FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Graph databases for HR relationships

Rafael Araújo Moura

MASTER IN ELECTRICAL AND COMPUTERS ENGINEERING

Supervisor: Prof. Doutor Nuno Filipe Moreira Macedo

Co-supervisor: Engº José Miguel Fernandes Barbosa da Silva

September 17, 2021

Resumo

A modelação de dados de Recursos Humanos usa intensivamente estruturas em grafo para armazenar objetos (posições, contratos, tarefas, centros de custo, unidades organizacionais, etc.) e as suas relações (uma unidade organizacional reporta a outra unidade organizacional, uma posição pertence a uma unidade organizacional e também reporta a outra posição, etc.). Tradicionalmente, bases de dados com dados de Recursos Humanos são relacionais, e a consulta de dados orientados a grafos armazenados numa base de dados relacional é muito ineficiente.

O objetivo principal desta dissertação é a comparação, em termos de performance e flexibilidade, entre uma base de dados relacional e outra baseada em grafos, quando ambas contêm dados fortemente relacionados, tal como é o caso de dados de Recursos Humanos. Foi então necessário modelar uma base de dados com base num cenário real de uma empresa que lida diariamente com esse tipo de dados. Foram também formuladas interrogações que serviram de suporte para interagir com as entidades e respetivas relações armazenadas, e que permitiram obter resultados indicativos da performance de cada base de dados. Para além de uma análise desses resultados foram também comparadas as próprias interrogações, em termos de legibilidade e expressividade, com o objetivo de determinar qual dos casos é mais fácil de interpretar o que se pretende com a interrogação e quão mais simples é de a formular.

Ao executar interrogações que percorrem, quase que por completo, uma estrutura de dados, a base de dados baseada em grafos teve um desempenho muito melhor do que a base de dados relacional, obtendo até valores de tempos de execução 200 vezes menores do que os obtidos em SQL. Quanto às consultas hierárquicas, somente quando aumentamos a quantidade de relações de gerentes por funcionário, num fator de dez, pudemos constatar que Neo4j teve um desempenho até 3.5 vezes mais rápido que SQL. Estes resultados referem-se a bases de dados com, no máximo um milhão de funcionários, e seria expectável que estas diferenças de desempenho fossem superiores para bases de dados de maior dimensão e com dados ainda mais relacionados.

Para além disso, concluiu-se que a modelação da base de dados em grafos é mais intuitiva e imediata, e quanto às interrogações, a sua formulação é mais rápida, simples e legível no caso da linguagem Cypher quando em contraste com a sua escrita em SQL. Tal deve-se ao facto de todas as interrogações escritas em Cypher terem ocupado metade das linhas da mesma interrogação escrita em SQL, recorrendo também a caracteres ASCII para representação de nós e relações na correspondência de padrões. Consultas mais concisas resultam numa menor probabilidade de ocorrência de erros e ao mesmo tempo tornam mais fácil para novos desenvolvedores acompanharem, entenderem e trabalharem com interrogações previamente escritas.

Abstract

Human Resources data modeling makes heavy use of graph like structure to hold objects (positions, contracts, jobs, cost centers, org units, etc.) and their relationships (an org unit reports to another org unit, a position belongs to an org unit and reports to another position, etc.). Traditionally, Human Resources' databases are relational, and querying graph data stored on a relational database is highly inefficient.

The main objective of this dissertation is the comparison, in terms of performance and flexibility, between a relational and a graph database, when both have highly-connected data as it is the case with Human Resources' data. It was then required to model a database based on a real scenario of a company who deals these types of data on a daily basis. Queries were also formulated and they enabled the interaction with all stored entities and relationships between them while allowing the retrieval of performance results for both databases. Written queries were also compared, in terms of readability and expressability, with the objetive of determining which case is easier to understand what is being queried and how much simpler it is to formulate such query.

When executing queries that traverse almost completely a data structure, the graph database performed much better than a relational database, even getting execution time values 200 times smaller than those obtained in SQL. As for hierarchical queries, only when we increased the amount of manager relationships per employee, by a factor of ten, were we able to see that Neo4j performed up to 3.5 times faster than SQL. These results regard databases with a maximum of one million employees and it is safe to believe that the differences in performance would grow larger for even bigger and more connected databases.

Furthermore, it was concluded that the database modeling in graphs is more intuitive and immediate, and as for queries, its formulation is faster, simpler and more readable in the case of Cypher language when in contrast to its writing in SQL. This is due to the fact that the analysed Cypher queries had half the lines of their SQL query equivalents while also making use of ASCII characters for node and relationship representation when pattern matching. More concise queries result in a lower probability of errors occurring while also making it easier for new developers to catch up, understand and work with previously written queries.

Agradecimentos

Gostaria de agradecer ao Professor Doutor Nuno Filipe Moreira Macedo e ao Engº José Miguel Fernandes Barbosa da Silva por me terem orientado nesta última etapa, sempre com disponibilidade e muita compreensão.

Quero também agradecer ao meu amigo Tiago e aos meus primos por me acompanharem nesta luta diária que tanto me enriqueceu.

Por fim, gostaria de agradecer aos meus pais e à minha namorada por todo o apoio, incentivo e força que me dão para enfrentar toda e qualquer adversidade. São o que de mais especial tenho na minha vida e sei que sem vocês não teria tido o mesmo ânimo e vontade para conseguir alcançar os meus objetivos.

Rafael Araújo Moura

"All things are difficult before they are easy."

Dr. Thomas Fuller

Contents

List of Figures

List of Tables

List of Listings

Abreviaturas e Símbolos

HR	Human Resources
SQL	Structured Query Language
DBMS	Database Management System
RDBMS	Relational Database Management System
ACID	Atomicity, Consistency, Isolation, Durability
CRUD	Create, Read, Update and Delete
RDF	Resource Description Framework
W3C	World Wide Web Consortium
URI	Uniform Resource Identifier
MDM	Master Data Management
IDE	Integrated Development Environment
CLI	Command-Line Interface
HTML	HyperText Markup Language
E&E	Experiences & Exposures
CSV	Comma-Separated Values
ASCII	American Standard Code for Information Interchange

Chapter 1

Introduction

This chapter aims to contextualize the dissertation, discuss its objectives and what motivated its writting as well as presenting the document structure.

1.1 Context

Companies nowadays have to deal with huge volumes of data which are too large and complex to store, and more importantly, with complex relationships too hard to model and manage with the use of a relational database.

The Human Resources department of a company, for example, has to deal with information about its positions, contracts, jobs, cost centers, organizational units and many other highly connected data. Modeling and managing this type of information gets increasingly slow and complex with the increase of data, when using a relational database.

A graph database, on the other hand, is a type of NoSQL database that uses graph structures to store and manage data. Those structures contain nodes, edges, and properties representing entities and the relationships between them which allow data in the store to be linked together directly and, in many cases, retrieved with a single operation.

1.2 Motivation

Querying highly connected datasets inside relational databases can get highly inefficient as data multiplies and the overall structure of the dataset becomes more complex and less uniform since they become burdened with large join tables, sparsely populated rows, and lots of null-checking logic.

This does not happen, to the same extent, when using a graph database. Graphs represent entities as nodes and the ways in which those entities relate to the world as relationships. Querying relationships within a graph database is fast because they hold these relationships between data as a priority within the database itself.

This dissertation aims to determine how much difference in performance can a company benefit from when changing from a relational database to a graph database.

1.3 Objectives

Since in most company's cases data is stored in a relational database, we first convert that information so that it can be stored and managed in a graph database engine. That engine is decided after a comparison of available products, depending on factors such as expected performance, scalability, query language readability and expressability.

The main objectives of this dissertation are:

- Performance and flexibility comparison between relational and graph databases.
- Development of a website to help users with query creation.

In order to measure both database performances, we compare query execution times when using different types of queries for the same dataset, and also using the same query with increasingly datasets.

1.4 Document Structure

This document is divided into seven chapters with the following content:

- **Chapter 2** — Presents the current state of the art regarding the topics adressed by the dissertation;
- **Chapter 3** — Decided work plan with its different tasks and respective objectives. Also describes the technologies and methodologies used during each phase.
- **Chapter 4** — Explains how to define a relational database model and respective convertion to a graph database model.
- **Chapter 5** — Defines what queries are executed against both databases and how to write them and compare them.
- **Chapter 6** — Discusses test data generation and how to import it, as well as how to execute written queries, extract information from them and analyze obtained results.
- **Chapter 7** — Presents conclusions and discusses possible future work regarding this dissertation's topics.

Chapter 2

Literature Review

This chapter includes the state-of-the-art regarding relational and graph databases. The information presented in this chapter will be the base of the dissertation work and will help to understand and characterize our problem, as well as finding a feasible solution to it.

2.1 Relational Databases

A relational database is a type of database that follows a model first introduced by E. F. Codd in 1969. [1, 2] Its creation, access and maintenance is made available by the means of a database management system based on a relational model.

2.1.1 Relational Database Management System

The relational database management system (RDBMS) is a piece of software capable of querying and manipulating the database. According to E. F. Codd, "All of these capabilities should be designed into the DBMS from the start, not added afterwards." There are plenty of options in the market such as Microsoft SQL Server, MySQL, Oracle Database, PostgreSQL, MariaDB, SQLite, IBM DB2 and so on. In order for a RDBMS to be called "fully" relational, which means it manages databases entirely through its relational capabilites (Codd's foundation rule), it needs to satisfy a set of twelve rules [3] defined by E. F. Codd:

- **Rule 1**: all information in a relational database is represented explicitly at the logical level and in exactly one way - by values in tables.
- **Rule 2**: each and every datum (atomic value) in a relational database is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name.
- **Rule 3**: null values (distinct from the empty character string or a string of blank characters, and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way (independent of data type).

- **Rule 4**: the database description is represented at the logical level just like ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.
- **Rule 5**: a relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language (1) whose statements are expressible per some well-defined syntax as character strings; and (2) which is comprehensive in supporting ALL of the following items:

  1. data definition
  2. view definition
  3. data manipulation (interactive & by program)
  4. integrity constraints
  5. authorization
  6. transaction boundaries (begin, commit, and rollback)

- **Rule 6**: all views that are theoretically updatable are also updatable by the system.
- **Rule 7**: the capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data.
- **Rule 8**: application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods.
- **Rule 9**: application programs and terminal activities remain logically unimpaired when informationpreserving changes of any kind that theoretically permit unimpairment are made to the base tables.
- **Rule 10**: integrity constraints specific to a particular relational database must be definable in the relational data sublanguage and storable in the catalog (not in the application programs).
- **Rule 11**: a relational DBMS has distribution independence.
- **Rule 12**: if a relational system has a low level (singlerecord- at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple records-at-a-time).

2.1.2 Relational Data Modeling

E. F. Codd published a book on relational models [4], which states that data can be classified into two types: atomic and compound. Atomic data is indivisible and compound data consists of structured combinations of atomic data that can be decomposed by the DBMS. In the relational model proposed by E. F. Codd, there is only one type of compound data: the relation. Those relations are

represented by tables with its rows being named tuples and its columns named attributes. Codd also states that there is no need for specific ordering of tuples and attributes since there is no idea of "nextness" as is the case with arrays.

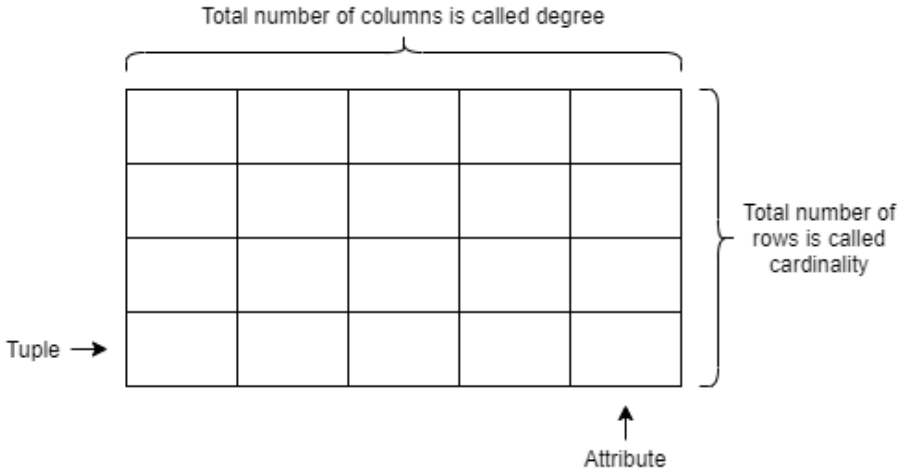

Figure 2.1: Relational model terminology.

An object represented in a relational database has different attributes. Each tuple or row contains the information about a single object's characteristics and a table with multiples tuples is able to describe the relations between those objects. Therefore, an object needs a unique key that distinguishes it from all the others and preserves its unicity. Attributes from different objects in the same domain (set of possible values for a given attribute) need to satisfy the same constraints.

Relations stored in the database are called "base relations" and when a query is applied to these "base relation" it results in what Codd calls "derived relations". The "derived relations" do not store data in the database as they are merely for visualization.

2.1.3 Constraints

There are four types of constraints:

- Domain constraints
- Key constraints
- Entity integrity constraints
- Referencial integrity constraints

Domain constraints exist to ensure that the attributes in relations are stored with respect to the intended datatype, for example, if a certain attribute cell is expecting an integer value the data to be stored cannot be a string. It is possible to limit the size of the data to be stored, but the attribute is restricted to a single value.

Key constraints were already described and they ensure that every tuple in a relation contains a unique object. This is accomplished by assigning the objects one or multiple keys, one of which is a primary key holding a unique value that identifies a single object. Examples would be our identity or phone numbers, that must be unique, otherwise it would be impossible to determine what person they refer to. Another type of key used to link two tables together is a foreign key which references another primary key in a different table.

E. F. Codd defines the two last **integrity constraints** in Section 13.2 of his book as: [4]

1. Type E, ***entity integrity***. No component of a primary key is allowed to have a missing value of any type. No component of a foreign key is allowed to have an I-marked value (missing-and-inapplicable).

2. Type R, ***referencial integrity***. For each distinct, unmarked foreign-key value in a relational database, there must exist in the database an equal value of a primary key from the same domain. If the foreign key is composite, those components that are themselves foreign keys and unmarked must exist in the database as components of at least one primary-key value drawn from the same domain.

2.1.4 Relational Operations

E. F. Codd divides the existing operators into two categories: basic and manipulative. The basic operators include **projection, theta-selection, theta-join, relational product, relational division, relational union, relational difference** and **relational intersection**. Codd also stated that all operators except the relational or cartesian product should be implemented in a RDBMS because of memory and time waste associated with that operation.

2.1.4.1 Projection

The project operator receives a table as its operand and generates another with only the attributes listed in the command, while omitting others. From this table, all duplicate tuples are removed and the final result contains only one ocurrence of each row.

2.1.4.2 Theta-selection

The theta-select or select operator also receives a table as its operand, generating another table with all the tuples containing the specficied value for a certain attribute. This operator also removes duplicate rows from the final result.

2.1.4.3 Theta-join

The theta-join or join operator receives two different tables as its operands. The result consists of a table with rows concatenated from both tables, but only if those rows return as true to a specified condition.

2.1.4.4 Relational division

The logic behind a relational division operator is the same as in integer arithmetic, given that it takes two tables as operands (dividend and divisor), resulting in two tables: quotient and remainder.

2.1.4.5 Relational union

The relational union operator copies tuples from both of its table operands and results in a single table with rows from both operands, but removing duplicated rows.

2.1.4.6 Relational difference

The difference operator receives two tables as operands. The result consists in a table with no duplicate rows from the first table that do not appear in the second.

2.1.4.7 Relational intersection

The relational intersection operator returns a table with no duplicate rows, containing only tuples that are present in both tables passed as operands.

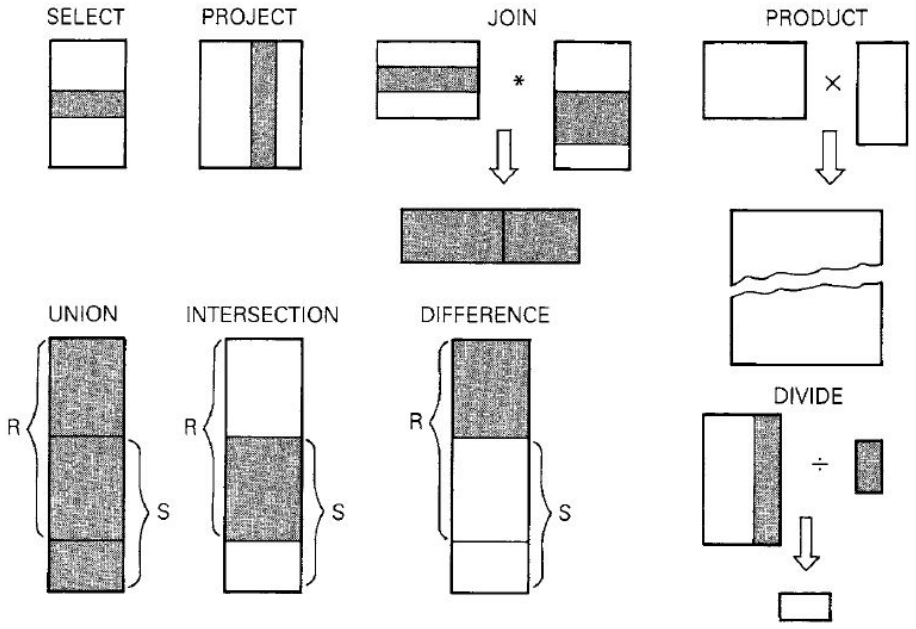

Figure 2.2: The basic operators of the relational model introduced by E. F. Codd

The eight manipulative operators described by Codd are the **relational assignment, insert, update, primary key update with cascaded update, primary key update with cascaded marking, delete, delete with cascaded deletion** and **delete with cascaded marking**. These operators, as the name suggests, manipulate the relations stored in a database.

