved and run on every required platform, without causing problems and data losses.



```
italo@ubuntu:~$ sudo docker run \
>     --publish=7474:7474 --publish=7687:7687 \
>     --volume=$HOME/neo4j/data:/data \ --volume=$HOME/neo4j/logs:/logs \  neo4j
Active database: graph.db
Starting Neo4j.
2017-12-07 17:01:15.183+0000 INFO  ======== Neo4j 3.3.1 ========
2017-12-07 17:01:15.489+0000 INFO  Starting...
2017-12-07 17:01:20.595+0000 INFO  Bolt enabled on 0.0.0.0:7687.
2017-12-07 17:01:32.529+0000 INFO  Started.
2017-12-07 17:01:38.222+0000 INFO  Remote interface available at http://localhost:7474/
```

Figure 9.3.2: The Neo4j execution within Docker

9.4) How to interact with Neo4j

9.4.1) Neo4j REST API

After starting the Neo4j server, the interaction with the database, through the Neo4j REST API, is possible both by the HTTP console or by applications created ad-hoc for our purposes. [3][7][11]

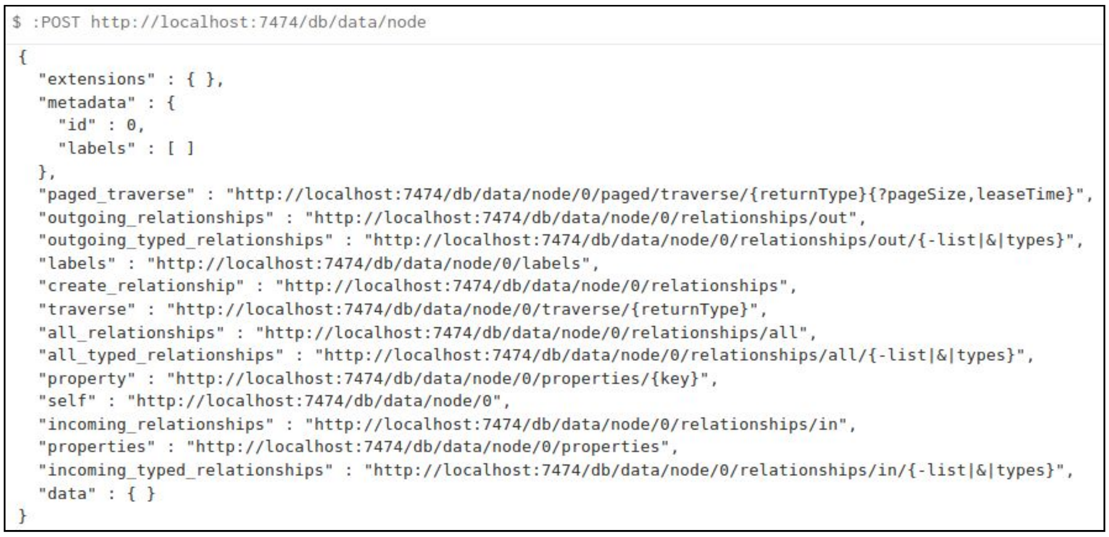
9.4.1.1) REST API by the HTTP API console

By using the HTTP API console in the Neo4j Browser, every REST call has to be executed putting a colon (**:**) just before the command. All commands consist of a keyword and a URL which indicate the resource we need to receive or create.

The traditional operations which can be performed in Neo4j using the REST API are:

- **Create an empty node**: This command creates an empty node, with no properties, except for a reference ID (0 in *Figure 9.4.1*) to itself.

```
:POST http://localhost:7474/db/data/node
```



```
$ :POST http://localhost:7474/db/data/node
{
  "extensions" : { },
  "metadata" : {
    "id" : 0,
    "labels" : [ ]
  },
  "paged_traverse" : "http://localhost:7474/db/data/node/0/paged/traverse/{returnType}{?pageSize,leaseTime}",
  "outgoing_relationships" : "http://localhost:7474/db/data/node/0/relationships/out",
  "outgoing_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/out/{-list|&|types}",
  "labels" : "http://localhost:7474/db/data/node/0/labels",
  "create_relationship" : "http://localhost:7474/db/data/node/0/relationships",
  "traverse" : "http://localhost:7474/db/data/node/0/traverse/{returnType}",
  "all_relationships" : "http://localhost:7474/db/data/node/0/relationships/all",
  "all_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/0/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/0",
  "incoming_relationships" : "http://localhost:7474/db/data/node/0/relationships/in",
  "properties" : "http://localhost:7474/db/data/node/0/properties",
  "incoming_typed_relationships" : "http://localhost:7474/db/data/node/0/relationships/in/{-list|&|types}",
  "data" : { }
}
```

Figure 9.4.1: The output of the create empty node command

- **Create a node with some properties**: Like before, the command creates a new node, but by passing an additional *JSON object*[170], it will set all the properties inside.

```
:POST http://localhost:7474/db/data/node
          {"name":"Italo", "{PROP2}":"{VAL2}"}
```

- **Create a relationship**: The command creates a relationship with a specific type between the two given nodes.

```
:POST
 http://localhost:7474/db/data/node/{ID1}/relationships
    {"to" : "http://localhost:7474/db/data/node/{ID2}",
        "type" : "{RELATIONSHIP}"}
```

[170] https://www.w3schools.com/js/js_json_objects.asp

- **Create a relationship with some properties:**

```
:POST
 http://localhost:7474/db/data/node/{ID1}/relationships
  {"to" : "http://localhost:7474/db/data/node/{ID2}",
   "type" : "{RELATIONSHIP}", "data":{"{PROP1}":"{VAL1}",
       "{PROP2}":"{VAL2}"}}
```

- **Read a node/relationship**: Once we have the element ID, to read all its information the command is:

```
:GET http://localhost:7474/db/data/node/{ID}
```

```
:GET http://localhost:7474/db/data/relationship/{ID}
```

- **Read only the property of a node/relationship**:

```
:GET http://localhost:7474/db/data/node/{ID}/properties
```

```
:GET http://localhost:7474/db/data/relationship/
                                         {ID}/properties
```

- **Add a new property to a node/relationship:**

```
:PUT http://localhost:7474/db/data/node/
                         {ID}/properties/{PROP} {VALUE}
```

```
:PUT http://localhost:7474/db/data/relationship/
                         {ID}/properties/{PROP} {VALUE}
```

- **Delete a relationship**: Like nodes, every relationship has an ID, and it can be used to delete the relationship or change its properties.

```
:DELETE http://localhost:7474/db/data/relationship/{ID}
```

- **Delete a node**: Nodes without relationships can be deleted using the following command

```
:DELETE http://localhost:7474/db/data/node/{ID}
```

However, when a node has relationships, it cannot be deleted directly. First, we have to remove all its relationships and then we can delete it.

