Topic:

Uniform Graph Query System based on openCypher

Master Thesis

towards fulfilment of the study programme:
M.Sc. International Software Systems Science,
Faculty of Information Systems and Applied Computer Science,
University of Bamberg,
Bamberg, Germany.

Author: Raj Joshi
Matriculation Number: 1943607
E-Mail: raj-dharmendra.joshi@stud.uni-bamberg.de
Date of Submission: 21.07.2021

Reviewer: Prof. Dr. Daniela Nicklas

Abstract

With more people becoming interested in graph database technologies, every developer, software architect, and stakeholder requires a concrete common system to work with. The requirement for such a system can be met by using a shared graph querying language such as Cypher. openCypher is an open-source project that was founded on the concept of vendor independence and platform agnosticism, and it serves as the foundation for this thesis. It currently provides a very natural way of working with property graphs and is regarded as the best on-ramp to the Graph Query Language (GQL) standard being developed by International Organization for Standardization (ISO). OpenCypher, which was originally contributed by Neo4j, is now used by more than ten products and many developers. This, however, does not completely solve the problem of using different data storage technologies to handle different data storage needs within a single software application. First, I proposed a system that addresses this issue with a "uniform" graph querying system based on some openCypher projects. The system plan will address all software components used to create this system. The common programming layer simplifies communication with the various datastores while also handling the various internal Cypher translations. Furthermore, a common internal format that can be easily read by other independent clients is produced, adding to the system's flexibility. Furthermore, both qualitative and quantitative evaluations are performed on this system. Eventually, the experimental results demonstrate the extent to which this system can be expanded or customized based on use-cases.

Contents

Abbreviations

GQL Graph Query Language

SQL Structured Query Language

ISO International Standard Organization

SDLC Software Development Life Cycle

oCIG openCypher Implementers Group

TCK Technology Compatibility Kit

EBNF Extended Backus–Naur form

PGQL Property Graph Query Language

ISO International Organization for Standardization

RDF Resource Description Framework

IRI Internationalized Resource Identifier

KMS Knowledge Management System

JSON JavaScript Object Notation

OOP Object-Oriented Programming

GUI Graphical User Interface

List of Figures

List of Listings

List of Tables

1 Introduction

Languages in humankind shape how you perceive the world. Words serve as classifications and labels in the process of comprehending reality and communicating with others. Technical languages, like their natural counterparts, aid in the realization, comprehension, and processing of data. Finding the best database solution for your application or development stack is a challenging task. This thesis presents a methodological approach with a system design that facilitates decision-making in a variety of scenarios. This work is incremental in nature, with an attempt to scale and add new features.

1.1 Motivation

A query language represents its model closely. On one hand, Structured Query Language (SQL) is all about tables and JOINs, on the other Cypher is about pattern matching relationships between entities. So, Cypher is very well suited to the challenges of querying connected data because it is a pictorial representation of circles and arrows that is understandable as well as an intuitive representation of data. Most original relational databases use a dialect of SQL as their query language, and while the graph database world has a few query languages to choose from, a growing number of vendors and technologies have adopted Cypher as their graph database query language [7].

The openCypher project [19] makes Cypher available to everyone – every data store, tooling provider, and application developer. It is an open-source project that pushes Cypher as a key inspiration for the ISO project of creating a standard graph query language. The motive of openCypher Implementers Group (oCIG) is to make a fully specified standard that can be independently implemented. The recently released Cypher 9 reference [5] along with accompanying formal grammar definitions (Extended Backus–Naur form (EBNF)

and ANTLR4) and conformance test suite Technology Compatibility Kit (TCK) – published under the Apache 2.0 license – already provide implementers with a solid basis for adopting Cypher [11]. At the time of writing, Cypher is supported by SAP HANA graph, Kantana Graph, Neo4j, Agens Graph, Redis Graph, and Memgraph, along with research frameworks. Ideally, we can build our application over one of the subsequent datastore but we might face a paradox of choice due to various factors like ease of scalability, reliability, cypher processing limitations, etc.

Following this work, new perspectives can be derived about possible multi-model and technology stack independent querying scenarios. Also, this will ease the application developer's struggle to cut time on backend configuration. They can then directly focus on the application needs on the front end side without worrying much about the backend. Querying using graphs gives an edge over traditional database systems by giving insightful information via relationships and pattern matching. This in turn contributes to the overall Software Development Life Cycle (SDLC).

1.2 Problem Statement

Since openCypher is still in its early stages and is still evolving, there is a lack of a common tool available to integrate all of the graph data stores in the back end in the context of applications using Polyglot Persistence. The phenomenon of polyglot persistence will be discussed in further chapters.

So the problem in hand can be stated as follows:

What components are required to access different graph data stores via a common interface in order to create a querying system uniform and independent of the client's application infrastructure?

This section explores the problem sub-classes involved in achieving the goal to unify graph querying. There are mainly two sub-classes that make up this composite problem.

1.2.1 Unified Querying

Before attempting to store and access data in various data stores we need to generalize the whole querying process. This becomes difficult with heterogeneous systems on the backend. To explain it more specifically, we need to make appropriate changes in the

Cypher query that a datastore accepts for processing requests. This adds a step for a client-side application to switch to different query writing methods depending upon the datastore to be used. Efforts are already made to achieve this by standardization of Cypher in general.

However, this does not mean there are no challenges involved with query translations with different data stores in the backend. The original schema of the dataset remains unchanged across the data stores but, there are still changes present with how each datastore store the data. If this is ignored we might end up with inconsistent results.

1.2.2 Generic Result Format

One of the key aspects of results is the format in which they are generated so the interpreter on the client-side can understand it. Most of the projects under openCypher use graph data structure as a result format. However, the heterogeneous nature of our proposed system will make the client dependent on using different parsers resulting in more work for client developers.

1.3 Challenges

This thesis proposes a solution that shows that using multiple data stores under the same application is possible using an abstraction layer that handles the stated problem.

There are several choices to make while designing such a system in context with an integrated or federated architecture to be taken into consideration, the design pattern of the interface, translations, and result format. Each of them comes with trade-offs and must be decided appropriately for a feature-rich system.

Finally, there are challenges related to performance benchmarking of the system which can be addressed with qualitative and quantitative evaluation. We can also determine the overhead added by the programming layer of the system vs the direct querying with the datastore.

1.4 Objective

The objective of this thesis for a student is to gain more in-depth knowledge of graph querying and designing a "state of art" system of uniform nature. To achieve that, a methodological approach is devised as follows :

- Understanding the property graph model.
- Learning about similar existing systems.
- Designing a robust software architecture with scalable components.
- Experimenting with real life dataset - cattle farming in our case.
- Evaluation of the system based on the quantitative and qualitative methodology.