2.1.4.8 Relational assignment

A relational assignment operator retains the result of the query (a table) in memory, under a specified name by the user. Codd states that, even though a relational assignment operator is beyond the capabilities of most programming languages, a fully relational DBMS should be able to perform such operation.

2.1.4.9 Insert

The insert operator allows the user to add rows to a table stored in the database.

2.1.4.10 Update

The update operator is used when there's a need to change values of certain attributes of a row.

2.1.4.11 Primary key update with cascaded update

Changing the value of a primary key needs extra care because of integrity database issues, but typically, primary keys never need their values changed.

2.1.4.12 Primary key update with cascaded marking

This operator behaves like the previous one but takes into consideration all the foreign keys that could be affect with primary key updating. If at least one of these foreign keys end up referencing an illegal primary key, the operation fails and the database rolls back to its previous state.

2.1.4.13 Delete

The delete operator works the opposite way compared to the insert operator as it is able to remove rows from a stored table.

2.1.4.14 Delete with cascaded deletion

The previous operator will often violate referential integrity as it does not take into account the fact that a row to be deleted might have a primary key associated to a base relation. This delete operator is then similar to the previous one but takes this into account. Use of the delete with cascaded deletion operator causes the RDBMS to propagate deletions to rows in the database that happen to contain dependent foreign-key values as components. Needless to say that this operator needs to be used with extreme care as it may end up deleting many rows that the user is unaware of.

2.1.4.15 Delete with cascaded marking

This last operator is less dangerous than the cascaded deletion because the initial cycle of cascading does not trigger any subsequent cycles of deletion.

2.1.5 Transactions

All of the operations previously described are not fail-proof since there could be errors present in the RDBMS software or these incidents could be triggered by external events such as a power failure. The though of preventing these events from compromising data integrity led to the concept of transactions, based on the four principles of atomicity, consistency, isolation and durability, also known as ACID transactions. [5, 6]

2.1.5.1 Atomicity

Since a transaction is composed of many operations, the atomicity property states that the transaction is successful if and only if every single operation that constitutes the transaction succeeds as well. If for whatever reason an operation fails to execute, the transaction fails and the database rolls back to the previous state.

2.1.5.2 Consistency

The consistency property prevents the database from allowing a transaction to change the contents of a database in such a way that it would violate its constraints. If an illegal transaction occured, the database would become corrupted and therefore violating the last property, durability.

2.1.5.3 Isolation

The isolation property consists of synchronization methods to ensure that every transaction can run concurrently without affecting each other. This property is of great importance since databases get a lot of requests at the same time.

2.1.5.4 Durability

The durability property guarantees that the contents of a database comitted by any transaction will not be lost or corrupted due to any kind of failure.

2.2 Graph Databases

A graph database is a type of NoSQL database that uses graph structures to store and manage data, unlike relational databases which represent data in tables. The management system of a graph database is generally an online transactional processing system (OLTP) that supports create, read, update and delete operations commonly known as CRUD methods [7].

The graph database market offers many options and it becomes difficult to choose one to work with. A website [8] calculates a score for every database based on mentions, searches in Google Trends, frequency of online technical discussions, relevance in social networks and job offers where the system is mentioned. This process is performed monthly and Figure 2.3 shows the top three results for the month of February, 2021.

Rank			DBMS	Database Model	Score		
Feb 2021	Jan 2021	Feb 2020			Feb 2021	Jan 2021	Feb 2020
1.	1.	1.	Neo4j	Graph	52.16	-1.62	+0.96
2.	2.	2.	Microsoft Azure Cosmos DB	Multi-model	31.66	-1.31	-0.29
3.	3.	3.	OrientDB	Multi-model	5.13	-0.20	+0.19

Figure 2.3: DB-Engines Ranking of Graph DBMS.

2.2.1 Graph Structures

A graph is a mathematical structure consisting of multiple nodes or vertices connected by edges representing the relationships between the nodes. There's a distinction between graphs with undirected edges which define equal relationships, no matter the starting node, from graphs with directed edges where a relationship exists from one node to another and never vice-versa.

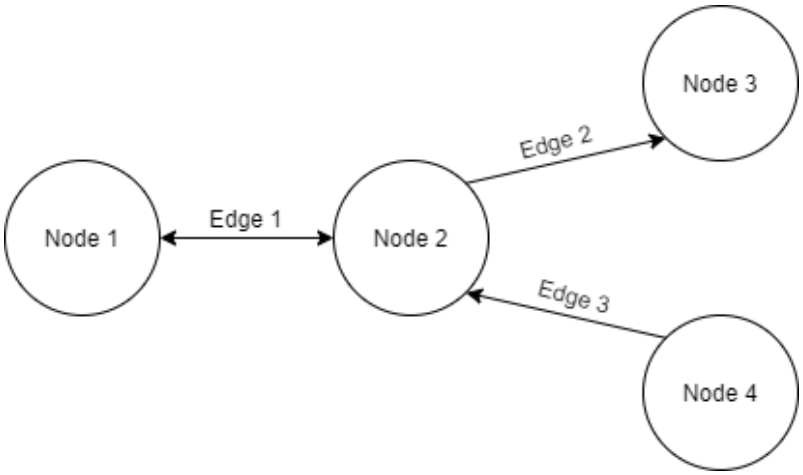

Figure 2.4: A basic graph structure.

2.2.2 Graph Data Modeling

There are three dominant graph data models: property graphs, triples or RDF and hypergraphs.

2.2.2.1 Labeled-Property Graph Model

A property graph model contains nodes and relationships. These nodes can be labeled to be associated with certain groups and also have properties defined by key-value pairs while the relationships have a name, properties and a direction, meaning they always start from one node to another. As an example [7] to represent Alice and Bob, each owning three vehicles, the graph structure would consist of five nodes (two individuals and three vehicles) and the relationships between the nodes as shown in Figure 2.5.

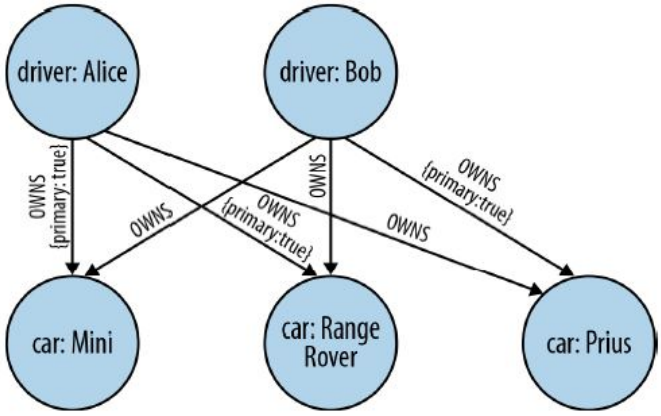

Figure 2.5: A property graph example.

2.2.2.2 Resource Description Framework Model

The World Wide Web Consortium (W3C) defines RDF as a framework that consists of a set of triples and is used for representing information in the Web [9]. A triple is a data structure divided into three parts: Subject, Object and Predicate or property

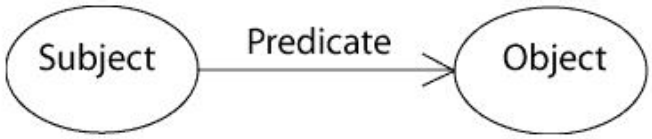

Figure 2.6: Resource Description Framework data model.

A triple has little information individually but it becomes obvious that with such a structure it is possible to define a big network of nodes (subjects and objects) and also their relationships (predicates). In this model, nodes may be left blank, identified by literals or by a Uniform Resource Identifier (URI), this latter one also used to identify the relationships. There's two types of literals: typed (strings combined with a URI) or plain (strings with an optional language tag). When a node has no name and is not identified by a literal or a URI, it needs to have a local blank node identifier to distinguish it from all the other identifiers.

2.2.2.3 Hypergraphs

Hypergraphs are of great usefulness when there's a lot of many-to-many relationships since this model allows the edges (called hyperedges) to connect to several nodes. While other models would require a higher number of edges to represent a many-to-many relationship, an hypergraph is capable of doing so with a single hyperedge. Using the same example shown for property graphs, we can conclude that the same information is representable using a single hyperedge.

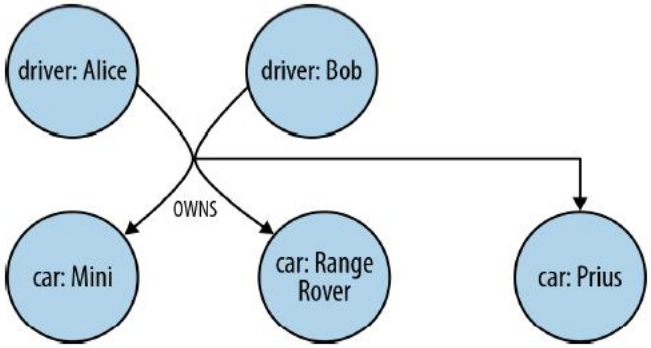

Figure 2.7: An hypergraph example.

2.2.3 Graph Database Properties

2.2.3.1 Graph Storage

Some graph databases use native graph storage that is optimized and designed for storing and managing graphs. Not all graph database technologies use native graph storage, however. Some serialize the graph data into a relational database, an object-oriented database, or some other general-purpose data store.

2.2.3.2 Index-Free Adjacency

In order to increase performance of graph database operations, the use of index-free adjacency was determinant. Index-free adjacency means that when two nodes share a relationship, and since each node has its physical RAM address, they physically point to each other, reducing the time needed for the data lookup. [10]

2.2.4 Graph Database Use Cases

There are many use cases for a graph database such as recommendation systems, social and network graphs, telecommunications, supply chain management and so on. However, since this dissertation focuses on the relationships present in HR, one important use case is master data management (MDM) which consists of critical business data. Master data includes information about a company's users, employees, products, contracts, departments, customers, job sites and all the information that a typical HR department deals with. Day by day, data generated by companies grow, both in size, complexity and in most cases, is constantly undergoing change meaning a database capable of representing all these complex relationships while providing a flexible schema is of extreme importance.

2.2.5 Comparison With Relational Databases

Despite relational databases being called relational, they are optimized for tabular structured data. A consequence of such limitation is that the resulting schema is strict and does not model ad hoc information with the same ease as a graph database. This does not mean relationships do not exist in a relational database, as it is shown in Section 2.1.2, but retrieving those relationships requires complex queries that access multiple tables and perform several joins which have an heavy impact on performance.

Graph databases, on the other hand, do not rely on join operations and have a schema flexible enough to deal with data arrival without changing the already existing data set. Graph database's way of storing data and processing the information using index-free adjacency increases its performance when compared to relational databases. A study was conducted by Roman Čerešňák [11] and he compared the performance between relational and non-relational databases when performing CRUD operations on a database with 100 000 records.

Type/Operation	Oracle	MySql	MsSql	Mongo	Redis	GraphQL	Cassandra
Insert	0.091	0.038	0.093	0.005	0.010	0.008	0.011
Update	0.092	0.068	0.075	0.009	0.013	0.012	0.014
Delete	0.119	0.047	0.171	0.015	0.021	0.018	0.019
Select	0.062	0.067	0.060	0.009	0.015	0.011	0.014

Figure 2.8: CRUD operation performance (in milliseconds) between relational and non-relational databases.

By inspecting his results, we can conclude that for all CRUD operations, the graph database GraphQL outperformed all RDBMS query times.

2.2.6 Graph Query Languages

The most common graph query languages are the following:

- **Cypher Query Language** was developed by Neo4j, Inc. [12] and is a declarative graph query language based on the property graph model with similarities to the SQL language.
- **Gremlin** was developed as a part of an Apache TinkerPop project [13].
- **SPARQL** is a query language developed to operate RDF databases.

Chapter 3

Work Plan

This chapter introduces the technologies and tools used while implementing both databases and while extracting the required information upon query execution. It also presents the work plan to achieve this dissertation main goal, by means of a Gantt chart which schedules the tasks to be performed.

3.1 Technologies and Tools

The database management system used to implement the relational database will be Microsoft SQL Server since this dissertation was proposed by a company that currently uses it as their RDBMS. As for the graph database, the decision derived from the state-of-the-art study (refer to Section 2.2) based on a website which assigns scores to several graph database management systems, in which Neo4j ranked highest. Data generation to populate both databases was achieved with Microsoft Visual Studio Code and the Python programming language (refer to Section 6.1).

3.1.1 Microsoft SQL Server

Microsoft SQL Server is a well-documented relational database management system developed by Microsoft, first released in April 1989 as SQL Server 1.0 and currently on version SQL Server 2019, released in November 2019. Microsoft SQL Server 2019 Developer Edition will be the selected version to work on since it is the most recent stable release. It will allow a personal computer to create the relational databases and to store and retrieve data from them with the assistance of an integrated environment (Microsoft SQL Server Management Studio 18.9.2) which provides tools for managing the SQL infrastructures as well as an interface for query editing and execution.

3.1.2 Neo4j

Neo4j is an open-source, native graph database, written in Java and Scala which implements labeled-property graph models. The first version of Neo4j was developed in 2002 and it has

been improved continuously ever since. For this dissertation, Neo4j Desktop 1.4.8 is the selected developer IDE because it allows the creation of multiple database servers locally so that we can have different projects and database instances for different data set sizes. Bundled with Neo4j Desktop are several tools that help developers with database management, query writing and data visualization, like Neo4j Browser and its command-line tool named Cypher Shell.

3.1.2.1 Neo4j Browser

Neo4j Browser is an interface for querying a graph database using the Cypher query language. It offers a Cypher editor with syntax highlighting, code completion, and warnings to help writing queries. The developer has the flexibility to control the format of its query results and has the ability to visualise them via a table or an interactive experience which displays the nodes and relationships in a way suitable for the developer to interpret the results. On the left side bar of Figure 3.1 we can see plenty of qualitative and quantitative database information on node labels, relationship types and property keys. It lets developers save their favorite queries so that they can easily return to them afterwards, and comes with examples of basic queries and common procedures such as visualising the database schema.

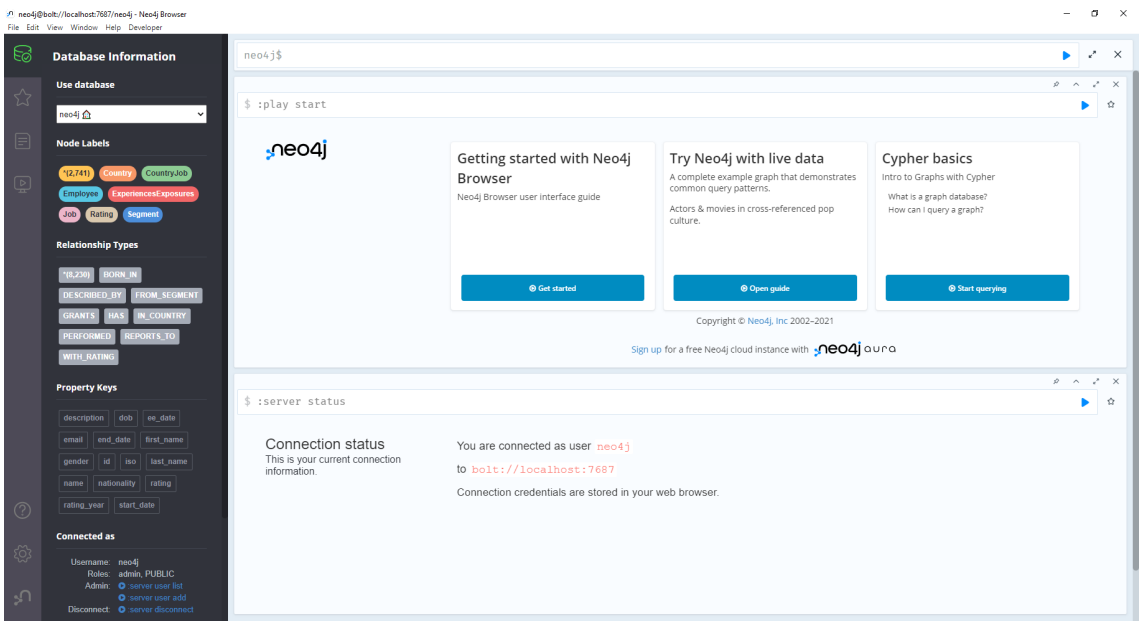

Figure 3.1: Neo4j Browser interface.

3.1.2.2 Neo4j Cypher Shell

Upon installing Neo4j, Cypher Shell will be located in the bin directory of a database instance. It is a command-line tool used to run queries, scripts and also allows for database management. When the terminal is running and cypher-shell is executed, a connection to the database is established by passing in the username and password arguments, as shown in Figure 3.2.

```
D:\.Neo4jDesktop\relate-data\dbmss\dbms-326f9be2-a328-436b-8472-7eb96fd97b8e>cd bin

D:\.Neo4jDesktop\relate-data\dbmss\dbms-326f9be2-a328-436b-8472-7eb96fd97b8e\bin>cypher-shell

username: neo4j
password: ********
Connected to Neo4j using Bolt protocol version 4.3 at neo4j://localhost:7687 as user neo4j.

Type :help for a list of available commands or :exit to exit the shell.
Note that Cypher queries must end with a semicolon.
neo4j@neo4j> :help

Available commands:
  :begin    Open a transaction
  :commit   Commit the currently open transaction
  :exit     Exit the logger
  :help     Show this help message
  :history  Print a list of the last commands executed
  :param    Set the value of a query parameter
  :params   Print all currently set query parameters and their values
  :rollback Rollback the currently open transaction
  :source   Interactively executes cypher statements from a file
  :use      Set the active database
```

Figure 3.2: Connecting to a database instance by using Neo4j Cypher Shell.

3.2 Work Plan

This section presents the Gantt chart which illustrates the precedence and estimated duration of each task, decided before the start of this dissertation, while the next section establishes a comparison between this estimate and what really ended up happening.