- **Delete a specific property**:

```
:DELETE http://localhost:7474/db/data/node/
                              {ID}/properties/{PROPERTY}
```

```
:DELETE http://localhost:7474/db/data/relationship/
                              {ID}/properties/{PROPERTY}
```

- **Get all the relationship types**:

```
:GET http://localhost:7474/db/data/relationship/types
```

- **See the relationships on a node:**

All	`:GET http://localhost:7474/db/data/node/{ID}/relationships/all`
Only the incoming	`:GET http://localhost:7474/db/data/node/{ID}/relationships/in`
Only the outgoing	`:GET http://localhost:7474/db/data/node/{ID}//relationships/out`

Other commands with examples can be found on the official manual regarding the Neo4j REST API[171].

[171] https://neo4j.com/docs/rest-docs/current/

9.4.1.2) REST API by Java

Neo4j offers a rich set of integration possibilities for Java, like some REST libraries which can be used in any JVM language.

Although there are multitudes of REST clients in Java used to connect with Neo4j, in this section will be presented a basic client that creates a node with some properties and then displays these properties on the screen, everything through REST APIs. As specified before, the first request is a POST, the other a GET, and both commands make use of JSON objects to store the properties.

```java
import org.apache.commons.io.IOUtils;
import org.apache.http.HttpResponse;
import org.json.JSONObject;
[.....] //Other imports

public class test {
 public static void main(String[] args) throws
                                            JSONException {
   try {
     //creating a client to execute our REST requests
     HttpClient client = HttpClientBuilder.create().build();

      //the ENDPOINT + RESOURCE to create a new node
     // username = neo4j, password = italo
     String postUrl =
          "http://neo4j:italo@localhost:7474/db/data/node";

     //creating a JSON object with properties for the node
     JSONObject jsonInput = new JSONObject();
     jsonInput.put("name", "Joanna");
     jsonInput.put("age", "27");
     StringEntity params = new
                    StringEntity(jsonInput.toString());

     //Creating the request object to execute the POST
     HttpPost requestPost = new HttpPost(postUrl);
     requestPost.addHeader("content-type",
                                    "application/json");
     requestPost.setEntity(params);
```

```
      //Executing the request
      HttpResponse response = client.execute(requestPost);

      //Extracting the ID of the new node
      String json =
          IOUtils.toString(response.getEntity().getContent(),
               "UTf-8");
      JSONObject obj = new JSONObject(json);
      JSONObject obj2 = (JSONObject) obj.get("metadata");
      int IDNewNode = (int) obj2.get("id");

      //The URL of the new node
      String getUrl =
          "http://neo4j:italo@localhost:7474/db/data/node/"
                 +IDNewNode;

      //Creating the request object to execute the GET
      HttpGet requestGet = new HttpGet(getUrl);

      //Executing the request
      response = client.execute(requestGet);

      //Showing the answer
      json =
          IOUtils.toString(response.getEntity().getContent(),
                "UTf-8");
      obj = new JSONObject(json);
      System.out.println("The node ID "+IDNewNode
               +" has these prop.: " + obj.get("data"));
     }
     catch (IOException ex) {
         //Exception, do something
     }
   }
}
```

The output of the code will change at every execution because the ID is assigned dynamically, but it will be similar to this:

```
The node ID 93 has these prop.: {"name":"Joanna","age":"27"}
```

9.4.2) Accessing Neo4j from Python

Python[172], released in 1991, is a high-level programming language for general-purpose programming. During the years, the developer community has realised lots of Neo4j clients available for this language, and some of them are officially supported at the moment[173].

One of the simplest Python clients is the *Neo4jRestClient*[174], whose syntax is fully compatible with Python-embedded, created to use the Neo4j REST Server.
All the features of this client are listed in its official manual[175].

The installation is automatic but requires *pip* (*Python Package Manager)*[176], a tool for installing and managing Python packages, and a Python Interpreter.

If they are not installed, the needed commands are:

```
apt-get update
sudo apt-get install python
sudo apt-get -y install python-pip
```

The client installation now is possible using

```
sudo pip install neo4jrestclient
```

The following source code creates a graph where are presented people and animals, connected by several relationships, to indicate the feeling that the person feels towards the animal.

```
from neo4jrestclient.client import GraphDatabase

# The database connection
db = GraphDatabase("http://localhost:7474",
                   username="neo4j", password="italo")

# Creating nodes with the label Person
person = db.labels.create("Person")
```

[172] https://en.wikipedia.org/wiki/Python_(programming_language)
[173] https://neo4j.com/developer/python/
[174] https://pypi.python.org/pypi/neo4jrestclient/
[175] http://neo4j-rest-client.readthedocs.io/en/latest/
[176] https://en.wikipedia.org/wiki/Pip_(package_manager)

```
p1 = db.nodes.create(name="Marta")
p2 = db.nodes.create(name="Anna")   [....]
person.add(p1, p2, p3, p4)

# Creating nodes with the label Animal
animal = db.labels.create("Animal")
a1 = db.nodes.create(name="Dog")
a2 = db.nodes.create(name="Lion")
a3 = db.nodes.create(name="Frog")
animal.add(a1, a2, a3)

# Creating relationships between the nodes
p1.relationships.create("likes", a1)
p2.relationships.create("likes", a2)
p4.relationships.create("is_scared_of", a3) [....]
```

By executing the code, the created graph will be *(Figure 9.4.2)*:

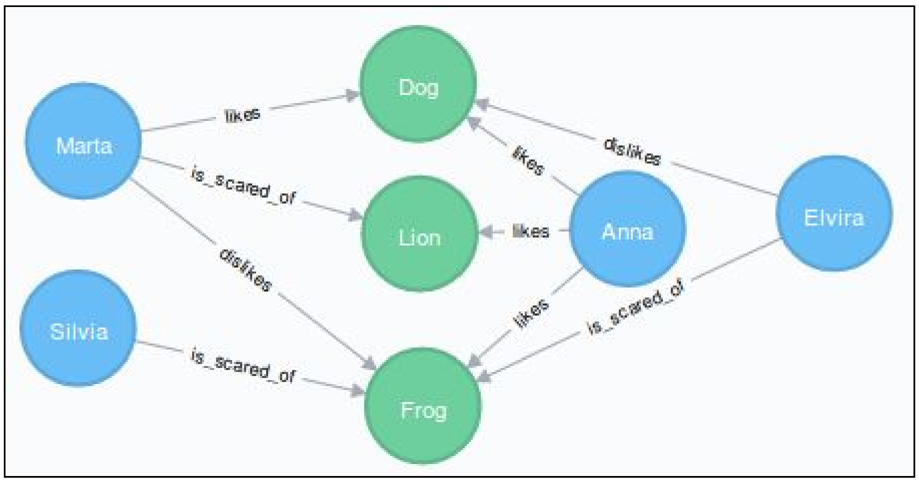


Figure 9.4.2: A graph about people and animals

Another significant functionality offered by the client is the possibility to embed Cypher into a Python code:

```
from neo4jrestclient.client import GraphDatabase
from neo4jrestclient import client

db = GraphDatabase("http://localhost:7474",
                        username="neo4j", password="italo")
```

```
q = "MATCH (p:Person {name: 'Elvira'}) - [r] -> (a:Animal)
     RETURN p, type(r), a"

results = db.query(q, returns=(client.Node,
                                  str, client.Node))
for r in results:
    print("(%s)-[%s]->(%s)" %
                       (r[0]["name"], r[1], r[2]["name"]))
```

The output of this code is:

```
(Elvira)-[is_scared_of]->(Frog)
(Elvira)-[dislikes]->(Dog)
```

9.4.2) Accessing Neo4j from JavaScript

The Neo4j JavaScript driver[177] [178], officially supported by Neo4j[179], connects to the database using the binary protocol.