1.5 Thesis Structure

My thesis is structured as follows:

- Chapter 2, provides some precursors about concepts used in conjunction with this thesis.
- Chapter 3 documents the software design of the proposed solution itself. It is further broken to understand various components of the system. This thesis' main artifact is an integrated approach, which is discussed in this chapter.
- Chapter 4 covers the technical aspects of the software implementation along with details on the technology stack being used. It sheds light upon reasoning for why certain implementation decisions were taken.
- Chapter 5 documents the results achieved by the thesis designed as already described in chapter 3 along with some implications and limitations of this work.
- In the end, Chapter 6 summarizes the key learning and makes suggestions for the next area of research or development.

2

Related Work

This chapter introduces some key concepts and approaches that can be considered a forerunner to the work. To begin, we will introduce the fundamentals of graph databases, followed by various types of graph data models, and then, a semantic concept of knowledge graphs, which has been one of foundations of the thesis. We then study graph query languages as a concept, discuss various projects that contribute, and grasp some of the common features provided. Understanding these concepts will help you digest the work itself.

We will conclude this chapter by showcasing some of the ideas that have been used in the past to implement similar systems, as well as their limitations. This thesis is inspired by the work presented in these systems.

2.1 Graph Databases

We live in a connected world! [17] There are no isolated pieces of information but rich connected domains all around us. In this section, we will go over the concept of graph databases, related use cases, and answer a few general questions about graph databases with appropriate reasoning.

Relationships are first-class citizens in graph databases, and most of the value of graph databases is derived from these relationships. Graph databases use nodes to store data entities, and edges to store relationships between entities. An edge always has a start node, end node, type, and direction, and an edge can describe parent-child relationships, actions, ownership, and the like. There is no limit to the number and kind of relationships a node can have [1]. Only a database that natively embraces relationships can store, process, and query connections efficiently. While other databases compute relationships at query time through expensive JOIN operations, a graph database stores connections alongside the

data in the model [17]. Independent of the total size of your dataset, graph databases excel at managing highly-connected data and complex queries. With only a pattern and a set of starting points, graph databases explore the neighboring data around those initial starting points collecting and aggregating information from millions of nodes and relationships, leaving any data outside the search perimeter untouched [17].

Learning about graph databases is not simpler than it appears; in the following section, we will look at a few use cases of systems that would benefit greatly from the graph database approach.

2.1.1 Use Cases

Graph databases really shine when working in areas where information about data interconnectivity or topology is important [16]. Many companies have developed in-house implementations in order to cope with the need of graph database systems. Examples would be Facebook's Open Graph, Google's Knowledge Graph, Twitter's FlockDB, and many more [16].

The following are some of the most well-known and widely used examples.

- **Social Graph:** A graph that depicts personal relations of internet users. It is a model or representation of a social network, where the word graph has been taken from graph theory to emphasize that rigorous mathematical analysis will be applied as opposed to the relational representation in a social network.

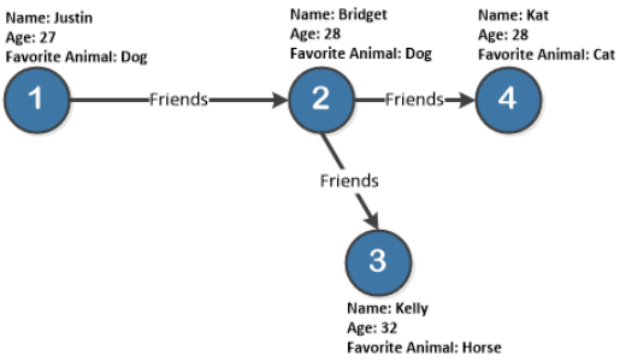


Figure 2.1: Friend-of-friend graph [16]

Fig. 2.1 represents a small social graph between friends and tracks the friendships, name, age, and favorite animal of each person. It offers an inexpensive solution to

finding the friends of Justin's friends. In traditional relational databases, it is quite challenging to deal with recursive data structures where each traversal along an edge in the graph would be a join. The friend of a friend problem is an example of a problem that would require far less work for a graph database when compared to RDBMS. Joins are expensive on an RDBMS while a traversal across an edge costs very little [16].

- **Recommendation Engines:** Graph databases are a good choice for recommendation applications. With graph databases, you can store in a graph relationships between information categories such as customer interests, friends, and purchase history. You can use a highly available graph database to make product recommendations to a user based on which products are purchased by others who follow the same sport and have similar purchase history. Or, you can identify people who have a friend in common but do not yet know each other, and then make a friendship recommendation [1].

  Such systems allow Google to present only the most relevant ads their user is most likely to enjoy. An example of this system would be Facebook's friend suggestion functionality or Amazon's item suggestion engine [16].

- **Bioinformatics:** Graph databases have seen extensive application in the field of bioinformatics. Bioinformatics use graph databases to relate complex web of information that includes genes, proteins and enzymes. A prime example is the Bio4j project. Bio4j is a framework powered by Neo4j that performs protein related querying and management. The project has aggregated most data available from other disparate systems (Uniprot KB, Gene Ontology, UniRef, RefSeq, ext.) into one golden source. The project is open source and the intention is for it to be easy to expand upon. This is made easier because the data is stored so that it semantically represents its own structure [16].

Due to the widespread use of graph databases in every domain, questions may arise after covering several use cases. In the following sections, we will answer the most frequently asked questions about graph databases.

2.1.2 What differentiates a graph database from a relational database?

The main difference is the way relationships between entities are stored. In a graph database, relationships are stored at the individual record level, while a relational database uses predefined structures, also known as table definitions.

Relational databases are faster when handling huge numbers of records because the structure of the data is known ahead of time. This also leads to a smaller memory footprint. Graph databases do not have a predefined structure for the data which is why each record has to be examined individually during a query to determine the structure of the data. For example, if you want to add different properties to some of the nodes, you would be able to. Unlike a table, where you need to add a column for each additional attribute, here you can be much more flexible with the data structure and types. A property that was meant to be a string can be used as an integer without any constraints. To be fair, this can cause problems for you in the long run, but you can do it if need be.

2.1.3 When to use a Graph Database?

There are always two sides to every story and graph databases are not a perfect solution for every problem. Far from it. There are a lot of use cases for which you should stick with relational databases or maybe search for other alternatives aside from graph databases.

Here are three simple questions you can ask yourself to determine whether or not you should use a graph database.

1. **Is data highly-connected?**

   Graph solutions are focused on highly-connected data that comes with an intrinsic need for relationship analysis. If the connections within the data are not the primary focus and the data is of a transactional nature, then a graph database is probably not the best fit. Sometimes it is just important to store the data and complex analysis is not needed

2. **Is retrieving the data more important than storing it?**

   Graph databases are optimized for data retrieval and if you choose one, then you should probably use this functionality often. If your focus is on writing to the database and you are not concerned with analyzing the data, then a graph database

would not be an appropriate solution. A good rule of thumb is, if you do not intend to use JOIN operations in your queries, then a graph is not a must-have.

3. **Does the data model change often?**

   If the data model is inconsistent and demands frequent changes, then using a graph database might be the way to go. Because graph databases are more about the data itself than the schema structure, they allow a degree of flexibility.

   On the other hand, there are often benefits in having a predefined and consistent table that is easy to understand. Developers are comfortable and used to relational databases and that fact cannot be downplayed.