	Name	Duration	Start	Finish	Percent Complete
1	Dissertation Planning	60 days	11/20/20 8:00 AM	2/11/21 5:00 PM	100%
2	Definition of Database Models	15 days	2/12/21 8:00 AM	3/4/21 5:00 PM	0%
3	Implementation of Database Models	20 days	3/5/21 8:00 AM	4/1/21 5:00 PM	0%
4	Comparison of Database Query Languages	10 days	4/2/21 8:00 AM	4/15/21 5:00 PM	0%
5	Migration of Relational Data to Graph Data	10 days	4/16/21 8:00 AM	4/29/21 5:00 PM	0%
6	Extraction of Query KPIs	10 days	4/30/21 8:00 AM	5/13/21 5:00 PM	0%
7	Results Analysis	15 days	5/14/21 8:00 AM	6/3/21 5:00 PM	0%
8	Development of a Website	10 days	6/4/21 8:00 AM	6/17/21 5:00 PM	0%
9	Dissertation Writting	95 days	2/12/21 8:00 AM	6/24/21 5:00 PM	5%

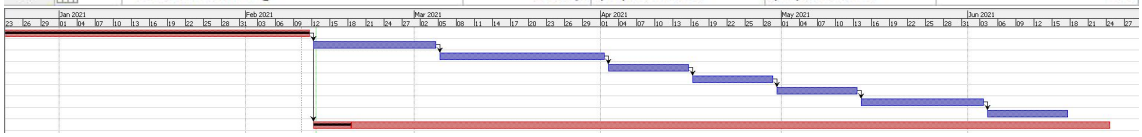

Figure 3.3: Gantt chart for the task list, developed with ProjectLibre software.

The first task was the dissertation planning which required the study of the state-of-the-art and deciding on what technologies and tools are to be used. Before starting with both databases implementation, their structure models need to be well-defined as well as what data is to be received. Upon database implementation, different types of queries are written so that database performance can be assessed. Not only that, the queries themselves will be compared in terms of readability and expressibility. Since most companies that explore graphs as an alternative to their relational

databases already use relational data, the need for data migration arises. In this step, relational data is exported to a CSV file that in return is adapted so that it can be imported by the graph database. In order to evaluate the performance of a given query, there's the need to decide how is the query going to be evaluated and how is such information going to be extracted. In this task, and for all the considered queries, key performance indicators will be decided as well as a method to retrieve data accordingly. After extracting the queries' key performance indicators, the next task is to analyse the obtained results, and if they are not satisfactory, changes need to be carried out before writting conclusions in the dissertation. The final task is the development of a a simple HTML website where the user can change filters, in dropdown menus, to query the database.

However, this work plan is merely an ideal sequence of tasks with their estimated duration when in reality they could be delayed for multiple reasons.

3.3 Risk Analysis

The objective of this section is to present the risks associated with the development of the dissertation. Each risk is qualitatively classified in terms of likelihood of occurrence as well as the impact it would have on the project in case it happened. Furthermore, all obstacles faced during this dissertation will be discussed and their impact on the rest of the project will be described.

Table 3.1: Possible risks associated with the dissertation development.

Type of Risk	Risk Description	Probability	Impact
Technological	Incorrect definition of the relational model	Low	High
Technological	Incorrect definition of the graph model	Medium	Low
Technological	Not enough resources for database operations	Low	Medium
Technological	Queries written in an inefficient manner	Medium	Medium
Technological	Queries behave differently between databases	Medium	High
Human	Task took longer than expected	High	Medium
Human	Task was more complex than expected	Medium	Medium

The first two technological risks involve an incorrect definition of the database models. As stated in 3.1, this dissertation was proposed by a company and thus, the definition of the relational model was carried out by the author and the co-supervisor (company representative). The likelihood of a relational model being incorrectly defined is consequently very low, however, if it did happen, the longer the error goes unnoticed the longer it would take to adjust everything that was done up until that moment. Even though the second risk has the same premises as the former, incorrectly defining the graph model is more likely to occur due to inexperience with Neo4j and graphs in general, but since a graph schema is significantly more flexible and adjustable to changes, it would have a low impact on the dissertation development.

As for the third risk, when performing database operations such as importing millions of rows of data or executing a rather complex query, the amount of CPU and memory resources need to be sufficient for successfully executing said operations. Usually it doesn't happen and when it does, the most likely cause is a poorly written or optimized query. That brings us to the last two

technological risks that involve writing a query in an inefficient way or even have it return different results when executed against both databases. Lack of experience is the most likely cause to these mistakes and, to mitigate them, queries need to be rewritten as many times as needed until they are performant if expected to be performant and to avoid, in this case, having SQL and Cypher queries returning different results we have to constantly validate and compare results from both types of queries.

The last two risks have a high likelihood of occurrence and can impact the development depending on how long the resulting delay gets. The worst case scenario would be the need to abandon an objective in order to allocate more time to the main focus of this dissertation. To avoid this situation, weekly meetings will take place where tasks in progress and tasks in the future are well planned and discussed.

3.3.1 Encountered Problems

During this dissertation, some tasks were carried out with little to no delays or unforeseen events while others took a considerable amount of time to complete, therefore impacting the rest of the project.

In retrospective, the dissertation's planning went as expected and it included the study of the state of the art, creating a work plan and deciding on what technologies and tools were to be used. The definition of the database models was also a task where no major difficulties were felt, however, since there was no prior contact with the database management system softwares, for any of the databases, getting used to them took its time, more so for Neo4j where some installation problems arised. Instead of using sensitive data provided by the company, we decided to generate our own data set samples for both databases. The task of generating said data was fairly simple and did not take longer than expected, however, importing them to Neo4j was not as simple as it was with SQL Server. The difference being that, at first, when trying to load the data set as a whole, Neo4j could not handle importing large data set sizes in one go because of lack of memory (Java heap size and page cache memory). In order to fix this problem, the first solution was to increase the allocated memory, but this only postponed the same issues for when even bigger data sets where being imported. To avoid having to allocate lots of memory for Neo4j to work properly, we make use of Cypher's `PERIODIC COMMIT` to run the same statement in batches, each importing a smaller and more manageable data set size into Neo4j. Therefore, implementing the databases, generating and importing data were tasks that took longer than expected but did not have a big impact on the remaining tasks.

The next task consisted of writting the first queries to test the behaviour of both databases and, at first, it was easy to write them for the relational database because of previous knowledge of the SQL query language. As for Neo4j, writting the queries to retrieve the correct data for each query was not harder than expected, it was quite the opposite. However, the obtained values for query execution times were way higher than expected. Finding ways to optimize the queries for both databases was a long process and one that had a huge impact on the other tasks. In fact, the impact was such that the additional task of developing a website to query the databases was

abandoned, consequently failing one of the objectives set for this dissertation. The last task of analysing the results did not take longer than expected and no major obstacles arised. In short, the lack of previous experience with some of the technologies and, in some moments, a poor time management, led to the delay of the dissertation's writting and impacted the outlined objectives for it.

Chapter 4

Database Models

This chapter introduces the relational database model to be used as basis to this dissertation. The next step, upon defining said model, is converting it to a graph database model by performing a given set of steps, followed by a short comparison between the obtained models.

4.1 Relational database model

This dissertation was proposed by a company which specializes in designing and implementing processes in the area of Human Resources, so the following model was thought out in a way that it comes close to a real scenario.

The model central entities are the **Employees** who comprise every person working at a company. Those employees report to their **Managers** and are also assigned yearly **Ratings** based on their performance. The employees take on different **Jobs**, each of them belonging to a certain **Segment** of work. Those jobs are performed in a given **Country** and may even grant the employees a set of skills, named **Experiences&Exposures**. Employees have the ability to claim them, even though they were not obtained by working on a job.

Therefore, the model contains seven entities stored in main domain tables while the relationships between them are represented by associative entity tables:

4.1.1 Main domain tables

- **Employees** have a unique identification number, first and last name, email, gender, date of birth and a foreign key to the Country table storing employee's nationality information.

- **Ratings** intend to qualify the employees' performance by assigning them a value between A, B, C and D, A being the highest ranking and D being the lowest. Each ranking is paired with a unique identification number.

- **Jobs** have a unique identification number, name, description and a foreign key to the Segments table.

- **Segments** and **Experiences&Exposures** have unique identification numbers, names and descriptions.
- **Countries** have a unique identification number, name, nationality and corresponding ISO 3166-1 alpha-3 code which are three-letter country codes that represent countries.

4.1.2 Associative entity tables

- **Managers** stores two foreign keys referencing the same employee table. One keys represents the manager while the other represents the subordinate.
- **Employees_Experiences&Exposures** represents all the E&Es an employee claims he has. The table contains two foreign keys, one for the Employee table and the other for the Experiences&Exposures table as well as the date it was claimed.
- **Employees_Ratings** stores information about yearly employees' ratings, therefore, a year and two foreign keys are required.
- **Jobs_Experiences&Exposures** contains data regarding what E&Es are granted to employee whenever they perform a job. For this table, only two foreign keys are needed to pair jobs with E&Es.
- **Employees_Jobs** gives information about what jobs an employee worked on, as well as in what country they were performed and between what dates. Three foreigns keys are needed for the Employee, Country and Job tables in addition to the start and end dates.

4.1.3 Database diagram

Knowing how data is organized, it is now possible to create the database structure by manually adding tables while selecting the correct data types for every attribute by using the Object Explorer tool in Microsoft SQL Server Management Studio, as shown in 4.1.

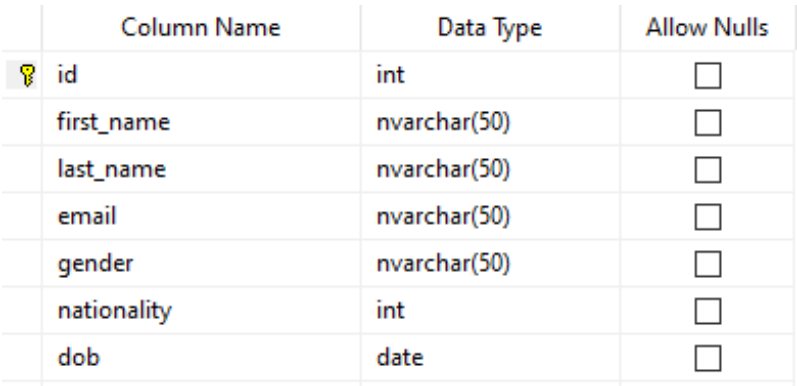

Figure 4.1: Example of employee table creation by using the Object Explorer tool in Microsoft SQL Server Management Studio 18.

After the whole database has been set up, the same Object Explorer tool was used to build a diagram, like the one in 4.2, displaying the entire database structure.

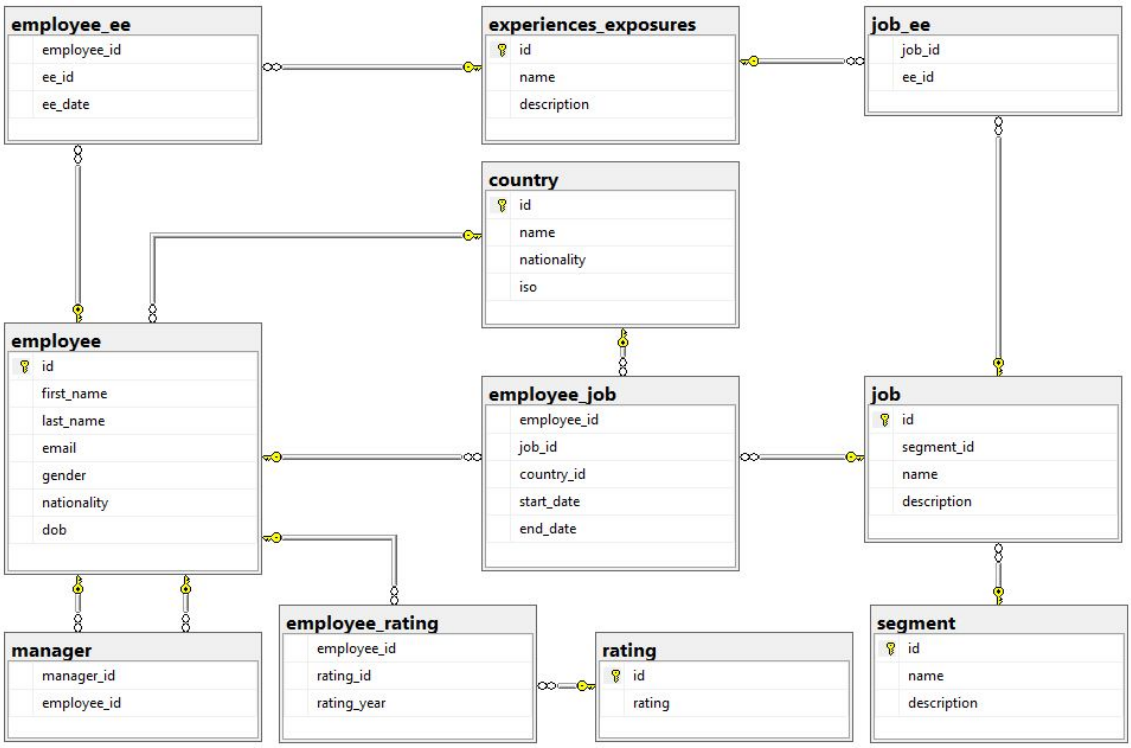

Figure 4.2: Relational database model drawn in Microsoft SQL Server Management Studio 18.

4.2 Graph database model

4.2.1 Relational to graph data model

Whether there is already an existing relational model that we wish to convert to a graph model, or the lack of experience using graphs forces a user/company to start from the relational model design first. Neo4j has made available developer guides that cover a variety of topics to help understand graph database concepts and also on how to build applications that interact with the software. One of the guides covers the subject of graph data modeling [12] and served as a starting point for converting the relational model to a graph model, since it lists several steps that help translate relational data model components to those of a graph data model.

- **Main domain table to node label** - each main domain table in the relational model becomes a label on nodes in the graph model.
- **Row to node** - each row in a main domain table becomes a node in the graph.
- **Column to node property** - columns (attributes) on the relational tables become node properties in the graph.

- **Business primary keys only** - remove technical primary keys and keep business primary keys.

- **Add constraints/indexes** - add unique constraints for business primary keys and, if needed, for frequent lookup attributes.

- **Foreign keys to relationships** - remove the foreign key and replace it with a relationship pointing to the referenced table.

- **No defaults** - There is no need to store nulls and empty values, so they should be removed entirely.

- **Index columns to array** - indexed column names (like email1, email2, email3) might indicate an array property.

- **Associative entity tables to relationships** - associative entity tables or join tables are transformed into relationships, columns on those tables become relationship properties.

By applying the above steps to the model described in 4.2 we can come up with the graph database model. Tables are already categorized by main domain tables and associative tables so the first step should be creating node labels from every single main domain table. That means six node labels are created: Employee, Rating, Job, Segment, Experiences&Exposures and Country. Every row in those tables will become a different node in the graph model and the columns associated to the same rows will be converted to node properties (e.g., every employee node will have its first and last name, email, gender and date of birth as properties). The nationality column in the employee table will not become a node property since all foreign keys are replaced by a relationship which, in this case, points to the country node.There will be no array properties in this example because there are no columns with values representing the same thing. Indexes or constraints are not visible in the model but will need to be added for the business primary keys in order to ensure data unicity. For this case, every business primary key is named 'id' in all main domain tables. Lastly, all associative entity tables become relationships between the referenced tables and any columns are now relationship properties. For example, the employee_rating table is replaced by a WITH_RATING relationship with the year as its property.

After following all the steps, the graph data model should look like the one presented in 4.3.

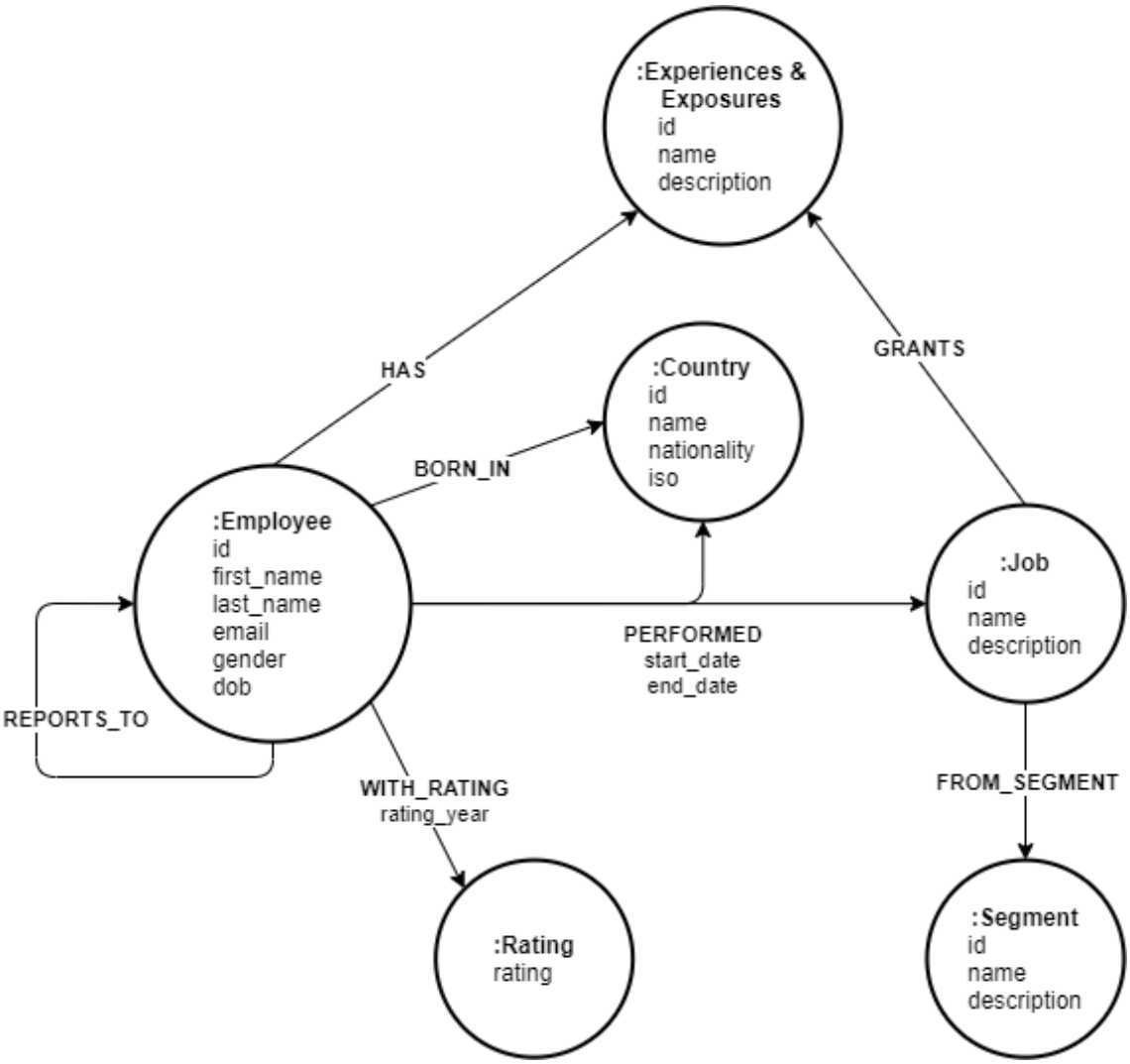

Figure 4.3: Obtained graph database model after conversion from the relational model.