NPM[180], the package manager for the JavaScript programming language, is required to install the driver. To install them, the necessary code is:

```
sudo apt-get install npm
npm install neo4j-driver
```

The execution of our JavaScript code requires a Node.js[181] interpreter installable through:

```
sudo apt-get install nodejs
```

The following portion of code, using the Neo4j JavaScript driver, connects to the database and creates a node with some properties. To show the correctness of the creation, the output will be the result of the Cypher query.

[177] http://neo4j.com/docs/developer-manual/current/drivers/
[178] https://www.npmjs.com/package/neo4j-driver
[179] https://neo4j.com/developer/javascript/
[180] https://github.com/npm/npm
[181] https://nodejs.org/

```
// Connecting to the database create a driver instance
const neo4j = require('neo4j-driver').v1;
const driver = neo4j.driver
   ("bolt://localhost", neo4j.auth.basic("neo4j", "italo"));

// Creating a session to run Cypher statements in
const session = driver.session();

// Defining and creating a Person
const personName = 'Joanna';
const personAge = '27';


//Executing the CYPHER statement
const resultPromise = session.run(
  'CREATE (a:Person {name: $name, age: $age}) RETURN a',
  {name: personName, age: personAge}
);

//Receiving and elaborating the result
resultPromise.then(result => {
  session.close();

  const singleRecord = result.records[0];
  const node = singleRecord.get(0);

  //Showing the result
  console.log("The node with name "+node.properties.name
                              +" was created");

  driver.close();
});
```

The output of this code will be:

```
The node with name Joanna was created
```

9.5) Batch data imports

The creation of a database, usually, can take much time, due to the amount of data that has to be stored. The process of batch import[182] can help during the creation of our database, allowing the data import faster than manual queries. [3][7]

Neo4j offers four different processes for importing data:

- CSV importer
- The spreadsheet (Excel) importer
- HTTP batch imports with REST API
- Java API

9.5.1) CSV importer

A **CSV** (**Comma-separated values**[183]) file stores data in plain text, following some rules:

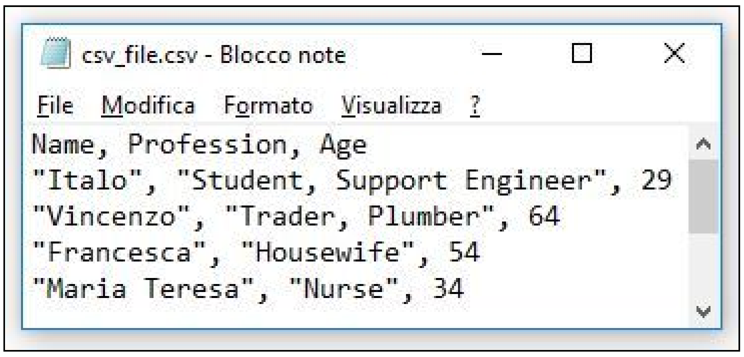
- Every row is a data record;
- Every data record consists of one or more fields, separated by commas;
- Every data record can store numbers or strings;
- Double quotes (**"**) are used to string quotation;
- The CSV file can have headers, but they are not required;
- The character encoding should be **UTF-8**[184].

A CSV file, for its characteristics, can be useful to store data for fast reading and to import them into the database. However, for its simplicity, there are no controls on duplicate values.

As explained before[185], Cypher offers the *CREATE* and *MERGE* clauses to create new data in the database. Consequently, the CSV import supports both commands. The first clause creates new elements each time it is executed, the other, considered a combination of MATCH and CREATE, first try to bind data, executing a MATCH on the graph, and if it finds something, instead of creating a new element, it uses the old one. The first clause is faster than the other, but, the second ensures the absence of duplicates at the cost of higher execution time.

[182] https://neo4j.com/blog/bulk-data-import-neo4j-3-0/
[183] https://datahub.io/docs/data-packages/csv
[184] https://www.w3schools.com/charsets/ref_html_utf8.asp
[185] Section 5.2

For example, considering a CSV where are stored people and their professions, a possible file structure could be (*Figure 9.5.1)*:


csv_file.csv - Blocco note
File Modifica Formato Visualizza ?

```
Name, Profession, Age
"Italo", "Student, Support Engineer", 29
"Vincenzo", "Trader, Plumber", 64
"Francesca", "Housewife", 54
"Maria Teresa", "Nurse", 34
```

Figure 9.5.1: Typical structure of a CSV file

The correct steps to import that CSV file into Neo4j, considering the using of the MERGE statement, are:

1. Start the server and open a new *Command Prompt*[186], executing the command:

```
$NEO4J_HOME/bin/neo4j-shell
```

This line shows the shell prompt, where many commands can be executed to manage the database.

2. Execute the LOAD CSV command with the option "WITH HEADERS":

```
LOAD CSV WITH HEADERS FROM "$NEO4J_HOME/csv_file.csv"
    as csv
        MERGE (person:Person {name : csv.Name} )
            ON CREATE SET person.Profession =
                        split (csv.Profession, "," )
            ON CREATE SET person.Age = csv.Age;
```

When the CSV file has no headers, the name can be replaced putting the column number within **[ ]**, like an array (for example *csv[1]*).

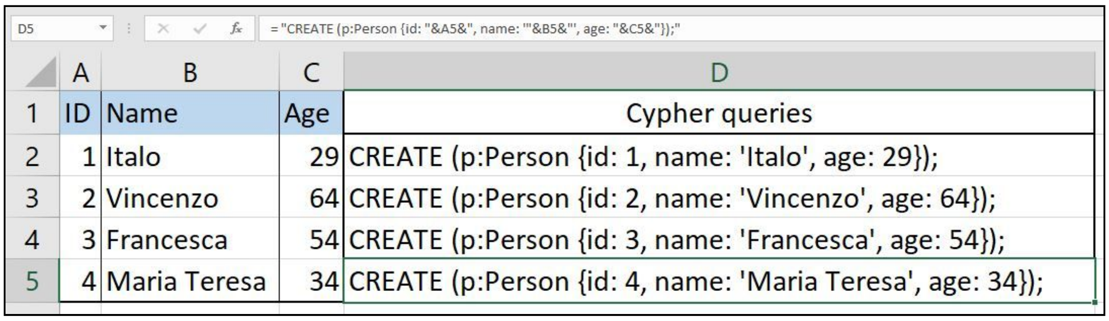
[186] https://neo4j.com/docs/operations-manual/current/tools/cypher-shell/

9.5.2) The spreadsheet (Excel) importer

Although Neo4j does not provide direct support for **Microsoft Excel**[187] documents, they can be used to create a script document with dynamic Cypher queries.