   For example, if you are storing personal information such as names, dates of birth, locations... and do not expect many new fields or a change in data types, relational databases are the go-to solution. On the other hand, a graph database could be useful if - additional attributes could be added at some point, not all entities will have all the attributes in the table and the attribute types are not strictly defined.

We now understand why a graph database is more efficient in certain situations. In the next section, we will go over some scenarios in which a graph database is not a optimal solution.

2.1.4 When not to use a Graph Database?

The following are some circumstances in which a graph database may fall short of expectations.

1. **When no specific starting points are specified in the query.**

   If you need to run frequent table scans and searches for data that fits defined categories, a graph database would not be very helpful. Graph databases are well equipped to traverse relationships when you have a specific starting point or at least a set of points to start with (nodes with the same label). They are not suited for traversing the whole graph often. While it is possible to run such queries, other storage solutions may be more optimized for such bulk scans.

2. **When there is a need of Key-Value storage.**

Very often, databases are used to lookup information stored in key-value pairs. When you have a known key and need to retrieve the data associated with it, a graph database is not particularly useful.For example, if the sole purpose of the database is storing a user's personal information and retrieving it by name or ID, then refrain from using a graph. But if there were other entities involved (visited locations for example), and a large number of connections is required to map them to users, then a graph database could bring performance benefits. A good rule of thumb is, if most of your queries return a single node via a simple identifier (key), then just skip graph databases.

3. **When there is a need to store large chunks of information.**

   If the entities in a model have very large attributes like BLOBs, CLOBs, long texts... then graph databases are not the best solution. While you can store those objects as nodes and link them to other nodes to utilize the power of traversing relationships, sometimes it just makes more sense to store them directly with the entities they are connected to.

Upon gaining knowledge about graph databases, the first basic step is to select a data model. The term graph data model will be explained in the upcoming section.

2.2 Graph Data Models

The graph data model is often referred to as being "whiteboard-friendly". Typically, when designing a data model, people draw example data on the whiteboard and connect it to other data drawn to show how different items connect [10]. Many data models are used to represent graphs; some of the relevant models are discussed in the following sub-sections.

2.2.1 The Property Graph Model

In this model, data is organized as nodes, relationships, and properties (data stored on the nodes or relationships).

Nodes are the entities in the graph. They can hold any number of attributes (key-value pairs) called properties. Nodes can be tagged with labels, representing their different

roles in your domain. Node labels may also serve to attach metadata (such as index or constraint information) to certain nodes [24].

Relationships provide directed, named, semantically-relevant connections between two node entities (e.g. Employee WORKS_FOR Company). A relationship always has a direction, a type, a start node, and an end node. Like nodes, relationships can also have properties. In most cases, relationships have quantitative properties, such as weights, costs, distances, ratings, time intervals, or strengths. Due to the efficient way relationships are stored, two nodes can share any number or type of relationships, same label/type that might differ in the attributes without sacrificing performance. Although they are stored in a specific direction, relationships can always be navigated efficiently in either direction [24].

Properties allow you to store relevant data about the node or relationship with the entity it describes. They can often be found by knowing what kinds of questions your use case needs to ask of your data.

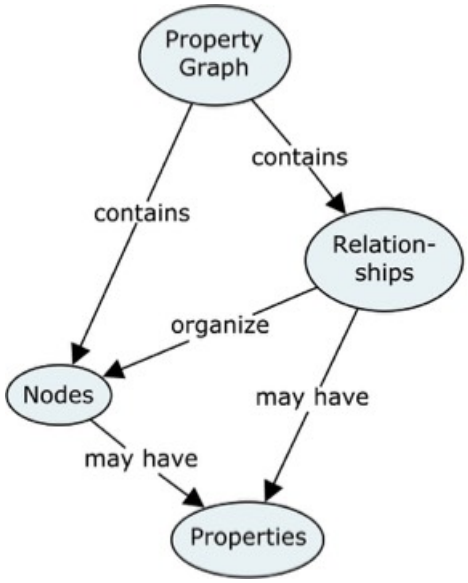


Figure 2.2: Concept map of a property graph [8].

The concept map represented in Fig. 2.1 explains the important constraints further exploring the above formal definitions.

Both Nodes and Relationships can (should) have “names” (formally called labels for nodes and types for relationships), just like concepts and their relationships have in the diagram above. Relationships are directed, which is visualized by the arrowheads. Both nodes and

relationships may be associated with properties, which are “key / value” pairs such as e.g. Color: Red. On the data model level, we call the key "Property Name" [8].

We chose to use property graph model for implementation in this thesis because it was widely supported by all openCypher projects. However, RDF is another data model that has a strong following in comparison to the property graph model, which will be discussed in the following section.

2.2.2 RDF Data Model

Resource Description Framework (RDF) is a framework which represents information on web. The RDF data model and its query language SPARQL have been standardized by the World Wide Web Consortium.

An RDF triple, the basic unit of representation in this model, consists of a subject, a predicate, and an object. A set of such triples is called an RDF graph. We can visualize an RDF triple as a node and a directed edge diagram in which each triple is represented as a node-edge-node graph as shown in Fig. 2.2 [27].

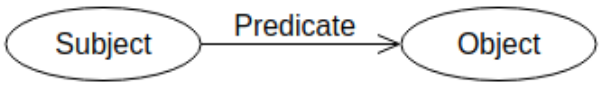


Figure 2.3: RDF representation [27].

There can be three kinds of nodes: An ***Internationalized Resource Identifier (IRI)*** is uniquely used to identify resources on the web. A ***Literal*** is a value of a certain data type, for example string, float, integer, etc. A ***Blank Node*** is also a node that does not have an identifier, and is similar to an anonymous or an existential variable.

It is an abstract model with several serialization formats (being essentially specialized file formats), in addition, the particular encoding for resources or triples can vary from format to format. They are atemporal in the sense that they provide a static snapshot of data. An RDF consists of a vocabulary that is a collection of IRIs intended for use in RDF graphs. With suitable vocabulary extension, they can express information about events, or other dynamic properties of entities.

The main purpose of RDF is to provide an encoding and interpretation mechanism so that resources can be described in a way that particular software can understand it.

Some of the key differences from the property graph and RDF are listed as follows [2].

- **RDF does not uniquely identify instances of relationships of the same type.**
- **Inability to distinguish between different types of partnerships.**
- **The labeled property graph can have arrays, and RDF can have multi-valued properties.**
- **Quads are used in RDF to define named graphs.**

Moving on to the next step, graph querying systems require the use of graph query languages to complete the required processing. The following section discusses the current state of the art in graph query languages.

2.3 Graph Query Languages

In the context of graph data management, a GQL defines the way to retrieve or extract data that have been modeled as a graph and whose structure is defined by a graph data model [28]. The main inspirations have been Cypher (now with over ten implementations, including six commercial products), Oracle's Property Graph Query Language (PGQL) and SQL itself, as well as new extensions for read-only property graph querying to SQL. Tigergraph's GSQL, although coming from a stylistically different starting point than the other inputs, is another noteworthy contribution, whose authors have shown a strong commitment to the GQL project [23].