There is a particular problem regarding the PERFORMED relationship, which can be classified as an hyperedge (refer to Section 2.2.2.3), since it connects more than one node and, in Neo4j, it is impossible to have those types of relationships. By first looking at the model, we might be tempted to split the hyperedge into two relationships: one to relate employee and job, and another for job and country, however, by doing so it becomes impossible to determine in what country a job was performed.

Mark Needham, a former Developer Relations Engineer at Neo4j, published an article in his blog [14] where he discusses how he managed to model a hyperedge in a labeled-property graph model without even realising. The problem came up when he had a relationship between a player and a football match and he wanted to say that a player participated in a match and represented a specific team in that match. The solution is as simple as creating an extra node, named Country_Job, which links employees, jobs and countries. The model presented in 4.4 takes those changes into account.

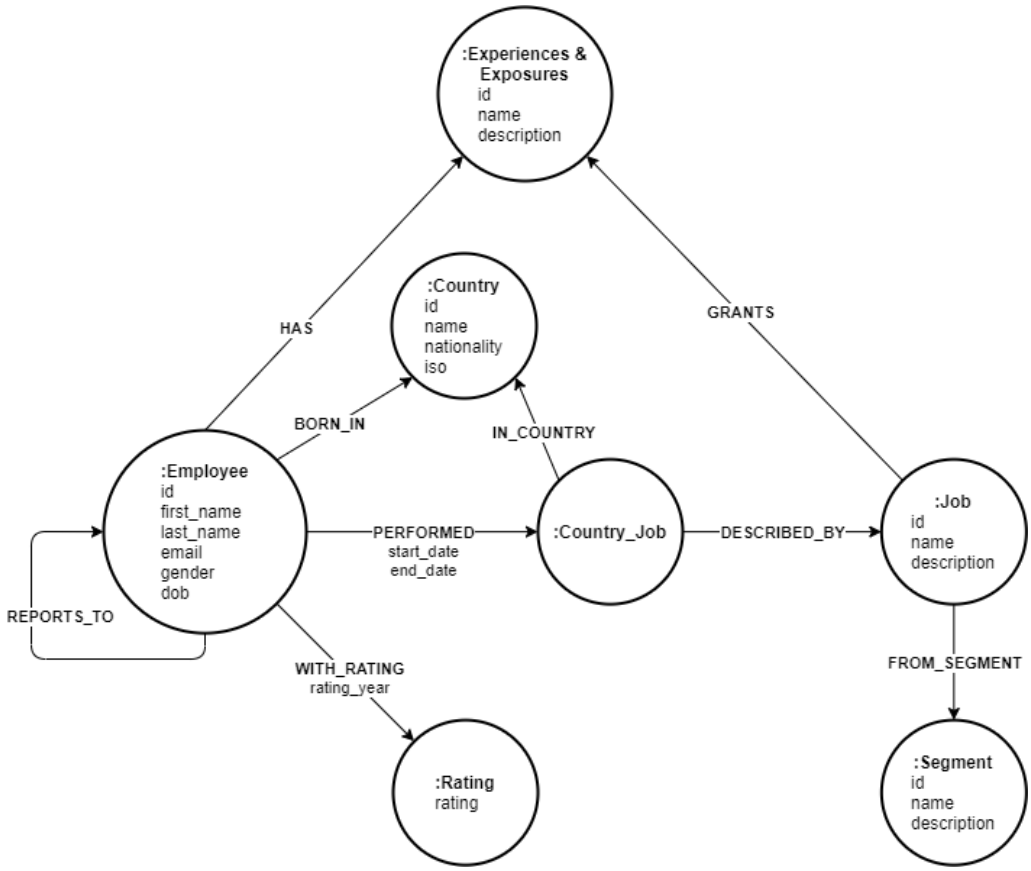

Figure 4.4: Graph database model after adding the Country_Job node.

4.2.2 Data model differences

To compare both databases in terms of querying performance, let's imagine we want to determine which jobs an employee performed. In the relational model, if we search for the identification number of a single employee out of millions of rows and then search for all the rows in employee_job that reference that same employee, we would still have to use the relevant rows to retrieve each job's information. It gets even worse if we continue the process up to the experiences_exposures table. In order to do the same search in the graph model, we would have to search for the same employee out of all employee nodes, then traverse the PERFORMED relationships to retrieve all jobs. The second operation requires no lookups, as it happens in the case of a relational database, resulting in a difference of performance that grows higher the further the traversing gets.

Let's establish beforehand that a model can only be considered suitable and performant for a given set of queries. It is impossible for any model to be ideal for every single query, thus, deciding on what queries matter and need to perform is extremely important when designing a database model.

The next chapter addresses the queries that were considered important to work with.

Chapter 5

Query Writing

Up to this point, both database models are defined and the next step should be focusing on how do we want to query the databases. In this chapter, we'll present and go through the process of writing five different queries, each with a separate purpose. It should also be noted that none of the queries written have the purpose of evaluating insertion and/or update performance, and it would be interesting to do such evaluation as future work, since such operations could have opposite results when compared to the ones obtained for this dissertation.

5.1 Deciding on what queries

Before presenting the written queries it is important to clarify that we chose these over so many others because they reflected the day-to-day of the company which proposed this dissertation. It is quite possible that for many other companies, these types of queries do not represent their day-to-day and this should be taken into account when using the results of this dissertation to serve as the basis for any decision. In order to put both database capabilities to the test, we started by writting queries that traverse most of the database structure. In the relational database model, it would mean going across most tables, while in the graph database model it would mean going through nodes and relationships. By observing Figures 4.2 and 4.4, notice that the central entities in both models are Employees, Jobs and Experiences&Exposures. Therefore, we opted to write queries that traverse most of those tables or nodes, if not all of them. Examples of queries which satisfy said conditions are :

- **Query 1** - For each Employee, return all Experiences&Exposures they claim they have, in addition to all those earned from performing jobs.
- **Query 2** - For each Manager, return all Experiences&Exposures each Manager's subordinate Employee claims he has, in addition to all those earned from performing jobs.
- **Query 3** - For each Employee, return all Jobs, and their respective Segment, started before a given date.

The query described in Query 1 would have to go through all employees, and, for each one, determine what Experiences&Exposures they claimed, in addition to finding all jobs performed which also grant Experiences&Exposures.

As for the second example, the query is very similar to the previous one since it only adds one extra step of finding all employees who report to a manager, named subordinates. After all subordinates are found, the query should execute in the same way as Query 1.

The query described in the last example is not as heavy as the other two since it only relates the Employee and Job tables, unless we want to add information on jobs' segment names and descriptions. The query is also filtered by a start_date which is an attribute of the employee_job associative entity table or a property of the PERFORMED relationship.

Since the central point of this dissertation is to assess performance differences between both databases when dealing with HR relationships, it would be interesting to have queries that revolve around a company's hierarchy. If we were dealing with a social network, this would mean a query that retrieves friends-of-friends-of-friends would be ideal and, in our case, we can take a similar approach and formulate a query that returns a chain of managers or subordinates, with a given depth.

- **Query 4** - For each Employee, return the identification numbers of all its subordinates-of-subordinates-of-subordinates, with a given query depth.

- **Query 5** - For each Employee, return the identification numbers of all its managers-of-managers-of-managers, with a given query depth.

Now that we decided on the queries we want, it's time to write them in both SQL and Cypher. For each query, all steps taken to formulate them using SQL will be addressed, followed by the explanation on how to write them using Cypher.

5.1.1 Query 1

As mentioned in Section 5.1, Query 1 returns, for each Employee, all Experiences&Exposures earned from performing jobs, in addition to those that employees claim they have.

The first `SELECT` statement, from lines 5-13, will return the first type of E&E's by performing `LEFT JOIN`'s between `experiences_exposures`, `job_ee`, `employee_job` and `employee` tables. The `LEFT JOIN` operation between two tables returns all records from the left table, and the matching records from the right table, so, by adding consecutive `LEFT JOIN`'s we are able to retrieve data related to each other in more than one table. In order to display the E&E's values separated by a comma, in a single row, we use the `STUFF` function together with `FOR XML` in `PATH` mode. It should be noted that using `FOR XML` to concatenate the results into a single line will affect the final execution time because it will take extra time to rearrange the data, however, in order to validate the results as described in Section 6.4, we needed the queries to output data in the exact same manner.

The `PATH` mode with `FOR XML` allows outputting the query results as XML elements, and with a blank string as an argument we get all the values separated by commas. However, it will insert an extra comma right at the beginning and that's where `STUFF` comes in. We pass the output of `FOR XML PATH` as the first argument to the `STUFF` function which will delete a single character from the start of the string. The second and third arguments of the `STUFF` function represent the position from where the deletion will begin and the number of characters to delete, respectively. The last argument is only used when appending a string is intended, so we pass an empty string to it.

The second `SELECT` statement, from lines 15-21, will return the second type of E&E's by performing `LEFT JOIN`'s between `experiences_exposures`, `employee_ee` and `employee` tables, and just like the previous `SELECT` statement, it will arranje the results in a single row, separated by commas.

Listing 5.1: Query 1 written in SQL.

```
1  USE [Relational_1M]
2  SELECT
3     employee.[id] AS employee_id,
4     job_ee = STUFF((
5           SELECT ',' + CONVERT(VARCHAR(MAX), experiences_exposures.[id])
6           FROM [dbo].[job_ee] AS job_ee
7           LEFT JOIN [dbo].[experiences_exposures] AS experiences_exposures
8                 ON job_ee.[ee_id] = experiences_exposures.[id]
9           LEFT JOIN [dbo].[employee_job] AS employee_job
10                ON employee_job.[job_id] = job_ee.[job_id]
11          WHERE employee.[id] = employee_job.[employee_id]
12          GROUP BY experiences_exposures.[id]
13          FOR XML PATH('')), 1, 1, ''),
14    employee_ee = STUFF((
15          SELECT ',' + CONVERT(VARCHAR(MAX), experiences_exposures.[id])
16          FROM [dbo].[employee_ee] AS employee_ee
17          LEFT JOIN [dbo].[experiences_exposures] AS experiences_exposures
18                ON employee_ee.[ee_id] = experiences_exposures.[id]
19          WHERE employee.[id] = employee_ee.[employee_id]
20          GROUP BY experiences_exposures.[id]
21          FOR XML PATH('')), 1, 1, '')
22 FROM [dbo].[employee] AS employee
23 ORDER BY employee_id ASC
```

Writting a query in Cypher is mainly about matching the desired patterns from the database model. In this case we want to match Employee with ExperiencesExposures and, by looking at our model from Figure 4.4, we can draw two possible patterns. The first pattern goes from the Employee node directly to the Experiences&Exposures node by traversing the HAS relationship while the second pattern is longer and traverses from the Employee node to the auxilliary CountryJob node by going through the PERFORMED relationship, followed by another relationship named DESCRIBED_BY which leads to the Job node. When traversing the GRANTS relationship we reach the Experiences&Exposures node and the pattern is complete.

A common mistake would be matching both patterns with two `MATCH` clauses like the example in Listing 5.2 because either the whole pattern is matched, or nothing is matched. This would affect the results of employees who have no E&E's granted by jobs but have claimed some themselves, and since the first pattern will fail, the second one will not be checked.

Listing 5.2: Example of pattern matching in Cypher.

```
1 MATCH (e: Employee {id: e.id})-[:PERFORMED]->(n:CountryJob)-[:DESCRIBED_BY]->(j:Job
      )-[:GRANTS]->(x1:ExperiencesExposures {id: x1.id})
2 MATCH (e: Employee {id: e.id})-[:HAS]->(x2:ExperiencesExposures {id: x2.id})
```

This situation would happen no matter the order of the patterns and the way to avoid this is by matching only employees from the Employee node and carrying the variable over to the rest of the query. The next two lines are `OPTIONAL MATCH` lines that perform just like a `LEFT JOIN` from SQL and will gather results from both patterns regardless of them not being fully matched. We then carry the same employee information to the rest of the query as well as the E&E's identification numbers. Before returning the results, we sort them by ascending order of employee and E&E's identification numbers. Finally, we return `employee_id` and two lists, one for each pattern's set of E&E's. To do so, we use `COLLECT` to create said list, with a `DISTINCT` clause to avoid inserting duplicates into it.

Listing 5.3: Query 1 written in Cypher.

```
1 MATCH (e:Employee {id: e.id})
2 WITH e AS employee
3 OPTIONAL MATCH (employee {id: employee.id})-[:PERFORMED]->(n:CountryJob)-[:
      DESCRIBED_BY]->(j:Job)-[:GRANTS]->(x1:ExperiencesExposures {id: x1.id})
4 OPTIONAL MATCH (employee {id: employee.id})-[:HAS]->(x2:ExperiencesExposures {id:
      x2.id})
5 WITH employee, x1, x2
6 ORDER BY employee.id, x1.id, x2.id ASC
7 RETURN employee.id as employee_id, COLLECT(DISTINCT x1.id) as job_ee, COLLECT(
      DISTINCT x2.id) as employee_ee
```

5.1.2 Query 2

This query is very similar to Query 1 as we can tell by comparing Listings 5.1 and 5.4. The only differences are the field names of both `SELECT` statements since we now want manager identification numbers, and the `LEFT JOIN`'s on the `SELECT` statements in lines 6-16 and 18-26.

The first `SELECT` statement matches E&E's identification numbers, from the experiences_exposures table, with job_ee. Then, ee_id is matched from both job_ee and employee_job tables so that we get employee_id from subordinates to match with their managers.

The second one matches the identification number of experiences_exposures with ee_id of the employee_ee associative entity table that in return is matched with the subordinate employee's identification number.

Listing 5.4: Query 2 written in SQL.

```
USE [Relational_1M]
SELECT
   manager.[manager_id] AS manager_id,
   manager.[employee_id] AS subordinate_id,
   job_ee = STUFF((
         SELECT ',' + CONVERT(VARCHAR(MAX), experiences_exposures.[id])
         FROM [dbo].[job_ee] AS job_ee
         LEFT JOIN [dbo].[experiences_exposures] AS experiences_exposures
               ON job_ee.[ee_id] = experiences_exposures.[id]
         LEFT JOIN [dbo].[employee_job] AS employee_job
               ON employee_job.[job_id] = job_ee.[job_id]
         LEFT JOIN [dbo].[employee] AS subordinate
               ON employee_job.[employee_id] = subordinate.[id]
         WHERE manager.[employee_id] = subordinate.[id]
         GROUP BY experiences_exposures.[id]
         FOR XML PATH('')), 1, 1, ''),
   employee_ee = STUFF((
         SELECT ',' + CONVERT(VARCHAR(MAX), experiences_exposures.[id])
         FROM [dbo].[employee_ee] AS employee_ee
         LEFT JOIN [dbo].[experiences_exposures] AS experiences_exposures
               ON employee_ee.[ee_id] = experiences_exposures.[id]
         LEFT JOIN [dbo].[employee] AS subordinate
               ON employee_ee.[employee_id] = subordinate.[id]
         WHERE manager.[employee_id] = subordinate.[id]
         GROUP BY experiences_exposures.[id]
         FOR XML PATH('')), 1, 1, '')
FROM [dbo].[manager] AS manager
ORDER BY manager_id, subordinate_id ASC
```

Just like the Cypher query written for Query 1, we want to match two possible patterns and we do so by emulating `LEFT JOIN`'s with the help of `OPTIONAL MATCH`. The difference is that now we start from a manager, which is still a row from an Employee node, and the first traversal is to the same Employee node. It is very important that the arrow's direction is facing the manager node, otherwise we would be querying E&E's of his/her managers. The rest of the path remains unchanged and the same logic is applied for the rest of the query.

Listing 5.5: Query 2 written in Cypher.

```
MATCH (e1:Employee {id: e1.id})<-[:REPORTS_TO]-(e2:Employee)
WITH e1 AS manager, e2 AS subordinate
OPTIONAL MATCH (subordinate {id: subordinate.id})-[:PERFORMED]->(n:CountryJob)-[:
    DESCRIBED_BY]->(j:Job)-[:GRANTS]->(x1:ExperiencesExposures {id: x1.id})
OPTIONAL MATCH (subordinate {id: subordinate.id})-[:HAS]->(x2:ExperiencesExposures
    {id: x2.id})
WITH manager, subordinate, x1, x2
ORDER BY manager.id, subordinate.id, x1.id, x2.id ASC
RETURN manager.id, subordinate.id, COLLECT(DISTINCT x1.id) as job_ee, COLLECT(
    DISTINCT x2.id) as employee_ee
```

5.1.3 Query 3

For this query we want the jobs, and respective segments, an employee performed before a given start date. The result set will have three fields: `employee_id`, `job_id` and `segment_id` as we can conclude from lines 3-5 of the `SELECT` statement. Two `LEFT JOIN`'s are needed to match data from three tables, them being: `job`, `employee_job` and `employee`. The `LEFT JOIN` from lines 9-10 selects rows where the identification number from job table matches `job_id` on `employee_job` table. The second `INNER JOIN`, from lines 11-12, selects rows where the employee's identification number from the `employee` table matches `employee_id` from `employee_job` table.