Considering an Excel sheet with three columns (*ID, Name and Age*) *(Figure 9.5.2)*, putting the following formula[188] into the next free column, will generate Cypher queries automatically:

```
= "CREATE (p:Person {id: "&A2&",
                              name: '"&B2&"', age: "&C2&"});"
```



D5 = "CREATE (p:Person {id: "&A5&", name: '"&B5&"', age: "&C5&"});"

	A	B	C	D
1	ID	Name	Age	Cypher queries
2	1	Italo	29	CREATE (p:Person {id: 1, name: 'Italo', age: 29});
3	2	Vincenzo	64	CREATE (p:Person {id: 2, name: 'Vincenzo', age: 64});
4	3	Francesca	54	CREATE (p:Person {id: 3, name: 'Francesca', age: 54});
5	4	Maria Teresa	34	CREATE (p:Person {id: 4, name: 'Maria Teresa', age: 34});

Figure 9.5.2: An Excel sheet

All those queries can be used to create a batch import file with the .txt[189] extension:

1. Create a new text file;
2. Write the keyword *BEGIN* at the top of the file;
3. Copy all the cypher queries;
4. Add *COMMIT* to the end of the file:
5. Save the file with the import.txt name into the *$NEO4J_HOME* directory;
6. Stop the Neo4j server;
7. Open a console and execute the command:

```
cat import.txt | $NEO4J_HOME/bin/neo4j-shell
                  -config conf/neo4j.properties
                               -path $NEO4J_HOME/data/graph.db
```

8. Start the Neo4j server, and all the queries will be executed automatically.

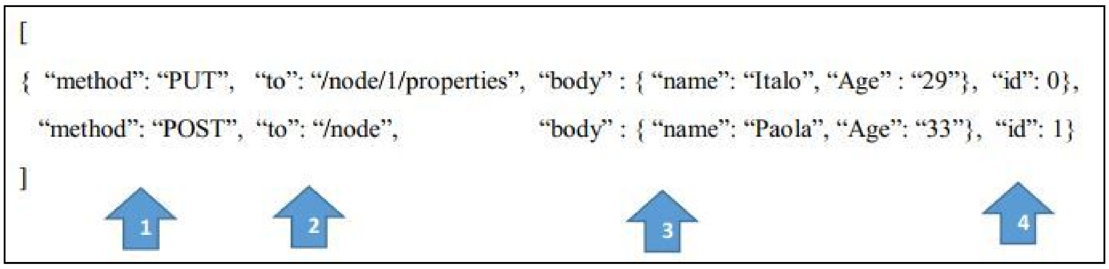
[187] https://g.co/kgs/QwrVuW
[188] https://goo.gl/HMEoqJ
[189] https://en.wikipedia.org/wiki/Text_file

9.5.3) HTTP batch imports with REST API

Neo4j REST architecture is designed to expose the service root to users who know the URIs to perform various operations. A URI is composed of a relative *ENDPOINT* (*http://<HOST>:<PORT>/db/data*) and a *RESOURCE* which indicate the element what we are working on.

To perform batch operations, Neo4j REST exposes the endpoint */batch/*, which improves performance for a large number of operations.

The API expects a list of job descriptions as input, where are defined all the required operations. All the job descriptions are executed in a single transaction, and the rollback is supported in case of any error.



```
[
{ "method": "PUT",   "to": "/node/1/properties",  "body" : { "name": "Italo", "Age" : "29"},  "id": 0},
  "method": "POST",  "to": "/node",               "body" : { "name": "Paola", "Age": "33"},  "id": 1}
]
```

Figure 9.5.3: Array of job descriptions for Neo4j REST batch imports

Considering the example in the *Figure 9.5.3*, each job description has four elements:

1. **method**: defines the type of operation (usually *GET, PUT, POST* and *DELETE*);
2. **to**: specifies the target URI where the request will be submitted;
3. **body**: the optional attribute for sending parameters;
4. **id**: defines the unique ID of each job.

In the example, the first job description adds new properties into a defined node (with the *ID 1*); the second creates a new node with other properties.

9.5.4) Java API

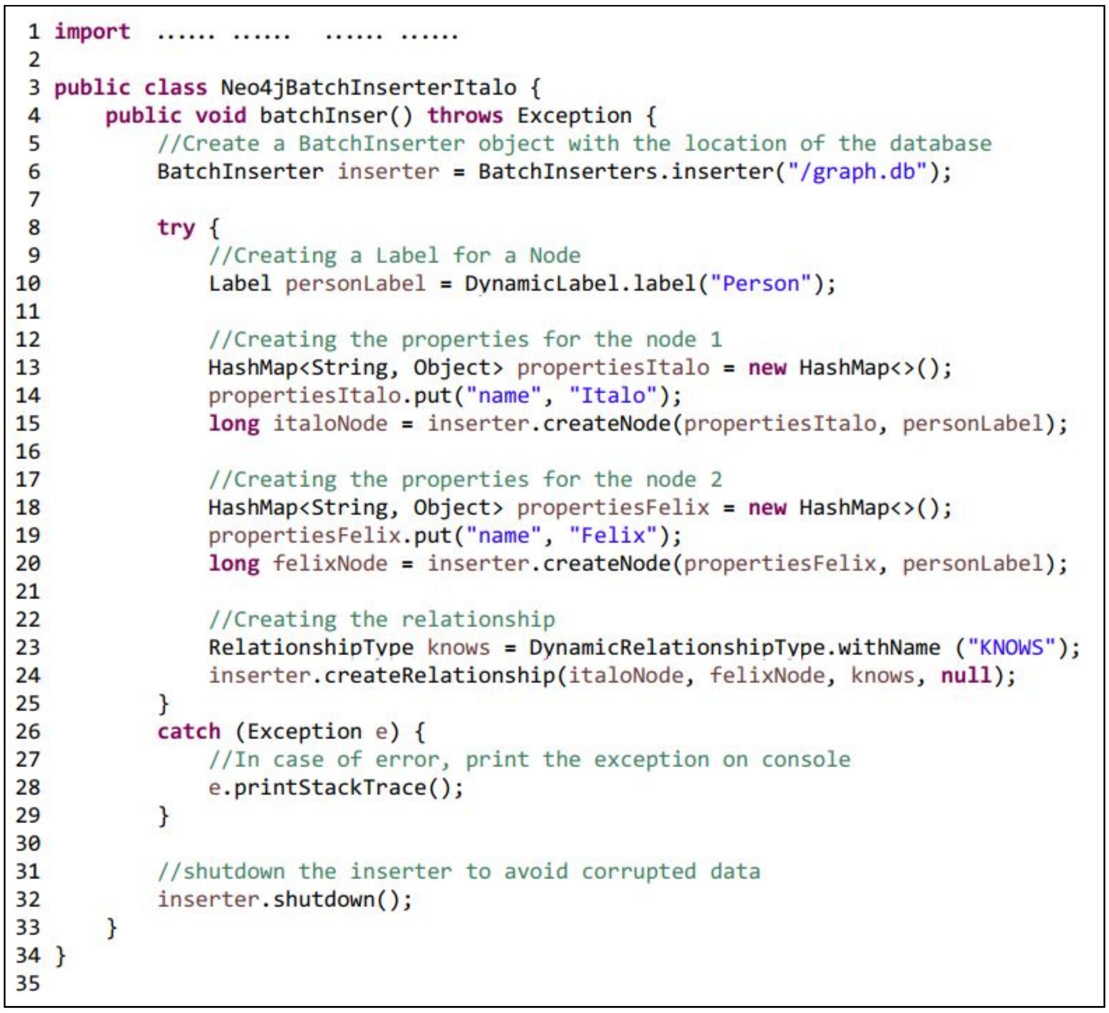
Neo4j exposes a low-level Java API, called BatchInserter[190], which helps during the insertion of data. There are some limitation using this API:

- Its priority is only performance, in consequence, there is no transaction support, is not thread safe and the request method type is only *POST*.

[190] https://goo.gl/nR9h5j

- The execution is allowed only in a single thread[191], so concurrent access to the API is restricted.
- The database must be stopped before the execution, to avoid corrupted data.

The following code *(Figure 9.5.4)* creates two nodes and a relationship using the Java API:



```java
import  ...... ......  ...... ......

public class Neo4jBatchInserterItalo {
    public void batchInser() throws Exception {
        //Create a BatchInserter object with the location of the database
        BatchInserter inserter = BatchInserters.inserter("/graph.db");

        try {
            //Creating a Label for a Node
            Label personLabel = DynamicLabel.label("Person");

            //Creating the properties for the node 1
            HashMap<String, Object> propertiesItalo = new HashMap<>();
            propertiesItalo.put("name", "Italo");
            long italoNode = inserter.createNode(propertiesItalo, personLabel);

            //Creating the properties for the node 2
            HashMap<String, Object> propertiesFelix = new HashMap<>();
            propertiesFelix.put("name", "Felix");
            long felixNode = inserter.createNode(propertiesFelix, personLabel);

            //Creating the relationship
            RelationshipType knows = DynamicRelationshipType.withName ("KNOWS");
            inserter.createRelationship(italoNode, felixNode, knows, null);
        }
        catch (Exception e) {
            //In case of error, print the exception on console
            e.printStackTrace();
        }

        //shutdown the inserter to avoid corrupted data
        inserter.shutdown();
    }
}

```

Figure 9.5.4: Java code to use the BatchInserter API

Although those processes are optimised to import an enormous amount of data fastly, Neo4j suffers from some constraints, which prevent, for example, the exceeding of 34 billion nodes. All these limitations have been covered in section *8.4*.

[191] https://en.wikipedia.org/wiki/Thread_(computing)

9.6) Configuring a Neo4j cluster M/S on Linux/Unix

A typical deployment of Neo4j uses a cluster Master/Slave of three machines to provide fault tolerance and read scalability. [3]

These functionalities, offered only in the Enterprise Edition, requires some settings to be activated, but fortunately, the latest versions of Neo4j[192], needs only a few changing[193].

After copying the same Neo4j version on all the machines, for each of them, is required an ID, which usually is a positive and unique integer.

Opening the configuration file *$NEO4J_HOME/conf/neo4j.conf* is shown the Neo4j configuration. The options that have to be changed are:

- **ha.server_id**: contains the unique server ID for each Neo4j instance.
  This is the only different property for every machine in the cluster.
  All other properties are the same.
- **ha.initial_hosts**: contains a list of IP and number port of the machines connected by the cluster.
  (Example *ha.initial_hosts=<Machine1>:<Port1>,<Machine2>:<Port2>*)
- **dbms.mode=HA** : indicates that the High Availability functionality is active.
  The default value for the single mode is *SINGLE*. There is another value which is *ARBITER*, used to define which machine will take part in the master elections if the master instance fail[194].
- **dbms.connector.http.enabled**: with the value *TRUE* enables connection to the server remotely.
- **dbms.connector.http.listen_address**: usually set with the number *7474*, defines the number port where the instance will receive commands.

Once the configuration file has been changed, it is possible to start all the Neo4j instances, and when all the machines are ready, the database will be available to all users. After this moment, every changing on any instance will be propagated automatically between the others.

[192] Tests performed on Neo4j version 3.3.x, Enterprise Edition on a clean Ubuntu installation
[193] https://neo4j.com/docs/operations-manual/current/tutorial/highly-available-cluster/
[194] https://goo.gl/xWbmjw

Chapter X

Case study: eBay ShopBot

eBay ShopBot[195] is a personal shopping assistant, which allows customers to converse with eBay[196], via text, voice or photo, using *Facebook Messenger*[197].

The bot[198], combining AI knowledge[199] with a natural language understanding activity of content parsing[200], helps customers finding which products are suitable for their requests, using Neo4j as the content delivery vehicle.

RJ Pittman *(Figure 10.1)*, *Senior Vice President* and *Chief Product Officer*[201] at eBay, explained that the result of the fusion between artificial intelligence and e-commerce has resulted in a highly personalised shopping assistance for everyone, called eBay ShopBot.

Figure 10.1: RJ Pittman

Its primary goal is to remove the hard work associated with shopping, to build a real-time recommendation engine[202] that understands clients' requests and increases its knowledge about the shoppers.