It is well over thirty years since the SQL project started, initially as an ANSI standard: GQL is the first ISO international standard database languages project after SQL. Ten countries voted for the new project: China, Korea, Sweden, the U.S., Germany, the U.K., the Netherlands, Denmark, Kazakhstan, Canada and Finland. Five abstained on grounds of lack of expertise to judge or comment on the proposal. Fig. 2.4 shows the countries who have not only voted "Yes", but also pledged expert participation [23].

SQL is a language very different from Cypher in one critical respect. Cypher and PGQ allow a user to explore the structure of the data graph without knowing up-front which types of data are going to be returned. They let you do an intuitive graph querying, where the interesting facts are just the values, but the shape of a subset of the data, defined

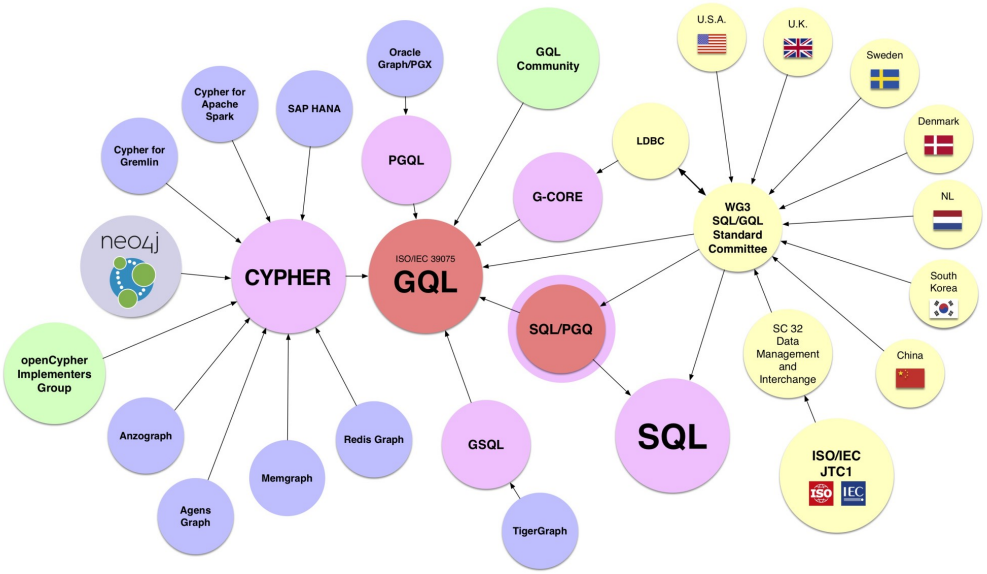


Figure 2.4: A broad overview of querying languages and their contributors [23].

concerning the values of elements that match a graph pattern. In other terms, graph queries describe sub-graphs or projected graphs computed over one or more input graphs.

Graph query languages come with very unique features, some of them are stated below:

- **Pattern Matching:** The idea of patterns is to define small sub-graphs of interest to look for in an input graph [28].
- **Extensions:** Graph patterns can be extended by adding numerous features. For starters, one could allow variables in the edge labels of patterns or allow a mixture between constant and variables [28].
- **Path Queries:** The idea of a path query is to select pairs of nodes in a graph whose relationship is given by a path (of arbitrary length) satisfying certain properties. For example, one could use a path query in a social network to extract all nodes that can be reached from an individual via friend-of-a-friend paths of arbitrary length [28].
- **Adding Path Queries to Patterns:** Path queries and patterns can be naturally combined to form what is known as *conjunctive regular path queries* or CRPQs. Just as with graph patterns and conjunctive queries, we can define CRPQs in two ways. From a pattern perspective, CRPQs amount to replacing some of the edges in a graph pattern

by a regular expression. However, this feature is beyond the scope of this Thesis [28].

- **Omnigraphs:** Here, you can take the union of two graphs and then add new elements (nodes and relationships) to create a third graph. This approach is a copy of the first two, plus any new data. As databases are mutable stores, in Neo4j it has started calling the mutable equivalent of this "two plus new" design pattern an omnigraph.

GQL will expand on the work done in openCypher Morpheus, which brings Cypher to Apache Spark and provides users with a composable graph query language, enabling all of those features. This will elevate GQL to the conceptual level of SQL... and then some.

To put our newly acquired knowledge of graph databases, models and languages to use, we must first understand how they are used in a software system. The motivation for learning about knowledge graphs stems from an Knowledge Management System (KMS) used in the SIMUTOOL EU Project [21], which serves as a common location where data and information from various organizations/activities/systems is constantly added to and retrieved. This thesis proposes a uniform querying system in order to facilitate knowledge transfer.

2.4 Knowledge Graphs

A Knowledge Graph is a model of a knowledge domain generated by subject-matter experts with the help of intelligent machine learning algorithms. It yields a structure and common interface for all of your data and enables the creation of smart multilateral relations throughout your databases. Structured as an additional virtual data layer, they lie on top of your existing databases or data sets to link all your data together at scale – be it structured or unstructured.

However, learning about this semantic concept can be complex. Taking inspiration from many other research work, the team of authors Aidan Hogan et al of the research paper - Knowledge Graphs [12] distinguish two types of knowledge graphs in practice: *open knowledge graphs* and *enterprise knowledge graphs*. Open knowledge graphs are published online, making their content accessible for the public benefit. The most prominent examples – DBpedia, Freebase, Wikidata, etc. cover many domains and are either extracted from Wikipedia, or built by community members. Open knowledge graphs have also been published within specific domains, such as media, government, geography, tourism,

life sciences, and more besides. Enterprise knowledge graphs are typically internal to a company and applied for commercial use-cases. Prominent industries using enterprise knowledge graphs include Web search (e.g., Bing, Google), commerce (e.g., Airbnb, Amazon, eBay, Uber), social networks (e.g., Facebook, LinkedIn), finance (e.g., Accenture, Banca d'Italia, Bloomberg). Applications include search, recommendations, personal agents, advertising, business analytics, risk assessment, automation, and more.

We learned from our brief information that a knowledge graph can be implemented into a system using database as a host. So, how about we apply the same idea to a graph querying system?

The following are some key learnings from such a system:

- Knowledge graphs is a collection of data that has to be stored, managed, extended, quality-assured and can be queried which requires databases and components on top, they are usually implemented in the Semantic Middleware Layer. This 'sits' on the database and at the same time offers service endpoints for integration with third-party systems. Thus graph databases form the foundation of every knowledge graph.

- To roll out knowledge graphs in organizations, however, more than a database is required: Only with the help of components such as entity extractors, graph mappers, validation, visualization and search tools, etc. can it be ensured that a knowledge graph can be sustainably developed and managed.

- While graph databases are typically maintained by highly qualified data engineers or Semantic Web experts, the interfaces of the Semantic Middleware also allow people to interact with the knowledge graph who can contribute less technical knowledge instead of business and expert knowledge to the graphs.