We used a big enough `start_date` parameter so that no rows are excluded from the final result set and sort it by ascending order of employees and jobs' identification numbers. There was no need to `INNER JOIN` both segment and job tables because the latter already has segment's identification number information stored in it.

Listing 5.6: Query 3 written in SQL.

```
1  USE [Relational_1M]
2  SET STATISTICS TIME ON
3  SELECT
4     employee.[id] AS employee_id,
5     job.[id] AS job_id,
6     job.[segment_id] AS segment_id
7  FROM
8     [dbo].[job] AS job
9     LEFT JOIN [dbo].[employee_job] AS employee_job
10       ON employee_job.[job_id] = job.[id]
11    LEFT JOIN [dbo].[employee] AS employee
12       ON employee_job.[employee_id] = employee.[id]
13 WHERE employee_job.[start_date] < '3000-01-01'
14 ORDER BY employee.[id], job.[id] ASC
15 SET STATISTICS TIME OFF
```

First of, we assign start date with the `WITH` clause to be used in the rest of the query and then we have to match the desired pattern. To retrieve information on Jobs, and respective Segments, an Employee has work on we have to go from the Employee node to the CountryJob node through a PERFORMED relationship and traverse a DESCRIBED_BY relationship to the Job node. Lastly, the Segment node is reachable by going through the FROM_SEGMENT relationship. After the pattern is matched, we need to ensure that the jobs were started before the date stored in variable maxdate. That information is stored as a property of the PERFORMED relationship and to access such data we have to name the relationship when we write the `MATCH` pattern, in this case it was named `r` and by typing `r.start_date` we retrieve the needed date value for comparison. To finish the query, save the results of the employee node, job and segment identification numbers by using the `WITH` clause and return the values by ascending order of employee identification number.

Listing 5.7: Query 3 written in Cypher.

```
1  WITH '3000-01-01' AS maxdate
```

```
2 MATCH (e:Employee {id: e.id})-[r:PERFORMED]->(n:CountryJob)-[:DESCRIBED_BY]->(j:Job
      )-[:FROM_SEGMENT]->(s:Segment)
3 WHERE r.start_date < date(maxdate)
4 WITH e, j, s
5 RETURN e.id as employee_id, j.id as job_id, s.id as segment_id
6 ORDER BY employee_id, job_id ASC
```

5.1.4 Query 4

This query returns, for each manager, their subordinates-of-subordinates-of-subordinates. The `SELECT` statement collects and separates the subordinates into columns representing the different depths, being depth 1 direct subordinates and depth 3 subordinates-of-subordinates-of-subordinates. For each depth we'll need an extra `LEFT JOIN` that matches the furthest away subordinate manager_id with the second furthest subordinate employee_id, and so on, up until the starting manager node.

Listing 5.8: Query 4 written in SQL.

```
1  USE [Relational_1M]
2  SELECT DISTINCT
3     manager.[manager_id] AS manager_id
4     ,manager.[employee_id] AS depth_1
5     ,sub_sub.[employee_id] AS depth_2
6     ,sub_sub_sub.[employee_id] AS depth_3
7  FROM [dbo].[manager] AS manager
8     LEFT JOIN [dbo].[manager] AS sub_sub
9        ON manager.[employee_id] = sub_sub.[manager_id]
10    LEFT JOIN [dbo].[manager] AS sub_sub_sub
11       ON sub_sub.[employee_id] = sub_sub_sub.[manager_id]
12 ORDER BY manager_id, depth_1, depth_2, depth_3 ASC
```

Depth querying in Cypher is very simple as we can tell from Listing 5.9. If we intend to query subordinates-of-subordinates-of-subordinates, like in this case, we have to `MATCH` the corresponding path. Therefore, we use the `MATCH` clause and assign the path from line 1 to a path named `p`. We name the start node and end node as `subordinate` and `manager`, respectively, and the REPORT_TO relationship now has extra information on the interval of loops we want for our query. This is represented by writing `[:REPORTS_TO*1..3]` and `*1..3` means that we are interested in hopping at least once and not more than three times. If we wanted to query subordinates-of-subordinates-of-subordinates-of-subordinates, in other words, increment the depth, we would only need to rewrite the relationship as `[:REPORTS_TO*1..4]`. After pattern matching, the initial node and a list of all subordinate identification numbers are passed on to the end of the query with the `WITH` clause and return in ascending order of manager's identification number.

Listing 5.9: Query 4 written in Cypher, using the variable relationship depth operator.

```
1 MATCH p = (subordinate)-[:REPORTS_TO*1..3]->(manager)
2 WITH manager.id as manager_id, COLLECT(subordinate.id) as subordinate_chain_ids
```

```
3 RETURN manager_id, subordinate_chain_ids
4 ORDER BY manager_id ASC
```

In order for query 4 to return data in the same way SQL did, we were required to rewrite the previous Cypher query, which resulted in the one described by Listing 5.10. Each `OPTIONAL MATCH` exists as a way to store intermediate depth data. If we had written a single `MATCH` starting from a manager up to an employee, by crossing the REPORTS_TO relationship three times, we would only obtain employees from depth 3. This happens because the pattern in `MATCH` is either fully matched or nothing is returned at all. By adding an `OPTIONAL MATCH` for each intermediate depth, we are able to return data from employees of all hierarchic depths. To filter out managers with no subordinates, line 7 was included.

Listing 5.10: Query 4 written in Cypher by using OPTIONAL MATCH.

```
1 MATCH (e:Employee {id: e.id})
2 WITH e AS manager
3 OPTIONAL MATCH (manager)<-[:REPORTS_TO]-(depth1)
4 OPTIONAL MATCH (manager)<-[:REPORTS_TO]-(depth1)<-[:REPORTS_TO]-(depth2)
5 OPTIONAL MATCH (manager)<-[:REPORTS_TO]-(depth1)<-[:REPORTS_TO]-(depth2)<-[:
      REPORTS_TO]-(depth3)
6 WITH manager.id AS manager_id, depth1.id AS depth_1, depth2.id as depth_2, depth3.
      id as depth_3
7 WHERE depth_1 <> 'null'
8 RETURN manager_id, depth_1, depth_2, depth_3
9 ORDER BY manager_id, depth_1, depth_2, depth_3 ASC
```

5.1.5 Query 5

This query does basically the same as Query 4 in Section 5.1.4 but in the opposite direction since we now start from a subordinate and want to return, for each subordinate, their managers-of-managers-of-managers. The `SELECT` statement collects and separates the managers into columns representing the different depths, depth 1 being direct managers and depth 3 managers-of-managers-of-managers, just like the previous query. Each depth will also translate into an extra `LEFT JOIN` that matches the furthest away manager `employee_id` with the second furthest manager `manager_id` , and so on, up until the starting subordinate node.

Listing 5.11: Query 5 written in SQL.

```
1 USE [Relational_1M]
2 SELECT DISTINCT
3    employee.[employee_id] AS subordinate_id
4    ,employee.[manager_id] AS depth_1
5    ,man_man.[manager_id] AS depth_2
6    ,man_man_man.[manager_id] AS depth_3
7 FROM [dbo].[manager] AS employee
8    LEFT JOIN [dbo].[manager] AS man_man
9       ON employee.[manager_id] = man_man.[employee_id]
```

```
10    LEFT JOIN [dbo].[manager] AS man_man_man
11       ON man_man.[manager_id] = man_man_man.[employee_id]
12 ORDER BY subordinate_id, depth_1 ,depth_2, depth_3 ASC
```

For the last query, we want the opposite case of Query 4 and to accomplish that we just have to save the subordinate node instead of the manager node and list the managers instead of the subordinates like shown in Listing 5.12.

Listing 5.12: Query 5 written in Cypher, using the variable relationship depth operator.

```
1 MATCH p = (subordinate)-[:REPORTS_TO*1..3]->(manager)
2 WITH subordinate.id as subordinate_id, COLLECT(manager.id) as manager_chain_ids
3 RETURN subordinate_id, manager_chain_ids
4 ORDER BY subordinate_id ASC
```

For the same reason given for query 4, we rewrote query 5 described by Listing 5.13 to return the exact same data as the SQL query.

Listing 5.13: Query 5 written in Cypher.

```
1 MATCH (e:Employee {id: e.id})
2 WITH e AS employee
3 OPTIONAL MATCH (employee)-[:REPORTS_TO]->(depth1)
4 OPTIONAL MATCH (employee)-[:REPORTS_TO]->(depth1)-[:REPORTS_TO]->(depth2)
5 OPTIONAL MATCH (employee)-[:REPORTS_TO]->(depth1)-[:REPORTS_TO]->(depth2)-[:
    REPORTS_TO]->(depth3)
6 WITH employee.id AS employee_id, depth1.id AS depth_1, depth2.id as depth_2, depth3
    .id as depth_3
7 WHERE depth_1 <> 'null'
8 RETURN employee_id, depth_1, depth_2, depth_3
9 ORDER BY employee_id, depth_1, depth_2, depth_3 ASC
```

However, if we want the query to look almost exactly like the one in Listing 5.9, all we have to do is invert the arrow direction in the `MATCH` pattern like shown in Listing 5.14. The only counterpart would be the names of node making no sense, even though the result set would be correct for what we wanted to retrieve.

Listing 5.14: Alternate way of writing Query 5 using Cypher.

```
1 MATCH p = (subordinate)<-[:REPORTS_TO*1..3]-(manager)
2 WITH manager.id as manager_id, COLLECT(subordinate.id) as subordinate_chain_ids
3 RETURN manager_id, subordinate_chain_ids
4 ORDER BY manager_id ASC
```

After presenting all queries, both in SQL and in Cypher, it is impossible not to notice how compact the Cypher queries are. For example on queries 1 and 2, we could write the equivalent of a `SELECT` statement with multiple `LEFT JOIN`'s in a single line of code. Not to mention that single line of code is formulated with the use of ASCII art which allows representing nodes with parenthesis and relationships with arrows and square brackets, as shown in line 1-2 of Listing 5.2, making the Cypher query much more readable than the corresponding SQL query, without

compromising expressability. The last two queries that retrieve the hierarchy of employees and their managers can be obtained with just four lines of code versus the over a dozen lines that come from joining multiples tables to retrieve relationship between all of them. Having such concise queries will help prevent errors because of its reduced number of lines and this also helps new developers that are just starting to work with these queries to better understand them and what they are trying to accomplish. This of course does not mean SQL queries are absolutely unreadable, but when comparing all of our examples, the ease-of-read of cypher queries is remarkable.

Chapter 6

Query Evaluation

In this chapter we're going to discuss how data was generated and then imported to both databases. Then, we decide how queries are going to be evaluated, and their results validated, as well as how to extract such information. Finally, we present the obtained results upon executing the queries described in the previous chapter, in order to draw conclusions about each query's performance.

6.1 Data generation

To avoid using real, yet sensitive, HR company data, we instead generate our data which will populate both databases. It would be bad practice to have a single database for a fixed data set size, therefore, the database structure scripts presented in Chapter 4 were executed with the intent of creating multiple databases with data of different set sizes. In order to streamline data generation, parameters were defined for the number of rows of each table, as shown in Listing 6.1, and the presented values were often discussed with the company's representative and changed to balance data proportion, in order to keep such proportion similar to a real scenario.

Listing 6.1: Code snippet for parameterization of data set sizes.

```
1  NUM_COUNTRIES = 197 # There are a total of 239 countries all over the world
2  NUM_RATINGS = 4 # There are a total of 4 ratings (A, B, C and D)
3  NUM_SEGMENTS = 800 # Total of segments to be created
4  NUM_JOBS = 5 * NUM_SEGMENTS # Total of jobs to be created
5  NUM_EE = 5 * NUM_SEGMENTS # Total of experiences and exposures to be created
6  NUM_EMPLOYEES = 1000000 # Total of employees to be created
7  NUM_MANAGERS = NUM_EMPLOYEES # Total of rows in table 'manager'
8  NUM_EMPLOYEE_JOB = NUM_EMPLOYEES # Total of rows in table 'employee_job'
9  NUM_EMPLOYEE_EE = NUM_EMPLOYEES # Total of rows in table 'employee_ee'
10 NUM_EMPLOYEE_RATING = NUM_EMPLOYEES # Total of rows in table 'employee_rating'
11 NUM_JOB_EE = 5 * NUM_JOBS # Total of rows in table 'job_ee'
```

There are eleven tables in total, thus, eleven data files should be generated. We chose the CSV format since it is one of the most common data formats and both SQL Server and Neo4j Desktop can import such files.

To help distinguish all databases from each other, they had their name appended with the amount of employees they hold, for example, a database with a hundred thousand employees would be called Database_100k.

A CSV file allows data to be saved in a table structured format with values separated by commas. Let's take Listing 6.2 as an example of what generating a CSV data file would look like.

Listing 6.2: Code snippet for employee table data generation.

```
# EMPLOYEE: [ id, first_name, last_name, email, gender, nationality, dob ]
def generate_employee():
    # Declaration of the list used to store all 'employee.csv' rows
    employees = []
    # Relative path of 'employee_list.txt', containing first and last names
    filename = os.path.join(dirname, 'Auxilliary Files/employee_list.txt')
    # Open and read to the CSV file
    with open(filename, 'r') as file:
        employee_list = file.readlines()
    # Prepare the first line to contain all employee.csv header information
    employees.append('id,first_name,last_name,email,gender,nationality,dob\n')
    # Generate NUM_EMPLOYEES employees to be stored in 'employee.csv'
    for i in range(1, NUM_EMPLOYEES + 1):

        # Auxilliary calculations for randomly generated data
        email_aux = random.randint(1, 3)
        if email_aux == 1:
            email_provider = '@gmail.com'
        elif email_aux == 2:
            email_provider = '@hotmail.com'
        else:
            email_provider = '@yahoo.com'

        # Choose a random gender from 'gender_list'
        gender_list = ['Male', 'Female']
        random_gender = random.randint(0, len(gender_list) - 1)

        # Generate a random date between 'min_date' and 'max_date'
        min_date = date(year=1960, month=1, day=1)
        max_date = date(year=2003, month=1, day=1)
        time_between_dates = max_date - min_date
        days_between_dates = time_between_dates.days
        random_number_of_days = random.randrange(days_between_dates)
        random_date = min_date + timedelta(random_number_of_days)

        # Current employee information to be stored
        id = str(i)
        first_name = employee_list[i].split(';')[0].lstrip()
        last_name = employee_list[i].split(';')[1].rstrip() # rstrip() removes '\n'
        email = first_name.lower() + id + last_name.lower() + email_provider
        gender = gender_list[random_gender]
```

```
42          nationality = str(random.randint(1, NUM_COUNTRIES))
43          dob = random_date.strftime('%Y-%m-%d')
44          employees.append(id + ',' + first_name + ',' + last_name + ',' + email + ','
                + gender + ',' + nationality + ',' + dob + '\n')
45
46      # Relative path of 'employee.csv'
47      filename = os.path.join(dirname, 'Generated CSV Data/employee.csv')
48      # Open and write to the CSV file
49      with open(filename, 'w') as csv_file:
50          for row in employees:
51              csv_file.write(row)
```

The first comment of Listing 6.2 on line 1 specifies the file data structure. Since in this example we're generating data for the employee table, we want a CSV file with information on the employee's id, first and last name, email, gender, nationality and date-of-birth. It is also important that values generated for each field name are meaningful and not scrambled, repetitive characters. If we named all employees 'John Doe', even though they would have different identification numbers, validating the results would become less immediate. That is why we used a sample database provided by Microsoft, named AdventureWorks, containing tens of thousands of first and last names. By performing a `CROSS JOIN` on both tables and exporting a portion of all rows as a CSV file, we were able to have millions of employees with unique names. The script will then access that file location and read its lines to a list variable named `employee_list`.

The employee CSV file will contain header information described in line 11 and, for a total of `NUM_EMPLOYEES`, we generate a random email provider, gender and date-of-birth. To ensure that all emails are unique we concatenate the employee's first name with its identification number, followed by their last name and email provider. Date-of-births are randomly generated between January 1st, 1960 and January 1st, 2003, while the nationality integer which will be used as a foreign key to the country table is a number between 1 and the total number of countries, `NUM_COUNTRIES`. All the generated lines are stored in a list named `employee` and written to a CSV file as shown in lines 46-51.

The approach for other tables is very similar and will not be discussed, but, in summary, generating the data can be divided into three parts: reading from auxilliary files if needed, generating random values for each table's field names and then writing them to a CSV file. The associative entity tables are much simpler to generate because most data is basically obtained by generating random integers which will be interpreted as foreign keys. At the end of the Python script, all defined functions are called, one for each table.

6.2 Data import

After we have data sets for all the created databases, the next step is importing the data from the generated CSV files which can be used, without a change, for both relational and graph databases.