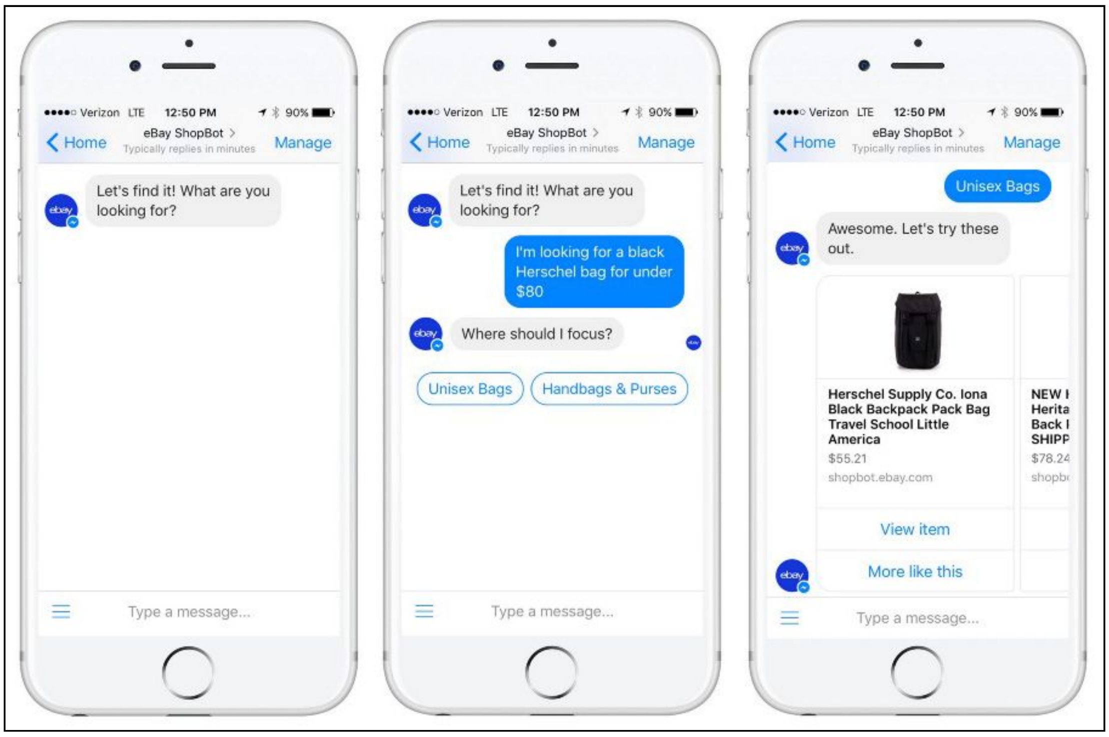
Traditional e-commerce main problem is the limited search capability. Usually, the search engine works only with boolean and simple keyword searches, losing all the additional context essential to determine the customer's real and full intent.

[195] https://shopbot.ebay.com/
[196] https://www.ebay.com/
[197] https://www.messenger.com/
[198] https://www.techopedia.com/definition/24063/internet-bot
[199] https://en.wikipedia.org/wiki/Artificial_intelligence
[200] https://en.ryte.com/wiki/Natural_Language_Processing
[201] https://en.wikipedia.org/wiki/Chief_product_officer
[202] https://neo4j.com/use-cases/real-time-recommendation-engine/

A Natural Language Processing (NLP), machine learning[203], predictive modeling[204] and a real-time storage and processing engine are the key concept to achieve these requirements.

Based on a Neo4j graph database, eight million nodes connected by twenty million relationships, the traversals are guided by simple questions, answered by the customers, creating a new kind of commerce, called **Conversational Commerce**[205].

10.1) How it works



Let's find it! What are you looking for?

I'm looking for a black Herschel bag for under $80

Where should I focus?

Unisex Bags

Handbags & Purses

Unisex Bags

Awesome. Let's try these out.

Herschel Supply Co. Iona Black Backpack Pack Bag Travel School Little America
$55.21
shopbot.ebay.com

NEW I
Herita
Back I
SHIPP
$78.24
shopb

View item

More like this

Figure 10.2: Example of a customer's request

Figure 10.2 shows a typical customer's request, where a person is looking for a black bag. Every request is analysed to extract the information required for the traversal.

In that simple sentence, the NLP is looking for every relevant word essential to identify the object of interest. By analysing the request, the extracted information is about the type of object, colour, brand and price.

[203] https://www.sas.com/en_us/insights/analytics/machine-learning.html
[204] http://searchdatamanagement.techtarget.com/definition/predictive-modeling
[205] https://goo.gl/gppAM9

All that information together with other, collected by asking questions to the customer, are used to identify the needed object, helping the user during his shopping *(Figure 10.3)*.

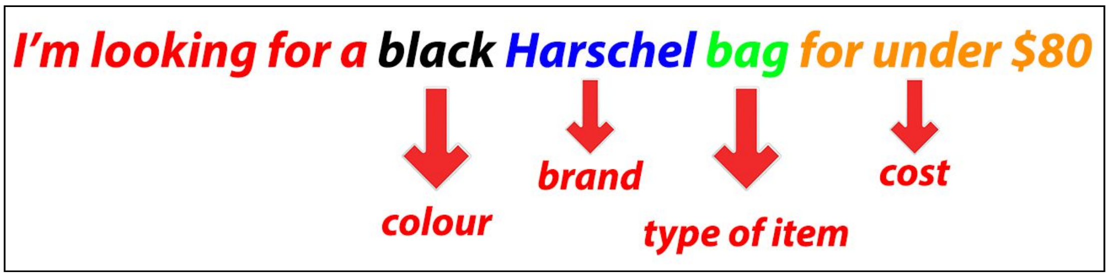


Figure 10.3: The relevant information in the customer's request

For every customer's answer, the engine performs a traversal on the eBay graph, where are listed all its products *(Figure 10.4)*.