The concept of a knowledge graph opens up a plethora of possibilities for ideas to be built upon. We will now define a term that is heavily related to the context of this thesis and is mentioned in section 1.2 titled Problem Statement.

2.5 Polyglot Persistence

Initially, monoglot was (and still is) fine for a simple application that uses one type of workload. However, applications quickly become more complex. Polyglot persistence on the other hand uses different database systems within a single application domain, addressing different functional and non-functional needs with each system. It allows you to fuel applications with data from different storage types. For example: You could pull completed financial transactions from Oracle for your reports while tracking user information in MongoDB and letting Redis handle cache [13].

To implement such a system based on the concept, we must consider the following architectural constraints:

- Determine a business function, potential issues to be eliminated and right data solutions.
- Data storage types: transactional (SQL based) or non-transactional databases (Key-Value, Document, Column, Graph).
- Data structure questions: *Does it have a structure or is it unstructured? How is it distributed? How much data are you dealing with?*
- Access patterns: *Is it uniform or random? Is reading or writing more important to your application?*
- Organizational needs like authentication, encryption, backups, monitoring tools, languages to be used, driver requirements etc.

Moreover, it should be avoided to push the burden of all of these query handling and database synchronization tasks to the application level – that is, at the end to the programmers that maintain the data processing applications. Instead, it is usually better to introduce an integration layer. The integration layer then takes care of processing the queries – decomposing queries into several subqueries, redirecting queries to the appropriate databases, and recombining the results obtained from the accessed databases; ideally, the integration layer should offer several access methods and should be able to parse all the different query languages of the underlying database systems as well as potentially translate queries into other query languages [**wiese_polyglot_nodate**].

A similar system design is proposed in chapter 4.

2.6 Related Systems

Similar attempts to create such a system have been made with different ideas such as creating a uniform query framework with heterogeneous data, with modern federated architecture that transparently maps multiple autonomous database systems into a single federated database, and with performance evaluation of multi-model data stores in polyglot persistence applications.

In subsequent sections, we discuss some of these retrospectives.

2.6.1 Uniform Query Framework for Heterogeneous Data Management Systems

Most organizations today store their data in a variety of data storage systems such as relational databases, columnar stores, document stores, text search engines, etc. The adoption of multiple data storage systems tailored towards specific needs has its challenges as described by authors Jayant B. Karanjekar and Manoj B. Chandak in their work [14]. They propose a SQL-like query mechanism for a combination of relational and NoSQL databases such as MySQL, MongoDB, Cassandra, and CSV files. Furthermore, they dive deeper into the architecture and optimization details but also analyze the response time of the framework and compare it with that of individual systems.

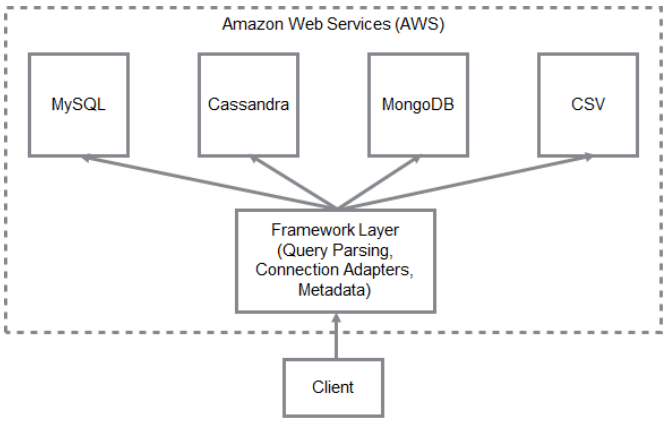


Figure 2.5: System architecture of the framework [14].

Fig. 3.1 shows the architecture of the framework consisting of the central component called the framework layer. It in turn talks to various databases that are part of the system. Also, the architecture offers flexibility to add or remove nodes for any particular database without impacting other database systems.

Individual database systems are meant for storing data whereas the framework layer is responsible for maintaining metadata, handling client requests, query processing, and returning the response to the client [14].

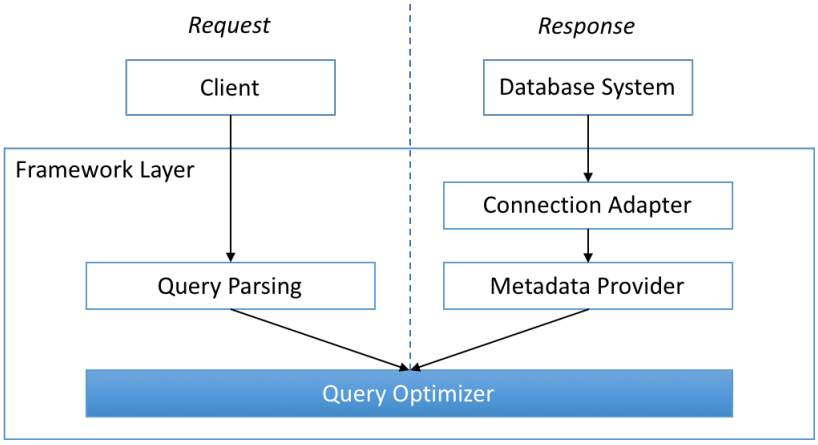


Figure 2.6: Framework Layer Interactions [14].

Fig. 3.2 outlines the main components of the framework layer. Once the request is received by the framework layer, it goes through the query parser that translates the SQL query into a tree of relational operators. Query optimizer applies query transformation rules on the relational operators to come up with an optimized execution plan [14].

We learn the concept of having a consistent query framework for multiple data sources and the ability to perform query processing as required by the client from this work. Such an approach motivates us to develop a similar uniform graph querying system based on the creation of a Programming Layer, which is discussed in detail in Chapter 4. We, on the other hand, implement using a graph data model.

2.6.2 Federated Graph Querying

A federated database system makes several databases appear to function as a single entity. Each component database in the system is completely self-sustained and functional. When an application queries the federated database, the system figures out which of its component databases contain the data being requested and passes the request to it. Federated databases can be thought of as database virtualization in much the same way that storage virtualization makes several drives appear as one [20].

When datasets reside in different data centers and cannot be collocated due to technical,

administrative, or policy barriers, a unique set of problems arise that hamper querying and data fusion. To address these problems, author Michael D. Lieberman et al of "Parasol: An Architecture for Cross-Cloud Federated Graph Querying" [15] proposed a system and architecture named Parasol that enables federated queries over graph databases residing in multiple clouds. They also conduct experiments on a prototype implementation of Parasol which indicated suitability for cross-cloud federated graph queries.