However, to address the hyperedge situation described in Section 4.2.1, a CSV file named 'countryjob.csv' was generated based on the contents of the file containing information on the relationships between employees, jobs and countries, named 'employee_job.csv'. This extra file takes into consideration the fact that there is no need to create repeated relationships for employees performing the same job in the same country. For example, if both employee A and employee B perform the job J in country C, we generate one relationship from each employee to the auxilliary 'CountryJob' node, but only one relationship needs to be created between 'CountryJob' and 'Job' nodes, as well as between 'CountryJob' and 'Country' nodes. To import the generated CSV files into a SQL Server relational database, we can use the 'Import Data...' option in SQL Server Management Studio's Object Explorer tool, shown in Figure 6.1. It will open the 'SQL Server Import and Export Assistant' where we need to select 'Flat File Source' as our source of data and then search for the CSV file we wish to import. Before proceeding, we have to make sure all the columns are correctly displayed by choosing the Columns option on the left bar as well as verifying if the field names have the right data types assigned to them. In this case, all tables with a description field name were required to have their data type changed from `string [DT_STR]` to `text stream [DT_TEXT]`. Failing to do so will result in data truncation or even import failure. We then choose 'SQL Server Native Client' and the corresponding database as our data destination and click 'Edit Mapping...' in case we have to check the 'Allow for identity insertion' box when dealing with main domain tables which typically have an identity field name.

This process is repeated for every table, and therefore for every CSV file. The order we choose to load data matters since, for example, we can't start adding job's table values without first adding segment's table values because job has foreign keys referencing segments.

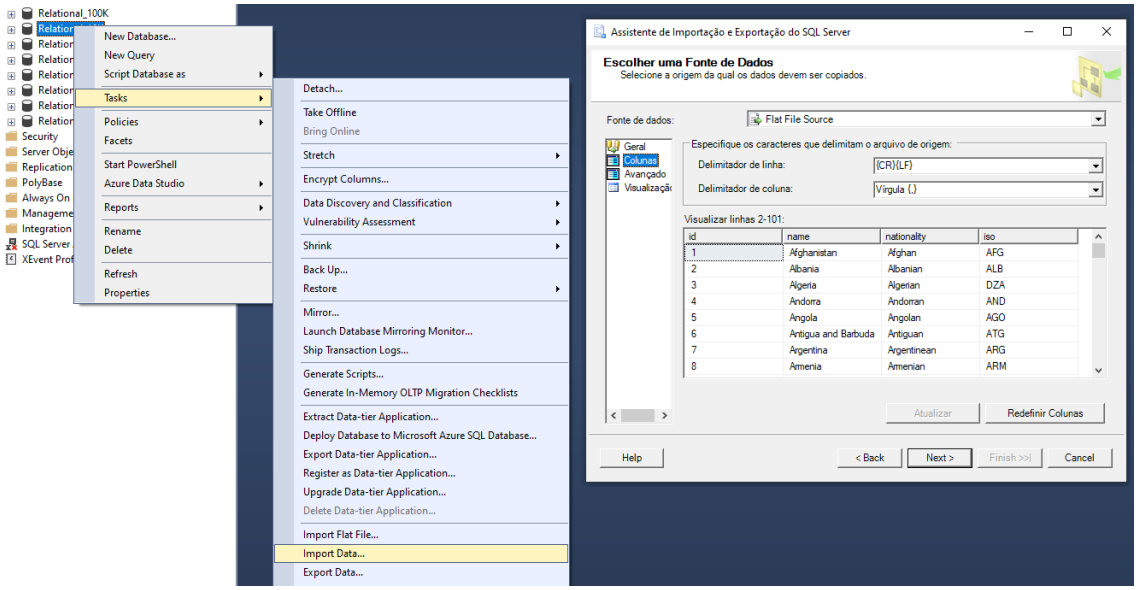

Figure 6.1: Importing data using SQL Server Management Studio's Object Explorer tool.

As for importing data to a Neo4j graph database, it allows CSV importing just like SQL Server.

Each database has a folder containing configuration files, report and crash logs, plugins and they also have an import folder where the user can place any CSV files. That import folder can be accessed by clicking the button displayed in Figure 6.2, and all CSV files can be placed there.

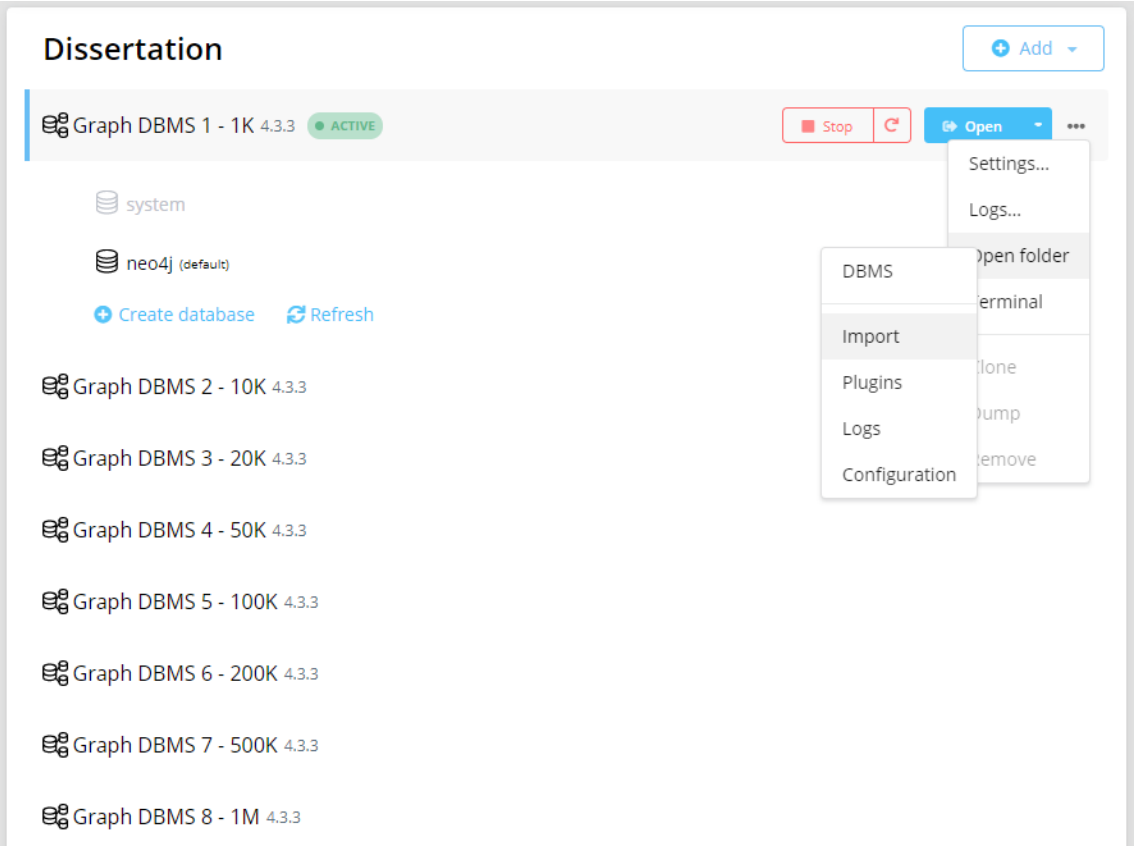

Figure 6.2: Access the graph database import folder using Neo4j Desktop interface.

When all CSV files are in the correct 'import' folder, the final step is to execute a query that reads the files from said directory and starts adding information to nodes and relationships. All of our import CSV queries have three parts, each represented by a single line as Listing 6.3 shows.

The first is not mandatory, but advisable, because with large enough CSV files (hundreds of thousands or millions of rows), loading the entire collection of data in one sitting will not be as performant when compared to an import that has a periodic commit set up. What periodic commit does is, as the name suggests, performing a commit after a given number of rows were read, one thousand in our case. Next comes the `LOAD CSV` command, and since we included the `WITH HEADERS` option, it will read the first CSV line in search for field names. Still in the same line, we add `FROM "file:///employee.csv" AS row` which specifies the CSV file name, in this case `employee.csv`, that should already be present in the 'import' folder discussed earlier. Lastly comes the part where we use the `CREATE` or `MERGE` clause to add nodes and/or relationships to the graph database, from the CSV data read.

In our examples, we always used `MERGE` instead of `CREATE` because as explained in Section 2.2.6, `MERGE` will only execute if the node or relationship does not exist in the database, while `CREATE` will add it regardless. In essence, `MERGE` is a combination of `MATCH` and `CREATE`. In this example we are saying that the new Employee nodes will have an identification number called `id` which will take on the values of `row.id` read from the CSV. We have to use the right data types when creating nodes and relationships because, for example, `row.id` is a string and we want it converted to an integer by using `toInteger()` before assigning it to the node property. Dates are handled in the same way, with the use of `date()` method.

Listing 6.3: Cypher query to create the Employee nodes.

```
1 :auto USING PERIODIC COMMIT 1000
2 LOAD CSV WITH HEADERS FROM "file:///employee.csv" AS row
3 MERGE (e:Employee {id: toInteger(row.id), first_name: row.first_name, last_name:
      row.last_name, email: row.email, gender: row.gender, dob: date(row.dob)})
```

When creating a new relationship, the third query part is now different since we first need to use `MATCH` so that both start and end nodes of the relationship are retrieved. As shown in Listing 6.4, we want to add PERFORMED relationships and, as we already know, they link both 'Employee' and 'CountryJob' nodes. Therefore, we first match the 'Employee' nodes by retrieving `row.employee_id` from the CSV file and finding it in the database. The same happens for the 'CountryJob' nodes and, to end the query execution, `MERGE` from line 5 will create the relationships between the matched nodes with both start and end dates, retrieved from `row.start_date` and `row.end_date` in the CSV file, respectively, as relationship properties.

Listing 6.4: Cypher query to create the Employee PERFORMED relationships.

```
1 :auto USING PERIODIC COMMIT 1000
2 LOAD CSV WITH HEADERS FROM 'file:///country_job.csv' AS row
3 MATCH (e: Employee {id: toInteger(row.employee_id)})
4 MATCH (n: CountryJob {id: toInteger(row.id)})
5 MERGE (e)-[:PERFORMED {start_date: date(row.start_date), end_date: date(row.
      end_date)}]->(n)
```

After importing the data from all CSV files, it's possible to view the newly added node labels, relationship types and property keys when using Neo4j Browser to interact with the database. In the next section, the process of executing queries is presented. To finish this section, we present Table 6.1 which contains values for data import execution times with regards to the database with one million employees and ten million manager relantionships and, as we can see from observing its contents, inserting CSV data to Neo4j using the `LOAD CSV` command takes longer than importing them to SQL Server. This difference was very high for the manager CSV file, which contained the most rows, and Neo4j needed over 20 times more time than SQL Server to import the whole file.

Table 6.1: Data import execution times for a database with a million employees.

CSV Files		Data Import Execution Times	
File Name	# Rows	SQL	Neo4j
Employee	1 000 000	16 017 ms	113 158 ms
Job	2 000	32 ms	233 ms
E&E	2 000	19 ms	78 ms
Segment	400	23 ms	93 ms
Country	197	7 ms	218 ms
Rating	4	4 ms	46 ms
Manager	10 000 000	37 714 ms	862 144 ms
Employee_E&E	1 000 000	8 170 ms	94 804 ms
Employee_Job	1 000 000	5 975 ms	-
Employee_Rating	1 000 000	4 968 ms	88 288 ms
Job_E&E	10 000	78 ms	921 ms
CountryJob	1 000 000	-	173 120 ms

6.3 Query execution

As for query execution, SQL Server Management Studio comes with a query editor where we can both write our queries and also execute them through the same interface, as displayed by Figure 6.3. In order to export the results to a .rpt file, which is basically a CSV file with a different extension, we can right-click the query editor and select 'Results To File' or simply 'Ctrl+Shift+F'. To obtain query execution times, we had already set the `STATISTICS TIME` execution setting on, but we can do it by accessing 'Query Options...' when right-clicking the query editor. In the 'Results' tab we changed the column delimiter character to be a semicolon and we selected the 'Include column headers in the result set' to help with data validation discussed in the next section.

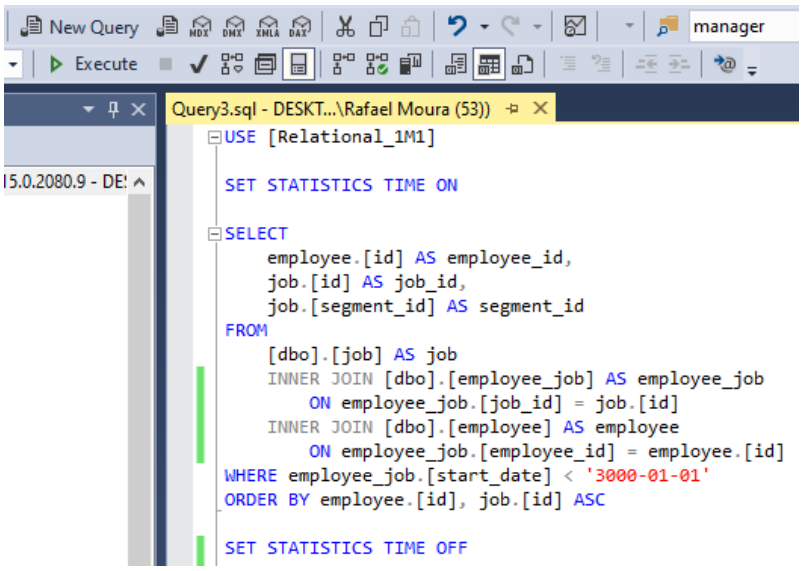

Figure 6.3: SQL Server Management Studio's query editor interface.

Neo4j, on the other hand, has two options we explored: Neo4j Browser and Neo4j Cypher Shell command-line interface, introduced in Sections 3.1.2.1 and 3.1.2.2, respectively. We can execute Cypher queries just as we would in SQL Server Management Studio's query editor. We write it into the edit pane and click the execute button. Running the same query in Cypher Shell is possible by first authenticating to the open database, and then writing the intended query terminated with a semicolon. However, to avoid writing the queries over and over every time we want to execute them, we can create a .cypher file which automatically queries the database when read. In order to do so, the .cypher file needs to be placed in the database's bin directory, which exists in the same folder as the 'import' described in Section 6.2, and executing it by running the commands in Listing 6.5. The commands from lines 2-6 will run the queries from .cypher files in the bin directory, after running cypher-shell.bat and authenticating to the database with `-u` and `-p` flags representing the values of username and password, respectively. Lastly, `--format verbose` is used to display results in tabular format and to print statistics.

Listing 6.5: Commands for the Cypher Shell CLI to execute the queries.

```
1 cd bin
2 type "query1.cypher" | cypher-shell.bat -u neo4j -p password --format verbose
3 type "query2.cypher" | cypher-shell.bat -u neo4j -p password --format verbose
4 type "query3.cypher" | cypher-shell.bat -u neo4j -p password --format verbose
5 type "query4.cypher" | cypher-shell.bat -u neo4j -p password --format verbose
6 type "query5.cypher" | cypher-shell.bat -u neo4j -p password --format verbose
```

The content of 'query1.cypher' is displayed in Listing 6.6, and as we can tell, the query is different from before. In lines 1-8 we store the original query as `query` by using the `WITH` clause. We then call on `apoc.export.csv.query` which is one of hundreds of procedures Neo4j's APOC library has to offer. The library can be installed by selecting the database 'Plugin' tab in Neo4j Browser and this procedure will export the query results to the provided file while returning statistics to the terminal. This procedure in particular requires the user to add an 'apoc.conf' file with a single `apoc.export.file.enabled=true` line to the 'conf' directory in the database folder, in order to allow CSV exporting.

Listing 6.6: Content of 'Query 1.cypher' file.

```
1  WITH "MATCH (e:Employee {id: e.id})
2        WITH e AS employee
3        OPTIONAL MATCH (employee {id: employee.id})-[:PERFORMED]->(n:CountryJob)-[:
             DESCRIBED_BY]->(j:Job)-[:GRANTS]->(x1:ExperiencesExposures {id: x1.id})
4        OPTIONAL MATCH (employee {id: employee.id})-[:HAS]->(x2:ExperiencesExposures
             {id: x2.id})
5        WITH employee, COLLECT(DISTINCT x1.id) + COLLECT(DISTINCT x2.id) AS full_list
6        WHERE SIZE(full_list) > 0
7        RETURN employee.id, full_list
8        ORDER BY employee.id ASC" AS query
9  CALL apoc.export.csv.query(query, "../bin/Query 1.csv", {})
10 YIELD file, source, format, nodes, relationships, properties, time, rows, batchSize
       , batches, done, data
```

```
11 RETURN file, source, format, nodes, relationships, properties, time, rows,
       batchSize, batches, done, data;
```

6.4 Data validation

In this section, we will discuss how data retrieved from queries were validated, that is, making sure the result set is entirely correct, and of course, is what we wished for. In this aspect, Neo4j Browser is a clear winner because it can display the results in graph format which you can interact with and expand on, like portrayed in Figure 6.4. This section also explains the process of executing a query for both relational and graph databases.

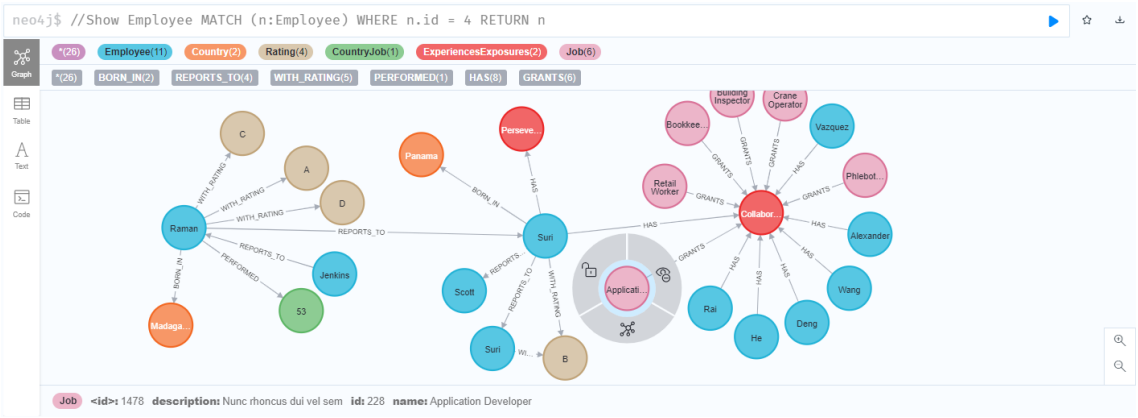

Figure 6.4: Neo4j Browser graph visualization interface example.