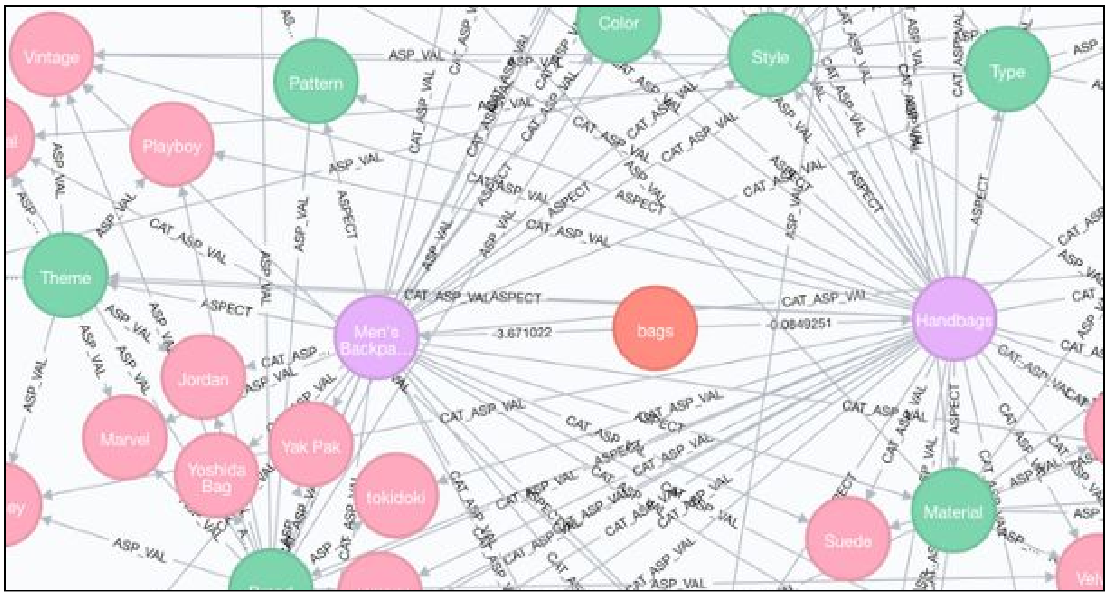


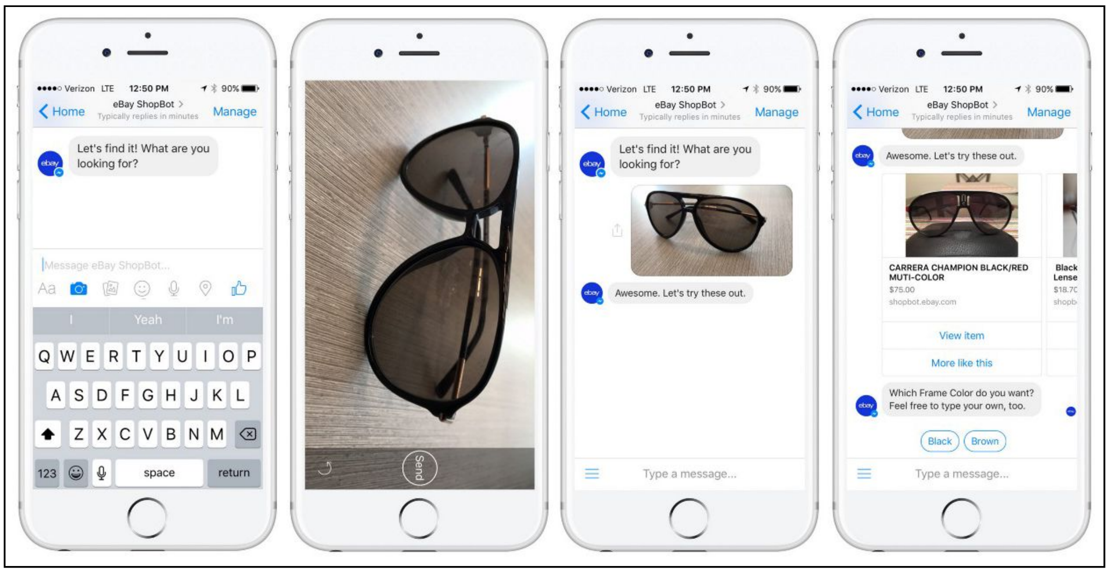
Figure 10.4: Part of the eBay graph

The first and most important node in this request is *bags*, connected with other nodes indicating the type of bag, materials, colour and so on. Every hop gets the customer close to the required object, allowing the searching only with some questions.

As mentioned before, the eBay ShopBot supports several types of requests. Another significant use case is the searching using an image. The engine, using Pattern Recognition[206] and Machine Learning, can differentiate the object in the

[206] Bishop, Christopher M; "Pattern Recognition and Machine Learning", Springer (2016)

image and convert the request in primary information to be used during the graph walking *(Figure 10.5)*.


Let's find it! What are you looking for?

Awesome. Let's try these out.

CARRERA CHAMPION BLACK/RED MUTI-COLOR
$75.00
shopbot.ebay.com

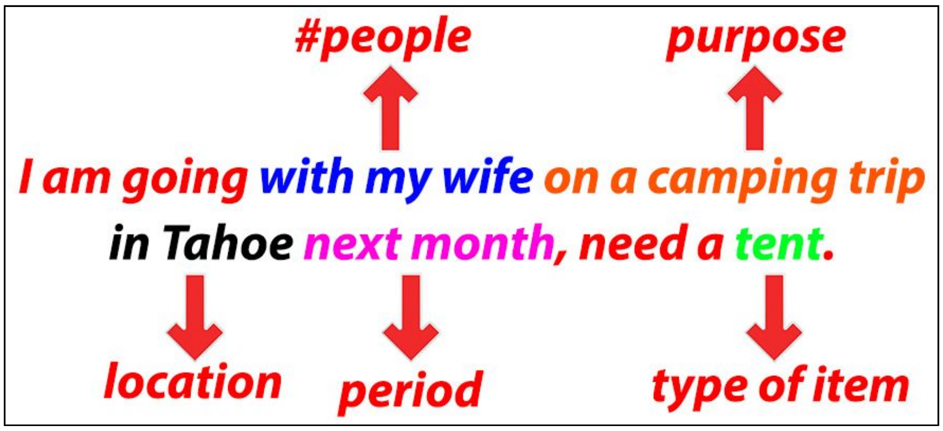
Which Frame Color do you want? Feel free to type your own, too.

Figure 10.5: Searching products using an image

The work performed by the eBay ShopBot behind each request is not limited to the extraction of the information in the sentence, but it also executes searching on external sources[207]. Considering the following request:

"I am going with my wife on a camping trip in Tahoe next month, need a tent."

The needed object is a *tent*, but the engine can extract other information, to show the perfect product for him *(Figure 10.6)*.



#people
purpose
I am going with my wife on a camping trip
in Tahoe next month, need a tent.
location
period
type of item

Figure 10.6: The analysis of another sentence

[207] https://goo.gl/xFNUfL

The request gives us information about the number of people, which can be used to choose the size of the tent, the location (*Lake Tahoe*[208]) and the period of the year. From external sources, it is possible to get the altitude of the location and the expected temperatures for the next month.

All this information together can be used to identify what tent will be suitable for the customer, using all the real power of the eBay ShopBot.

10.2) Customers preferences and characteristics

However, its functions are not only limited to the product searching. After every request, eBay ShopBot keeps track of every relevant information about the customer, to use them during the next search. Considering the following request:

"*I want to buy a baseball glove, for left-handed people, size 13",*
because I have to play a match on Saturday."

The request says that the customer is looking for a baseball glove, delivered fastly. However, at the same time, the system can store information about the fact that he is left-handed and his hand is size 13". All this information can be used in the future to increase searching performance and accuracy.

Considering another request:

"*I need a gift for my son's birthday on Monday.*
He is 7 years old, and he likes Dragon Ball."

The request could be used to define two people, the first is the customer, and the other is his child. The customer is definitely a father, then his child's birthday is an important anniversary. About the son, we can store the date of his birthday and his preference. This information can be used the next year to send a reminder email to the customer, with some possible gifts for his son.

In conclusion, the eBay ShopBot, running in Docker containers in the cloud, is a powerful recommendation tool, which based its performance and stability on the graph database technology.

All its functions are offered thanks to Neo4j, which allows the presentation of a high-quality recommendation service, confirming it as an excellent Database Management System for Graph solutions.[209] [13]

[208] https://wikitravel.org/en/Lake_Tahoe
[209] https://neo4j.com/case-studies/ebay-shopbot/

Chapter XI

Conclusion

This thesis aim has been to give an overview of Graph Database, with a particular focus on the Neo4j technology, considered at the present times, the most widespread to handle data in a graph form.

The research developed firstly with a functional and a critical comparative analysis of the relational databases, which was aimed at determining the advantages, disadvantages and characteristics that are made available by Graph Databases.