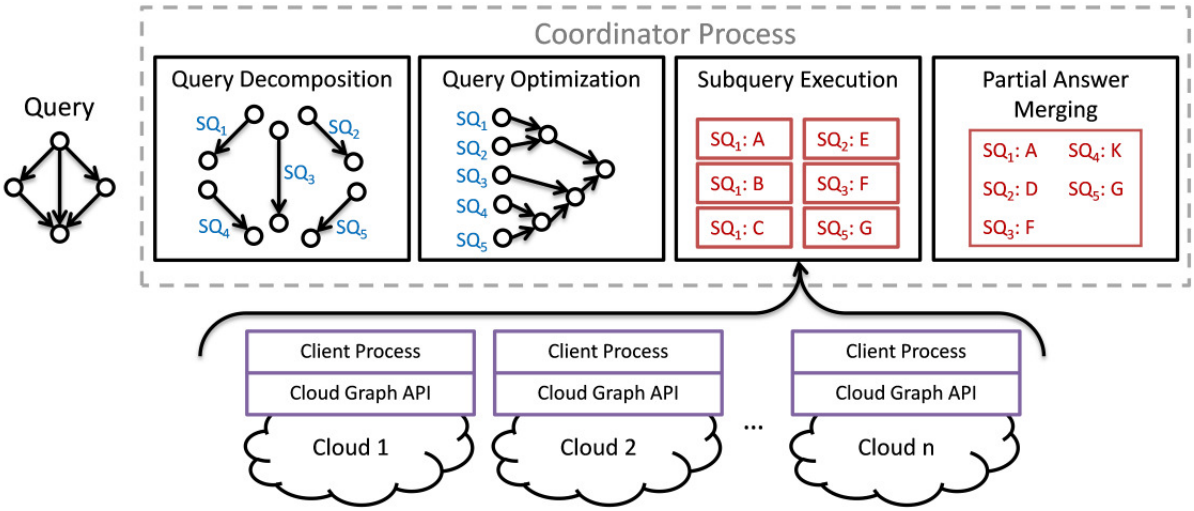


Figure 2.7: Parasol's architecture [15]

Fig. 3.3 illustrates Parasol's architecture. The architecture is defined by two lightweight processes: a coordinator process running in a coordinator cloud, and multiple client processes running in client clouds. Users interact with the coordinator by sending it graph queries. The coordinator then decomposes each query into appropriate subqueries, performs query optimization to devise a query plan, and delegates subquery execution to client processes as appropriate [15]. Then, clients execute the subqueries, returning their partial results to the coordinator, which then merges them to produce final answers that are returned to the originating user.

The important point here is that client processes assume no particular infrastructure for the client clouds. Instead, they only provide the logic to communicate with the coordinator, and a minimal API that the client clouds are responsible for implementing, which is a fairly standard practice in data integration [15]. The API includes simple methods for computing answers to individual subqueries, which is the minimal functionality for a working federated graph query system.

Similarly, we have taken this API-centric approach and encapsulated query translations

along with generic output formatting in our system. This brings more abstraction towards processing.

2.6.3 Performance Evaluation of Graph Data Stores

As we move forward towards evaluating the system, a question arises of how well these data stores perform. One such methodological approach was presented by authors Fábio Roberto Oliveira et al in "Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications" [18]. Their work focuses on the performance evaluation of multi-model data stores used by an application with polyglot persistence. In the same process, they evaluate the performance of benchmarks based on a set of basic database operations on a single model and multi-model data store systems. However, the lesson here for us is the methods used in the work.

Two synthetic datasets with different sizes were generated. The first is a SMALL dataset with 100,000 documents and nodes, each one with 10 fields and 547,993 edges. The second is a MEDIUM dataset, with 500,000 documents and nodes, each one with 50 fields and 6,500,237 edges [18].

Benchmarks relevant to this work were:

- Graph load: This benchmark inserts the graph data in the data stores. Each insertion adds a node or an edge of the graph. It evaluates the load performance of the multi-model data stores used as a single-graph database system.
- Document query with graph traversal: This benchmark executes 10,000 queries of documents using the graph adjacency relationship between documents. It evaluates the performance of multi-model data stores in querying documents with traversal of several depth levels in the graph.

Based on this idea, we conduct an performance based experiment with scenarios on query processing. These parameters will be thoroughly discussed in Chapter 5

3 System Design

This chapter describes the software architecture of the querying system used in this thesis. To begin, we will explain system architecture, followed by the software framework used for query processing, and a few design decisions made throughout the process.

3.1 System Architecture

Fig. 3.1 illustrates the recommended system design for the uniform graph querying component of this architecture.

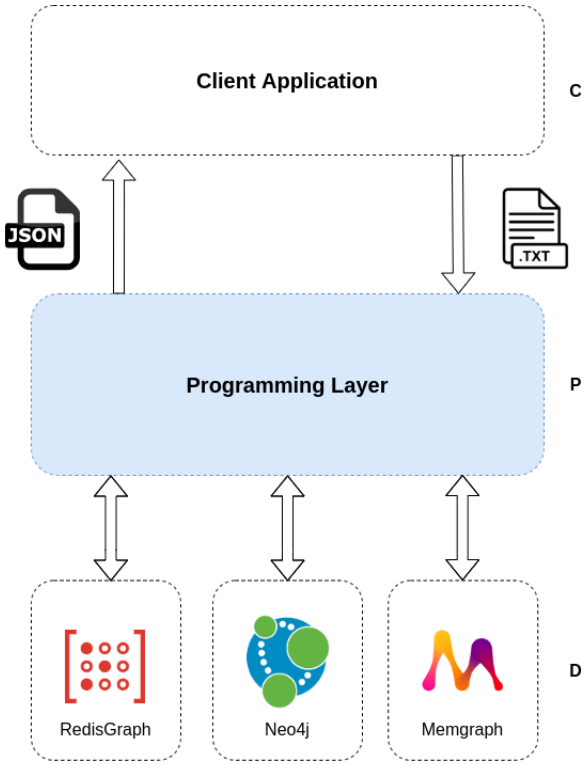


Figure 3.1: End to end system architecture

Client 'C' sends a plain text file to Programming Layer 'P'. The programming layer interprets the datastore selection and consume queries from the same file. It also handles query execution with the help of underlying data stores labeled 'D.' The layer can also return results in JavaScript Object Notation (JSON), which is a generic output format.

The system as a whole can be split into four logical steps. These steps are detailed in the following subsections.

3.1.1 Pure Cypher

The proposed graph querying system requires *plain or raw Cypher* in order to parse through the Programming Layer. It is recommended to follow the complete reference of the Cypher query language. At the time of writing version 9 [5] is the latest. The plain text file's input format is: first line of the file should contain a numeric value in the range 1-3, with each number representing a datastore. 1 is an example for Neo4j followed by Cypher queries in a single line. The layer will then interpret *plain Cypher* from that text file which is explained in detail in Section 4.2 titled Software Architecture for Programming Layer. However, there is an alternative; for testing purposes, the command-line prompt, which is not recommended for mass query execution, can be used. This aspect of the design is embedded within the implementation and will be covered in detail in Chapter 4.

The goal here is to use as much pure Cypher as possible, so we went with a plain text file for the time being. The client application module must expose only the necessary configuration to write plain text, allowing it to be independent of lower-level components.

3.1.2 Interpretation - Choice of Data Store and Query Consumption

Once a request is received at the Programming Layer, underlying program will interpret the choice by reading first line of the text file. A connection to the data store is established accordingly. Thereafter, queries are consumed with the help of inbuilt parsers of respective database. All the query consumption is handled at the Programming Layer with the help of CypherPersistence API as shown in 3.1.