By being able to view and interact with your data the way Neo4j Browser allows you to, it becomes so much more simple to verify if queries are returning what you would expect them to. In comparison, doing the same thing in SQL Server is exhausting since it would mean going through all the tables and, in tabular format, manually matching data between them for each row in the result set.

Even though it's very practical and informative to use Neo4j Browser as a way of validating if our queries are returning what they were supposed to, a better solution is to create a script that takes two files as an input, containing query results from each database, in order to return the differences between the files. That script is described in Listing 6.7, and in order for it to work, the following conditions need to be met:

- A folder named 'Query Results' needs to exist in the same directory as the validation script.
- Both CSV files are to be placed in the folder from the previous item.
- The first line of both CSV files is exclusively for header field names. Those names need to appear in the same order and be written exactly the same way for the script to work. If a

different number of headers is detected beforehand, the script will stop with an appropriate error message.

- The generated CSV file from SQL Server has the .rpt extension, while Neo4j exports data to a .csv file. The script is sensitive to those extensions, but that is easily adjusted in the script in case a different format is to be read.
- Each column data obtained from SQL Server is separated by semicolons and if the columns have multiples values, they have to be separated by commas. Neo4j column delimiter is a comma and it also adds quotes at the start and end of each column data. When a column has multiple values, they are shown between square brackets and separated by commas.
- The script saves the totality of results in memory and, therefore, it will likely result in a memory error if used to compare very large CSV files. The solution would be to make the comparison by chunking the data.
- Upon executing, all lines where a difference was detected are signaled and the total number of errors is returned.

Listing 6.7: Python script for comparing SQL and Cypher query results.

```
1  import sys
2  import os
3
4  dirname = os.path.dirname(__file__)
5
6  def validate_data(filename_sql, filename_neo4j):
7      # Variable used to count the number of differences found between the 2 files
8      errors = 0
9      # Relative path of both files
10     filepath_sql = os.path.join(dirname, 'Query Results/' + filename_sql + '.rpt')
11     filepath_neo4j = os.path.join(dirname, 'Query Results/' + filename_neo4j + '.csv
           ')
12     # Open and read both file contents
13     with open(filepath_neo4j, 'r') as file_neo4j:
14         results_neo4j = file_neo4j.readlines()
15     # Expects and strips off the UTF-8 Byte Order Mark.
16     with open(filepath_sql, 'r', encoding='utf-8-sig') as file_sql:
17         results_sql = file_sql.readlines()
18     # Store headers
19     headers_sql = results_sql[0].split(',')
20     headers_neo4j = results_neo4j[0].split(',')
21     # Store number of header columns
22     n_headers_sql = len(headers_sql)
23     n_headers_neo4j = len(headers_neo4j)
24     # Check for different number of header columns
25     if n_headers_sql != n_headers_neo4j:
26         return -2
```

```
27    # Check for different header column names
28    else:
29       # Remove trailing newline from the last header column name
30       headers_neo4j[n_headers_neo4j-1] = headers_neo4j[n_headers_neo4j-1].rstrip()
31       headers_sql[n_headers_sql-1] = headers_sql[n_headers_sql-1].rstrip()
32       # [1:-1] because we want to strip the string from the double quotes
33       for i in range(n_headers_sql):
34          if headers_sql[i] != headers_neo4j[i][1:-1]:
35             return -1
36       # Data validation
37       for i in range(1, len(results_neo4j)):
38          # Get both SQL and Cypher rows
39          neo4j_row = results_neo4j[i].split(',')
40          sql_row = results_sql[i].split(',')
41          # Remove trailing newline from the last column name
42          neo4j_row[n_headers_neo4j-1] = neo4j_row[n_headers_neo4j-1].rstrip()
43          sql_row[n_headers_sql-1] = sql_row[n_headers_sql-1].rstrip()
44          # Compare current row's column values
45          for j in range(n_headers_sql):
46             if (neo4j_row[j][1:-1] == '' and sql_row[j] == 'NULL') or (neo4j_row[j
                   ][1:-1] == sql_row[j]):
47                pass
48             else:
49                print('Mismatch on line ' + str(i+1))
50                errors += 1
51       return errors
52
53 def main():
54
55    args = sys.argv[1:]
56
57    if len(args) == 2:
58       filename_sql = sys.argv[1]
59       filename_neo4j = sys.argv[2]
60       validation = validate_data(filename_sql, filename_neo4j)
61       if validation == -2:
62          print('Number of header columns do not match.\n')
63       elif validation == -1:
64          print('Header column names do not match.\n')
65       else:
66          print('Validation completed with ' + str(validation) + ' errors.\n')
67    else:
68       print('Incorrect number of arguments.\nUsage: $ python validate_data.py <
             filename_sql> <filename_neo4j>\nNote: Both files must be located in the
             \'Query Results\' folder.\n')
69
70 if __name__ == "__main__":
71    main()
```

When writting the data validation script, we realised that it was better to rewrite all queries so that they would return the results in the exact same manner for each row and columns. Up until then we had, for example, Neo4j returning all identification number of jobs an employee performed, in a single line, while in SQL Server, we displayed information in multiple lines. This would require the script to be very flexible and, consequently, more complex.

6.5 Results

In this section, we present tables containing execution time results obtained from running all five queries against eight databases, each with a different data set size.

- **Query 1** - For each Employee, return all Experiences&Exposures they claim they have, in addition to all those earned from performing jobs.
- **Query 2** - For each Manager, return all Experiences&Exposures each Manager's subordinate Employee claims he has, in addition to all those earned from performing jobs.
- **Query 3** - For each Employee, return all Jobs, and their respective Segment, started before a given date.
- **Query 4** - For each Employee, return the identification numbers of all its subordinates-of-subordinates-of-subordinates, with a given query depth, in our example, depth 3.
- **Query 5** - For each Employee, return the identification numbers of all its managers-of-managers-of-managers, with a given query depth, in our example, depth 3.

The relational and graph queries were executed in SQL Server Management Studio's query editor and Neo4j Cypher Shell command-line interface, respectively. The statistic reports are shown by Figure 6.5, for the Neo4j case, and Listing 6.8, for the SQL Server case.

```
+-------------------------------------------------------------------------------------
| file                                | source                | format | nodes | rela
+-------------------------------------------------------------------------------------
| "../bin/[E&E] Employee_results.csv" | "statement: cols(2)" | "csv"  | 0     | 0
+-------------------------------------------------------------------------------------

1 row
ready to start consuming query after 50 ms, results consumed after another 39244 ms
```

Figure 6.5: Statistic results upon executing Query 1 in Neo4j Cypher Shell.

Listing 6.8: SQL Server query execution report file.

```
1 (1262686 rows affected)
2
3  SQL Server Execution Times:
4    CPU time = 7828625 ms, elapsed time = 8008896 ms.
5
```

```
6 Completion time: 2021-09-15T05:21:29.3241479+01:00
```

Upon query execution, we extracted the obtained results from both databases as CSV files and, for each query, loaded them into the data validation script for comparison. Since for each query, and according to the script, both SQL and Cypher queries' result files had no differences with respect to the data itself, the results were then assumed equivalent.

As a note, the queries were executed through an i7-7700K processor with 4 cores and 8 threads, with 8 GB of available RAM, on a non-dedicated computer. Next we organized the results from Tables 6.2 and 6.3 in data graphics for a better visualization of how the databases perform for every query. It should be noted that the results were obtained after only a few executions and, as future work, they should be retrieved by running the queries several times in order to return an average query execution time and its deviation interval.

Table 6.2: Query execution times on Microsoft SQL Server.

Databases	SQL Server Queries				
	1	2	3	4	5
1K	280 ms	292 ms	31 ms	40 ms	42 ms
10K	2 525 ms	1 911 ms	115 ms	117 ms	163 ms
20K	5 659 ms	4 272 ms	139 ms	197 ms	195 ms
50K	16 366 ms	3 553 ms	248 ms	414 ms	418 ms
100K	42 575 ms	8 751 ms	621 ms	843 ms	808 ms
200K	98 321 ms	20 503 ms	818 ms	1 510 ms	1 549 ms
500K	2 041 847 ms	366 977 ms	1 943 ms	3 531 ms	3 813 ms
1M	8 008 896 ms	1 643 794 ms	6 723 ms	7 294 ms	7 400 ms

Table 6.3: Query execution times on Neo4j Cypher Shell.

Databases	Neo4j Queries				
	1	2	3	4	5
1K	700 ms	547 ms	130 ms	140 ms	50 ms
10K	1 304 ms	904 ms	190 ms	260 ms	130 ms
20K	1 746 ms	1 382 ms	270 ms	330 ms	180 ms
50K	2 902 ms	2 970 ms	380 ms	540 ms	340 ms
100K	4 619 ms	5 622 ms	570 ms	936 ms	664 ms
200K	8 673 ms	10 801 ms	1 012 ms	1 638 ms	1 360 ms
500K	25 103 ms	35 443 ms	2 085 ms	4 487 ms	3 484 ms
1M	39 244 ms	54 863 ms	3 578 ms	8 400 ms	7 084 ms

By observation of Figures 6.6, 6.7, 6.8, 6.9 and 6.10, we can draw some conclusions:

- The difference of performance on queries 1 and 2 increases dramatically as database data set size increases. For the heaviest database of a million employees, the relational database query takes 200 times more to finish execution.
- Regarding queries 4 and 5 there are no major differences in absolute terms. The relational database curiously has a better performance than the graph database for Query 4, no matter

the database size. This suggests we should keep increasing the database set size in order to determine if one million employees is still not big enough for the relational database to keep up with the graph one. Another possibility, since Query 4 heavily depends on the REPORTS_TO relationship in Manager table, would be using, for example, ten times the number of rows of said table. Query 5, just like the previous, depends on the REPORTS_TO relationship but instead of querying a chain of subordinate employees, returns a manager chain, and therefore, should output similar results for query execution times.

- Between queries 1 and 2, the one with the quickest execution time increase, for the relational database, is query 1. However, for the graph database we actually see a bigger increase on query 2, even though the obtained values are still somewhat close for both queries, regarding the graph database.
- There's barely no differences between the performance of query 3 when comparing both databases. The only noticeable difference, in relative terms, is for the database with one million employees since SQL takes almost twice the time to finish the query execution.
- There is not a single query where Neo4j outperforms SQL Server when the database has one thousand employees so we can deduce that for low data set sizes, the relational database has no problems keeping up with the graph database and even has better execution times in relative terms. However, only Query 4 still takes longer for the graph database to execute since, for the remaining queries, the execution times of relational database queries surpasses the graph ones, even if by a minimal absolute margin.

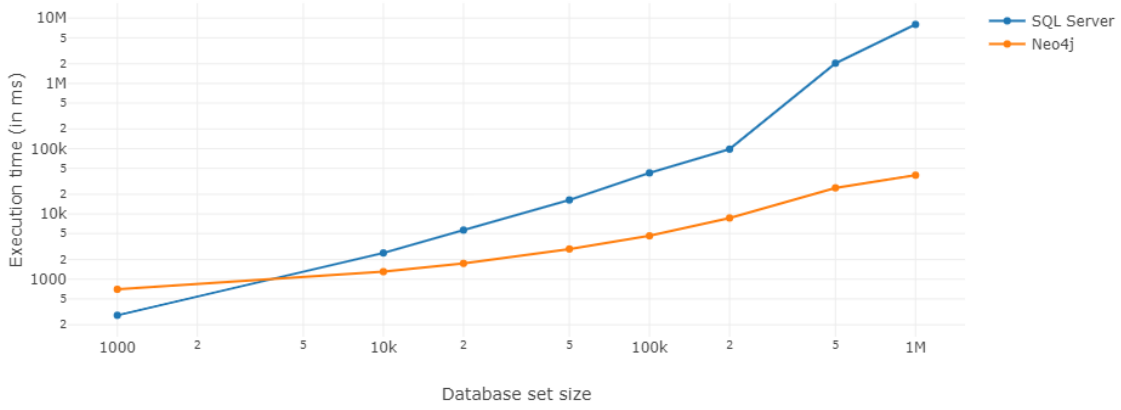

Figure 6.6: Chart with Query 1 execution times, in logarithmic scale.

When querying data which traverses most of the graph structure, as is the case with queries 1 and 2, we can already see a big difference in performance, however, graph databases should be very performant when retrieving hierarchic data. This means we should have been able to notice a disparity of performance from queries 4 and 5 and that was not the case, since for both manager and subordinate chains, the results were pretty much the same. Out of many possible

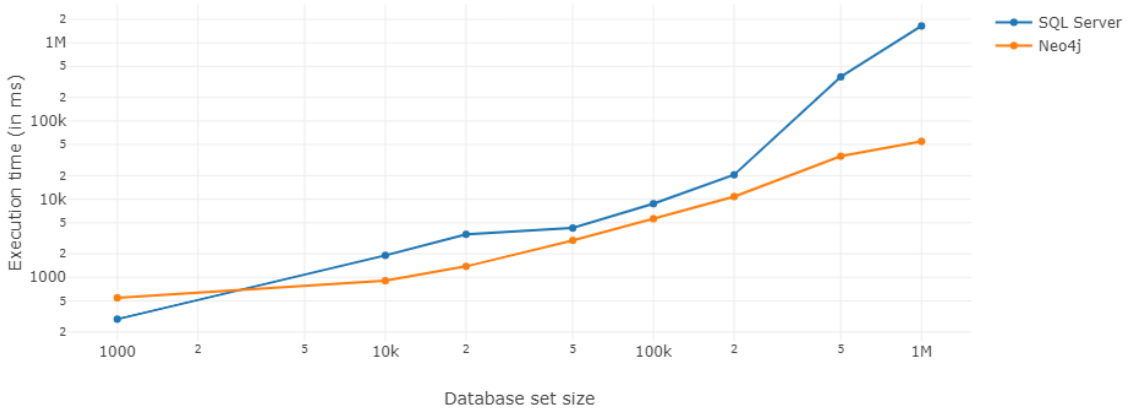

Figure 6.7: Chart with Query 2 execution times, in logarithmic scale.

reasons such as not querying with enough depth to see different results, or having to increase the database set size to tens or hundreds of millions, the one reason that stands out is having a CSV file which stores employee hierarchic data that is not big enough to impact the queries performance between databases. Firstly, we know that for each database we are generating a certain number of employees while also creating a CSV file with the same data set size, that holds relationships between employees. If we build a tree starting from a manager and, for each tree level, we represent the corresponding subordinates, we will notice that, on average, each manager will have a single subordinate. To address this "problem" we multiplied the amount of manager rows by ten, so that now, on average, each manager will have ten employees under his command. As described in query 4 and 5 statements, we want to the return the manager or subordinate chain for all of the one million employees. In our first database with one million manager rows, the result set would be of the same order as the number of employees because as stated before, on average, each manager has a single subordinate. By having ten million manager rows, the result set will increase by ten, depending on the decided query depth. For example, if we want to retrieve

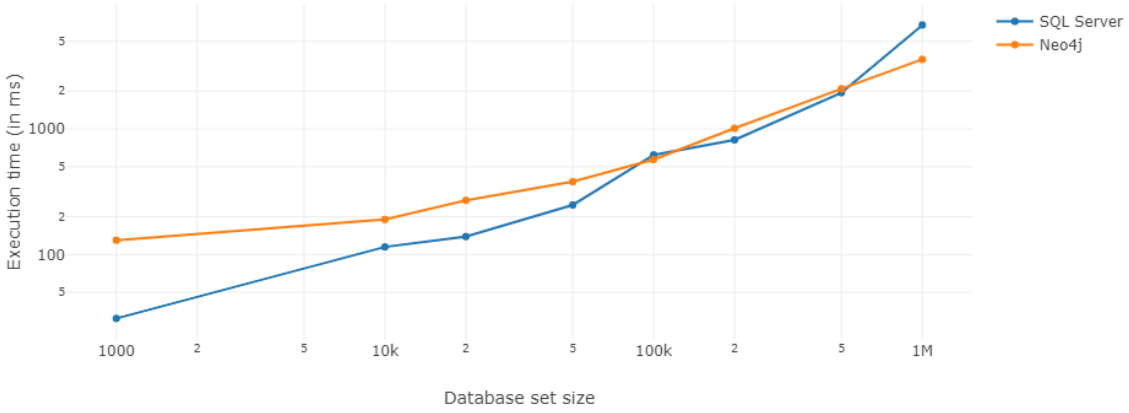

Figure 6.8: Chart with Query 3 execution times, in logarithmic scale.

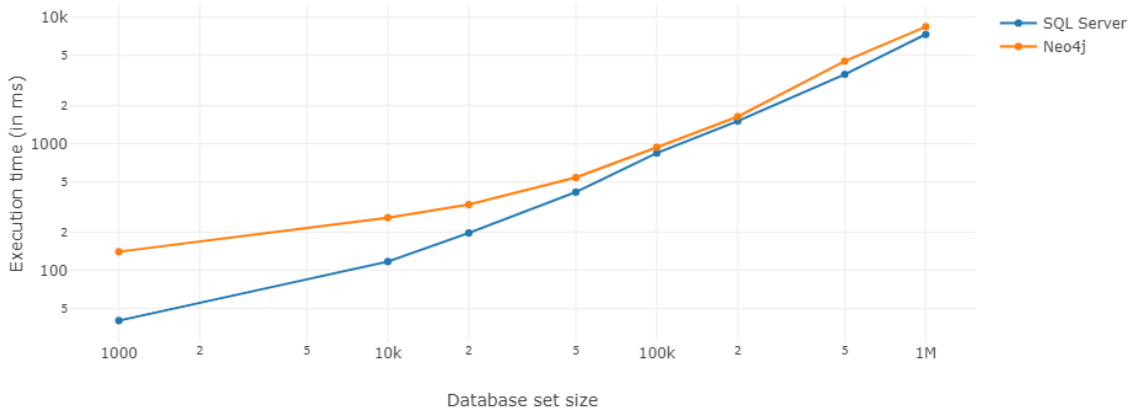

Figure 6.9: Chart with Query 4 execution times, in logarithmic scale.

subordinate chain data for one million employees, up to a depth of four levels, the result set will have approximately ten billion rows, which corresponds to the number of employees multiplied by ten to the power of four, because we're querying for a depth of four. Querying for a depth of five levels would increase the result set size to one hundred billion rows, and so on. Tables 6.4 and 6.5 contain results regarding query 4 and 5 execution times for databases with one million employees and one million manager rows while Tables 6.6 and 6.7 contain results for the same queries, but now associated to databases with ten times the rows of managers from the two previous tables. Instead of executing the queries for the entire employee table, we limited the amount of results to a certain number of employees, otherwise, we would be obtaining billions of rows when querying for depth four, most likely generating memory errors from trying to display or store such a huge quantity of data.