Subsequently, the work continued with a presentation of the features and functionality of Neo4j, with the aim of providing the necessary tools to evaluate the possible use of it as an alternative, in certain sectors, to relational or other NoSQL solutions.

11.1) Positive aspects

The study conducted has established that Neo4j is a tremendously impressive and promising product, fated to deepen its presence on the market in the near future.

The main features of this Graph Database Management System are:

- Full support of the ACID properties, which ensures validity in our data, even after a fail situation
- Scalability support to maintain high availability
- Advanced cache management to improve performance
- Accessible through the REST API
- Three different APIs to store and manage data
- Extensible, thanks to server extensions
- Open source, written in Java
- Online community accessible and active.

11.2) Negative aspects

1) Although the Neo4j use domain is very extensive, currently, the most famous software systems is still making use of other data storage technologies, achieving, sometimes, insufficient results regarding performance or stability.

   The reason behind this is the lack of information and promotion regarding Graph Databases, in fact, although the case study in this dissertation demonstrates excellent potential and future uses in other sectors, Neo4j is merely considered useful for Social Networks only, where the high obtained performance is not comparable to any currently existing technology.

   A demonstration of the limited academic success of this system is the scarcity of scientific articles in the most prestigious digital libraries, where other NoSQL systems are more used, even in the wrong conditions.

2) Another weakness is the lack of a universal standard, as can be the SQL.

   Every Graph Database has its APIs and its method of storing and accessing data, causing the problematic diffusion of a stable and performing language such as Cypher.

   However, in the last two years, other graph-oriented systems are adopting Cypher[210], thus demonstrating confidence in this declarative language created by Neo Technology, Inc.

3) Data security is a critical issue in any software system, because, data must reside on servers in a secure form, free from theft.

   Beside considering server security, a vulnerability that could be problematic for database administrators resides in store files being protected by authentication only.

   All store files, used to store data, can be copied and transported easily, and regrettably Neo4j does not offer any protection regarding them.

   Subsequently, they could be decrypted using brute force techniques and powerful calculators.

   Using complex passwords, could alleviate the situation, but this remains a major problem which cannot be underestimated.

4) Neo4j does not offer any scheduling tool for backups or other operations.

   Sometimes, database administrators perform planned procedures, used to execute predefined actions. However, they must rely on external tools to perform them.

[210] SAP HANAand AgensGraph.

An internal Neo4j support would allow more natural management of the scheduler, to simplify the execution of time-related actions.

5) Server extensions, used to execute Java code inside the server, are considered a powerful means of extending the functionality of our system.

   Nevertheless, at the same time, this practise is extremely dangerous and might create problems in the data.

   Improper use could result in the loss of critical data or the execution of malicious code aimed at the theft of personal information.

   Although Neo4j offers tools and guides to secure extensions[211], it does not offer advanced controls on the code to be executed, leaving the data in the system vulnerable. More controls and data protection tools would be likely to improve the usability and security of the system.

6) The use of the START clause, deprecated in the latest versions of Cypher, causes the regression to an old version of the query interpreter, producing a loss of performance and a reduction in the functions offered.

   At this time, no automatic tools offers the change of old Cypher queries into new ones, where the MATCH clause has incorporated the START clause.

   A query converter would, therefore, be a useful system for inexperienced or outdated users.

   In order to increase performance, indexes are essential. Usually, it is not necessary to specify which ones to use. However, to make sure which indexes should be applied, the USING INDEX clause is the most appropriate.

Auspiciously, the Neo4j development team is constantly working on extending new features and fixing any bug that might arise.

[211] https://neo4j.com/blog/custom-security-plugins-user-defined-procedures-neo4j-enterprise/

Bibliography

[1] *Graph Databases* by Ian Robinson, Jim Webber, and Emil Eifrem (O'Reilly). Copyright 2015 Neo Technology, Inc. 978-1-491-93089-2.

[2] *Neo4j in action* by Jonas Partner, Aleksa Vukotic, Nicki Watt (Manning). Copyright 2015 Manning Publications Co. 978-1-617-29076-3

[3] *Neo4j Essentials* by Sumit Gupta (Packt Publishing Limited) Copyright 2015 Packt Publishing. 978-1-78355-517-8

[4] *Learning Cypher* by Onofrio Panzarino (Packt Publishing Limited) Copyright 2014 Packt Publishing. 978-1-78328-775-8

[5] *Beginning Neo4j* by Chris Kemper (aPress) Copyright 2015 Chris Kemper. 978-1-4842-1228-8

[6] *Learning Neo4j* by Bruggen, Rik Van (Packt Publishing Limited) Copyright 2014 Packt Publishing. 978-1-84951-716-4

[7] *Neo4j Cookbook* by Ankur Goel (Packt Publishing Limited) Copyright 2015 Packt Publishing. 978-1-78328-725-3

[8] *Practical Neo4j* by Gregory Jordan (aPress) Copyright 2015 aPress. 978-1-4842-0023-0

[9] *Neo4j High Performance* by Sonal Raj (Packt Publishing Limited) Copyright 2014 Packt Publishing. 978-1-78355-515-4

[10] *Neo4j Graph Data Modeling* by Mehesh Lal (Packt Publishing Limited) Copyright 2014 Packt Publishing. 978-1-78439-344-1

[11] *Building Web Applications with Python and Neo4j* by Sumit Gupta (Packt Publishing Limited)
Copyright 2014 Packt Publishing. 978-1-78398-398-8

[12] *Neo4j 2.0 - Eine Graphdatenbank für alle* by Michael Hunger (entwickler.press)
Copyright 2014 entwickler.press. 978-3-8680-2128-8

[13] *Knowledge Graphs Webinar - 07/11/2017* by Jeff Morris (Head of Product Marketing - Neo Technology, Inc.)
Online Webinar

[14] *Solutio problematis ad geometriam situs pertinentis* by Leonhard Euler. Presented to the St. Petersburg Academy on August 26, 1735.

[15] *An Eulerian Trail through Königsberg* by Wilson, R. J., Journal of Graph Theory (1986)

[16] *Graph Theory (1st ed.)* by Balakrishnan, V. K. (1997-02-01). McGraw-Hill. 0-07-005489-4

[17] *A Relational Model of Data for Large Shared Data Banks* by Edgar Frank Codd. IBM Research Laboratory, San Jose, California

[18] *Database Systems - A Practical Approach to Design, Implementation, and Management* by Thomas M. Connolly, Carolyn E. Begg. - 5th ed. Pearson Education. 978-0-321-52306-8

[19] *Distributed Database System* by Chhanda Ray, Pearson India. 978-8131727188

[20] *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* by Pramod J. Sadalage and Martin Fowler. 978-0-321-82662-6

[21] *Basi di dati, modelli e linguaggi di interrogazione* by P. Atzeni, S. Ceri, S. Paraboschi, R. Torlone. 88-386-6292-4

[22] *Fondamenti di Basi di Dati* by A. Albano, G. Ghelli, R. Orsini. 88-08-07003-4