This model allows you to send multiple queries at one time and consume the results in order, without having to wait the whole latency cycle.

3.1.3 Query Processing and Execution

The API-centric approach at the Programming Layer helps the queries to run in isolation mode. In other words, queries are executed in a single persistence mode only on their respective data store of choice. During the execution, a query is checked for any possible translation during the process. Underlying libraries for data stores provide a feature set for valid connection using an appropriate driver-adapter combination, transaction modes, inspection, and profiling. Thus contributing to logical query execution.

3.1.4 Generic Output

Each query execution returns a result. A result consists of set of records and result metadata. They are pulled out continuously from the stream of records from the database and represented as graph data structure. However, every database has its own way of representing graph data structure. For example : Neo4j's Record class is an immutable ordered collection of key-value pairs along with metadata while on the other hand Memgraph returns three main types of objects namely Node, Relationship and Path.

A custom serializer is used to generate a generic JSON-like format. The generated JSON format closely resembles RFC-8259 format. This serializer eliminates the time-consuming task of type conversion, removes unnecessary metadata, and only returns result-related properties. Using a generic format like this makes the output more human-readable and less error-prone for the client. It also resolves the issue raised in section 1.2.2 titled Generic Result Format.

3.2 Software Framework for Query Processing

Fig. 3.2 illustrates the framework and databases that the implementation utilises.

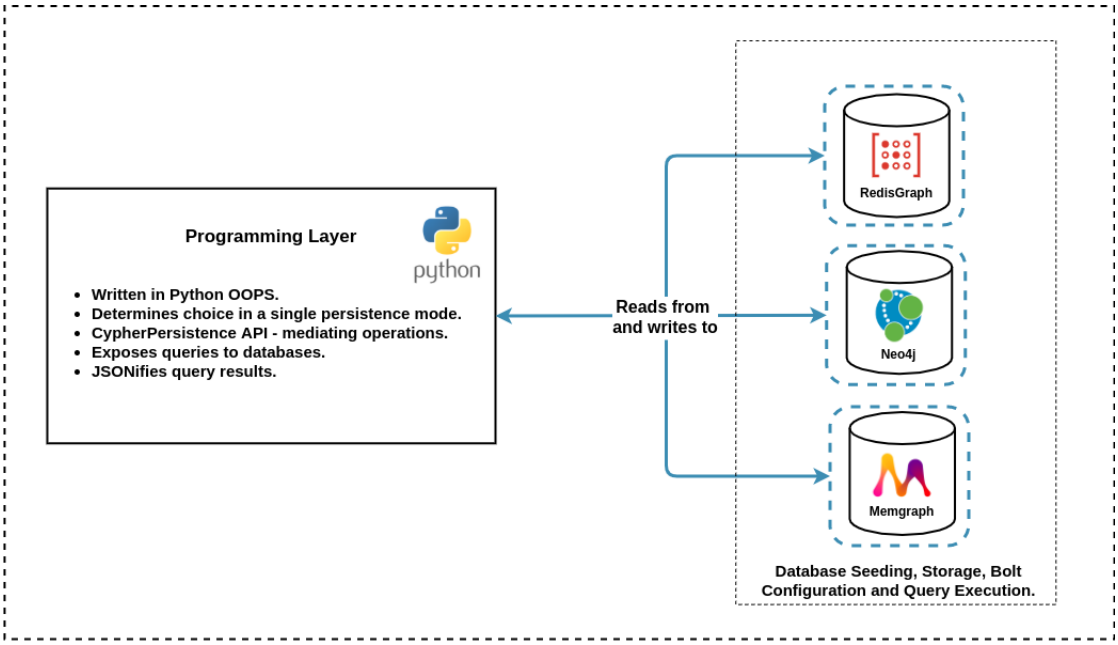


Figure 3.2: Higher level software architecture

The programming layer is composed of a software package written in Python. There are three databases namely - Memgraph, Neo4j, and RedisGraph (from right to left respectively) configured to run in separate containers using Docker as represented. Both of these components are loosely coupled with synchronous reading and writing operations possible between them.

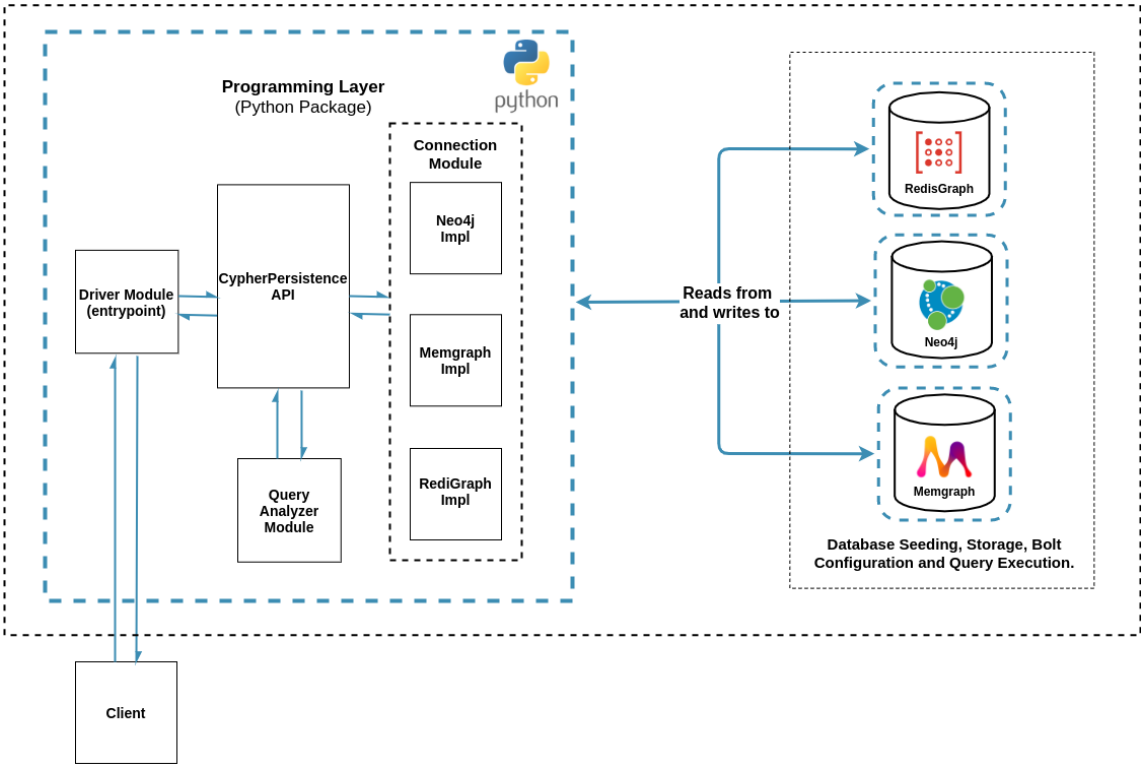


Figure 3.3: Higher level software architecture with programming layer magnified

Fig. 3.3 has a magnified view of the logical components of the software package that utilizes Python Object-Oriented Programming (OOP) principles. Here is a summary of each module.