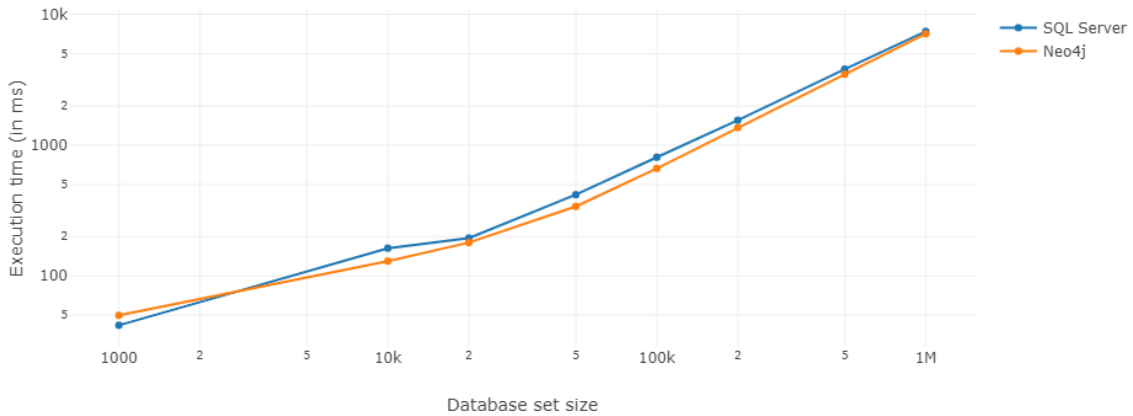

Figure 6.10: Chart with Query 5 execution times, in logarithmic scale.

Table 6.4: Query 4 execution times on Neo4j Cypher Shell and SQL Server, for databases with one million employees and one million manager rows.

Subordinate Chain		DB 1M Employees & 1M Managers		Differences	
		SQL	Neo4j	Absolute	Relative
Depth 1	10K	96 ms	123 ms	- 27 ms	- 21.95 %
	25K	126 ms	220 ms	- 94 ms	- 42.73 %
	50K	201 ms	339 ms	- 138 ms	- 40.71 %
	100K	383 ms	523 ms	- 140 ms	- 26.77 %
	250K	891 ms	870 ms	+ 21 ms	+ 2.41 %
	500K	1 714 ms	1 256 ms	+ 458 ms	+ 36.46 %
	1M	3 570 ms	2 301 ms	+ 1 269 ms	+ 55.15 %
Depth 2	10K	115 ms	360 ms	- 245 ms	- 68.06 %
	25K	181 ms	685 ms	- 504 ms	- 73.58 %
	50K	305 ms	1 160 ms	- 855 ms	- 73.71 %
	100K	610 ms	2 249 ms	- 1 639 ms	- 72.88 %
	250K	1 316 ms	5 549 ms	- 4 233 ms	- 76.28 %
	500K	2 601 ms	11 048 ms	- 8 447 ms	- 76.46 %
	1M	5 004 ms	22 161 ms	- 17 157 ms	- 77.42 %
Depth 3	10K	155 ms	1 129 ms	- 974 ms	- 86.27 %
	25K	240 ms	1 841 ms	- 1 601 ms	- 86.96 %
	50K	419 ms	3 435 ms	- 3 016 ms	- 87.80 %
	100K	793 ms	6 639 ms	- 5 846 ms	- 88.06 %
	250K	2 036 ms	16 074 ms	- 14 038 ms	- 87.33 %
	500K	3 649 ms	32 264 ms	- 28 615 ms	- 88.69 %
	1M	6 775 ms	64 128 ms	- 57 353 ms	- 89.44 %
Depth 4	10K	178 ms	1 754 ms	- 1 576 ms	- 89.85 %
	25K	328 ms	3 421 ms	- 3 093 ms	- 90.41 %
	50K	519 ms	6 662 ms	- 6 143 ms	- 92.21 %
	100K	960 ms	13 083 ms	- 12 123 ms	- 92.66 %
	250K	2 275 ms	32 071 ms	- 29 796 ms	- 92.91 %
	500K	4 352 ms	64 081 ms	- 59 729 ms	- 93.21 %
	1M	8 753 ms	134 880 ms	- 126 127 ms	- 93.51 %

Table 6.5: Query 5 execution times on Neo4j Cypher Shell and SQL Server, for databases with one million employees and one million manager rows.

Manager Chain		DB 1M Employees & 1M Managers		Differences	
		SQL	Neo4j	Absolute	Relative
Depth 1	10K	89 ms	121 ms	- 32 ms	- 26.45 %
	25K	132 ms	183 ms	- 51 ms	- 27.87 %
	50K	305 ms	277 ms	+ 28 ms	+ 10.11 %
	100K	563 ms	432 ms	+ 131 ms	+ 30.32 %
	250K	942 ms	704 ms	+ 238 ms	+ 33.81 %
	500K	1 716 ms	1 175 ms	+ 541 ms	+ 46.04 %
	1M	3 609 ms	2 258 ms	+ 1 351 ms	+ 59.83 %
Depth 2	10K	113 ms	344 ms	- 231 ms	- 67.15 %
	25K	184 ms	725 ms	- 541 ms	- 74.62 %
	50K	299 ms	1 367 ms	- 1 068 ms	- 78.13 %
	100K	555 ms	2 407 ms	- 1 852 ms	- 76.94 %
	250K	1 274 ms	5 753 ms	- 4 479 ms	- 77.86 %
	500K	2 510 ms	11 203 ms	- 8 693 ms	- 77.60 %
	1M	4 985 ms	22 294 ms	- 17 309 ms	- 77.64 %
Depth 3	10K	138 ms	785 ms	- 647 ms	- 82.42 %
	25K	246 ms	1 779 ms	- 1 533 ms	- 86.17 %
	50K	412 ms	3 424 ms	- 3 012 ms	- 87.97 %
	100K	746 ms	6 621 ms	- 5 875 ms	- 88.73 %
	250K	1 773 ms	16 058 ms	- 14 285 ms	- 88.96 %
	500K	3 434 ms	31 914 ms	- 28 480 ms	- 89.24 %
	1M	6 640 ms	64 970 ms	- 58 330 ms	- 89.78 %
Depth 4	10K	185 ms	1 548 ms	- 1 363 ms	- 88.05 %
	25K	301 ms	3 837 ms	- 3 536 ms	- 92.16 %
	50K	560 ms	6 721 ms	- 6 161 ms	- 91.67 %
	100K	926 ms	13 077 ms	- 12 151 ms	- 92.92 %
	250K	2 156 ms	32 080 ms	- 29 924 ms	- 93.28 %
	500K	4 930 ms	64 614 ms	- 59 684 ms	- 92.37 %
	1M	8 702 ms	134 440 ms	- 125 738 ms	- 93.53 %

By inspecting both Tables 6.4 and 6.5 we can observe that there is no difference to the query execution times when retrieving a subordinate or a manager chain. For any number of employees, both databases have the same performance when querying for depth one, however, Neo4j suffers an increase for each extra depth which does not happen to the same extent when comparing with SQL. SQL queries take, in relative terms, between 67% to 93.5% less time when compared to Neo4j queries. This means that if a company is dealing with HR data, but that data is not heavily connected, having just a few relationships connected to each node, SQL will still perform better than Neo4j.

Next, we extracted results for databases with ten times the number of manager rows and they are present in Tables 6.6 and 6.7.

Table 6.6: Query 4 execution times on Neo4j Cypher Shell and SQL Server, for databases with one million employees and ten million manager rows.

Subordinate Chain		DB 1M Employees & 10M Managers		Differences	
		SQL	Neo4j	Absolute	Relative
Depth 1	10K	456 ms	213 ms	+ 243 ms	+ 114. 08 %
	25K	941 ms	521 ms	+ 420 ms	+ 80.61 %
	50K	1 789 ms	959 ms	+ 830 ms	+ 86.55 %
	100K	3 536 ms	1 721 ms	+ 1 815 ms	+ 105. 46%
	250K	8 541 ms	4 623 ms	+ 3 918 ms	+ 84.75%
	500K	17 448 ms	8 162 ms	+ 9 286 ms	+ 113.77 %
	1M	34 716 ms	16 138 ms	+ 18 578 ms	+ 115.12 %
Depth 2	10K	3 782 ms	3 946 ms	- 164 ms	- 4. 16 %
	25K	9 278 ms	9 090 ms	+ 188 ms	+ 2.07 %
	50K	18 380 ms	17 985 ms	+ 395 ms	+ 2.20 %
	100K	36 603 ms	36 719 ms	- 116 ms	- 0.32 %
	250K	101 551 ms	92 592 ms	+ 8 959 ms	+ 9.68 %
	500K	433 852 ms	181 443 ms	+ 252 409 ms	+ 139.11%
	1M	999 372 ms	364 886 ms	+ 634 486 ms	+ 173. 89%
Depth 3	10K	38 532 ms	51 988 ms	- 13 456 ms	- 25.88%
	25K	104 921 ms	130 026 ms	- 25 105 ms	- 19.31 %
	50K	617 297 ms	226 838 ms	+ 390 459 ms	+ 172.13 %
	100K	1 477 813 ms	518 740 ms	+ 959 073 ms	+ 184.89 %
	250K	4 521 733 ms	1 292 298 ms	+ 3 229 435 ms	+ 249.90 %
Depth 4	10K	1 575 695 ms	711 980 ms	+ 863 715 ms	+ 121.31 %
	25K	3 430 355 ms	1 753 348 ms	+ 1 677 007 ms	+ 95.65 %

We can already notice the impact on query execution times when drastically increasing the number of relationships a single node has. By inspecting Table 6.6, we can see that SQL queries went from taking less than ten seconds to execute to last over an hour, in some cases. For depth one, queries take about ten times more than the same queries executed against the database with less manager rows. As for depth two, those differences already go up to two hundred times more and, the deeper we go, the bigger the difference becomes.

Neo4j on the other hand, had negligible differences for small result set sizes and smaller depths, however, they take half the time to execute even for depth one. When increasing the query depth to two, they are almost three times faster and, as expected, the deeper the queries go, the bigger the differences. For the last two depths analysed, we didn't obtain values for querying the entire employee table because it would take a long time but, when retrieving data regarding only a quarter of employees, SQL takes one hour and fifteen minutes to execute while Neo4j does the same in about twenty minutes. Query five behaves in the same way as query four and it is expected that these performance differences get even larger for bigger and more connected databases other than the ones we analysed in this dissertation.

Table 6.7: Query 5 execution times on Neo4j Cypher Shell and SQL Server, for databases with one million employees and ten million manager rows.

Manager Chain		DB 1M Employees & 10M Managers		Differences	
		SQL	Neo4j	Absolute	Relative
Depth 1	10K	477 ms	593 ms	- 116 ms	- 19.56 %
	25K	930 ms	623 ms	+ 307 ms	+ 49.28 %
	50K	1 807 ms	1 016 ms	+ 791 ms	+ 77.85 %
	100K	3 475 ms	1 800 ms	+ 1 675 ms	+ 93.06 %
	250K	8 608 ms	4 147 ms	+ 4 461 ms	+ 107.57 %
	500K	17 288 ms	8 083 ms	+ 9 205 ms	+ 113.88 %
	1M	35 217 ms	16 666 ms	+ 18 551 ms	+ 111.31 %
Depth 2	10K	3 846 ms	4 154 ms	- 308 ms	- 7.41 %
	25K	9 239 ms	8 994 ms	+ 245 ms	+ 2.72 %
	50K	21 424 ms	17 604 ms	+ 3 820 ms	+ 21.70 %
	100K	36 415 ms	35 004 ms	+ 1 411 ms	+ 4.03 %
	250K	103 122 ms	87 875 ms	+ 15 247 ms	+ 17.35 %
	500K	442 670 ms	184 168 ms	+ 258 502 ms	+ 140.36 %
	1M	1 116 247 ms	372 499 ms	+ 743 748 ms	+ 199.66 %
Depth 3	10K	39 862 ms	49 471 ms	- 9 609 ms	- 19.42 %
	25K	108 147 ms	124 463 ms	-16 316 ms	- 13.11 %
	50K	532 120 ms	255 865 ms	+ 276 255 ms	+ 108.30 %
	100K	1 318 871 ms	508 327 ms	+ 810 544 ms	+ 159.45 %
	250K	4 437 561 ms	1 283 629 ms	+ 3 153 932 ms	+ 245.70 %
Depth 4	10K	1 624 959 ms	732 319 ms	+ 892 640 ms	+ 121.89 %
	25K	3 351 445 ms	1 784 178 ms	+ 1 567 267 ms	+ 87. 84 %

Chapter 7

Conclusions and Future Work

In this chapter are presented the conclusions we could take as well as discussing possible future work regarding this dissertation's topic.

7.1 Conclusions

In this section we present conclusions from this dissertation work, in three parts. The first part is related to the database model definition and implementation. When defining the database models and assuming previous experience from SQL and relational databases, it might be discouraging to start by defining the graph database model first. However, upon inspecting both model structures we can tell how straight forward the definition of a graph model is since there are only two types of data: entities represented by nodes and relationships which link those entities.

In regards to query writing, after the initial learning curve is overcome, having ASCII as a way of formulating a query allows for a more visual representation of what we want to query. The central part of a Cypher query is the pattern to be matched and, being able define that path with a single line of ASCII art to represent node and relationships makes the query much more concise and readable without losing expressability. From the set of five queries this dissertation analyses, we observed that all of them need half the lines to write using Cypher when compared to SQL. This helps avoiding errors when writing the queries and they consequently become simpler and more readable for someone that has to pick up on previously done work.

Lastly, the most important part of this dissertation, are conclusions taken from analyzing the obtained statistic results from query execution. Queries 1 and 2 which traversed most of the database structure had the biggest difference in performance for increasing database set sizes. In our database with a million employees, querying the relational database for all E&Es of each Employee took 200 times more than doing the same thing in a graph database. However, we expected to have better results when querying for hierarchic data, that is, a manager or subordinate chain of all employees. The first results obtained for those hierarchic queries were retrieved upon executing them against a database with an average of a single manager/subordinate per employee. When increasing that number by a factor of ten, we were able to see the differences we expected

and were not able to obtain with the previous databases. Neo4j performed better, up to 3.5 times faster than SQL, and due to memory and time constraints, we were not able to get results for queries with a higher depth and higher data set size, however, the results are already indicative that if a company deals with highly connected HR data and is having performance issues with relational databases, graphs could be very helpful for their business.

7.2 Future work

There was one objective we failed to accomplish during this dissertation which was the development of an HTML website to help users with query writing by being able to change filters, in dropdown menus, to query the database. It would be helpful to return to this objective in the future and develop said website. It would also be interesting to, purposefully, think of possible changes to the model in order to assess how flexible to changes both databases really are. Formulating even more queries with different purposes, ideally with more computacional power, would be a possible future work, as well as using actual HR data and modeling the same type of databases for different use cases, for example, a social network company. In this dissertation, there was no detailed information on the difference of performance between insertion and updating queries when applied to relational and graph databases. As stated in Section 5, such information would help understand whether those operations could have opposite results to the obtained ones and, consequently, lead to different conclusions.

References

[1] E.F.Codd. Derivability Redundancy and consistency of relations stored in large data banks Codd.pdf, 1969.

[2] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. URL: https://doi.org/10.1145/362384.362685, doi:10.1145/362384.362685.

[3] E. F. Codd. Evaluation Scheme for Database Management Systems That Are Claimed To Be Relational. pages 720–729, 1986. doi:10.1109/icde.1986.7266284.

[4] E. F. Codd. *The Relational Model for Database Management: Version 2*. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[5] Jim Gray. Transaction Concept: Virtues and Limitations. *Very Large Data Bases, International Conference on Very Large Data Bases*, (1):144–154, 1981.

[6] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. URL: https://doi.org/10.1145/289.291, doi:10.1145/289.291.

[7] Joe Celko. *Graph Databases*. 2014. doi:10.1016/b978-0-12-407192-6.00003-0.

[8] DB-Engines. DB-Engines Ranking - popularity ranking of graph DBMS, 2019. URL: https://db-engines.com/en/ranking/graph+dbms.

[9] G. Klyne and J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, 2004. URL: https://www.w3.org/TR/rdf-concepts/#section-Concepts.

[10] Marko A. Rodriguez and Peter Neubauer. The graph traversal pattern, 2010. arXiv:1004.1001.

[11] Roman Čerešňák and Michal Kvet. Comparison of query performance in relational a non-relation databases. *Transportation Research Procedia*, 40:170–177, 2019. URL: https://doi.org/10.1016/j.trpro.2019.07.027, doi:10.1016/j.trpro.2019.07.027.

[12] Inc. Neo4j. The Neo4j Cypher Manual v3.5, 2019. URL: https://neo4j.com/docs/cypher-manual/current/.

[13] Apache Tinkerpop. Apache TinkerPop™, 2017.

[14] Mark Needham. Neo4j: Modelling hyper edges in a property graph. URL: https://www.markhneedham.com/blog/2013/10/22/neo4j-modelling-hyper-edges-in-a-property-graph/.