- **Driver Module:** Serves as an entry point to the programming layer with responsibilities such as performing plain text checks, communicating datastore choice to CypherPersistence API, logging the execution times, and success or failure logs. Additionally, it also handles JSON file write operations for the client.
- **CypherPersistence API:** Exposes the endpoints for database connection and query execution to the connection module. Moreover, it feeds the query for possible translations to the Query Analyzer module. Once, everything is checked upon, endpoints complete the execution cycle in their responsible implementation. Both operations are mediated by the API.
- **Query Analyzer Module:** Includes a regular expression-based check for numerous types of queries that match predefined query types. After that, any possible translations are applied and returned to the CypherPersistence API.

- **Connection Module:** In this module, the actual core lies. Depending on the endpoint given by CypherPersistence API, execution is transferred to the specific sub-module of the datastore. Each implementation sub-module consists of a set of functions that carries out query execution. Lastly, a serialized output is returned to the mediator which then works with the Driver module to JSONify the output.

3.3 Design Decisions

In this section, we go over some of the critical design decisions that were made for the software implementation and explain why they were made.

3.3.1 Why should you use the Bolt protocol instead of HTTP?

Fig. 3.4 shows a quick overview of what the HTTP and Bolt protocols look like side by side in a typical set-up:

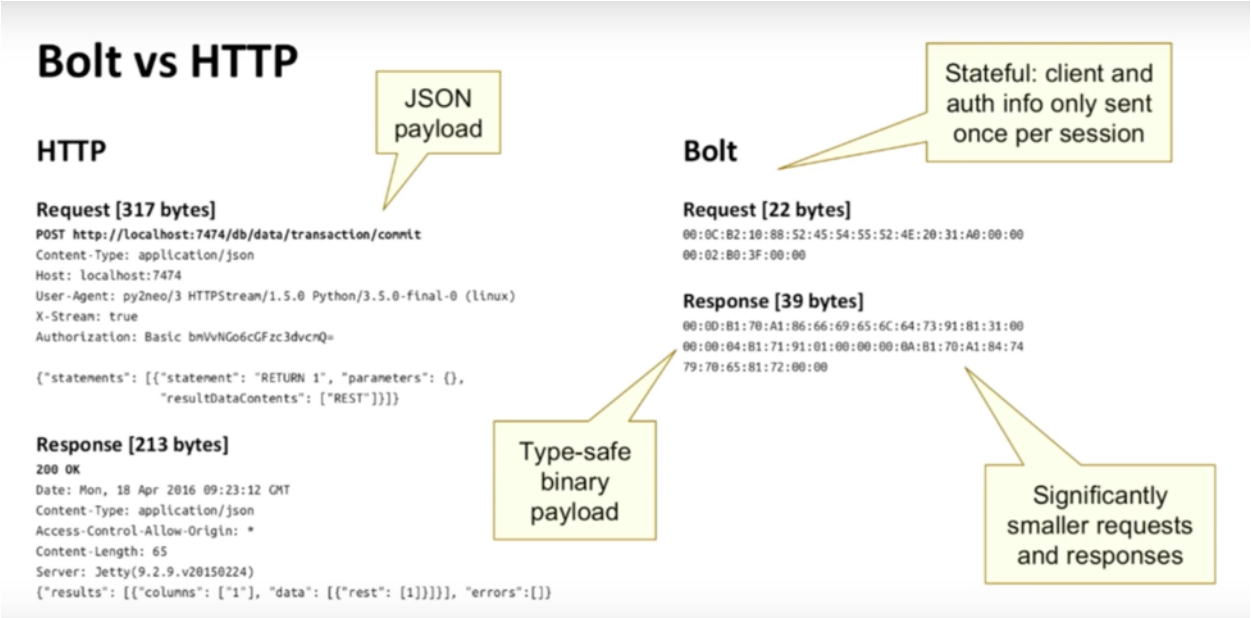


Figure 3.4: Bolt vs HTTP [22]

Request and response will typically contain a load of HTTP headers and adjacent payload. If we contrast that with the binary that is sent to and from with Bolt, we obtain some significantly smaller requests and responses. So naturally, it gets a bit of better performance, overall.

The payload for Bolt itself uses a data sterilization mechanism that Neo4j built in-house and it is very closely tied to the Neo4j type system, which was then worked to standardize. Additionally, as we are using a session in Bolt world, we have a stateful set-up rather than the stateless set-up of HTTP, so it does not need to send all the information for every request resulting in a secure tunnel-like connection.

To get the maximum out of our system, we need to implement specialized algorithms or talk to a third-party system. The best way to do that is to use an extension that directly plugs into the database and executes directly on the bare metal. It was obvious to the development team of the Bolt protocol and drivers project that they would need to preserve that capability. Hence, all the available official drivers use this phenomenon in case of Neo4j.

Memgraph on the other hand borrows the same approach from Neo4j for programmatically querying the database using Bolt. Additionally, it supports versions 1 and 4 of the protocol that serves our purpose.

3.3.2 Why would you use a JSON-like format as a generic output?

JSON is a lightweight data-interchange format. One can describe a use for an API, which would be an ideal situation to use JSON output over something like XML. In our case, we communicate with external clients that are usually web-based applications or of a similar kind. The use of JSON-like format allows us to send only the entities needed by the client. Moreover, it has in-built parsing capability from many programming languages like Python, JavaScript, PHP, etc.

```
{
  "type": "node",
  "id": "1",
  "labels": ["TBox", "property"],
  "properties": {
    "unique": "False",
    "namespace": "dcterms",
    "description": "Date of creation of the resource.",
    "title": "created",
    "uri": "http://example.org/tbox/created",
    "xsd_type": "xsd:dateTimeStamp"
  }
}
```

Figure 3.5: Sample JSON snippet from a dataset

Fig. 3.5 represents a sample JSON snippet with RFC 8259 specification [3] extracted from the dataset being used in this Thesis. We can learn a lot of information with minimal representation about the entity.

In our software system, a problem arises while working with results. Cypher returns quite a few different things (a table of discrete values, nodes, relationships, paths, a subgraph,

...) and when accessing the database through Bolt driver it is the developer that has to determine what to do with the result. So we are given a result-set (which from the point of view of the database could contain anything) and have to do the post-processing by ourselves (since only the developer knows what he/she asked for). If a developer wants a JSON-like result, one could query the database through the HTTP endpoint. However, as we discussed in the above section 3.3.1, it requires heavy processing from our end. As a result, we are obligated to write a custom serializer, which will be described in the following Chapter 4.

4

Implementation

The purpose of this chapter is to explain the reasoning behind various implementation decisions made during the implementation of the designed architecture. First, we will look at the OOP paradigm that was used while developing the software, followed by a utility based on command-line interface for initial testing, as mentioned previously in Section 3.1.1, and Cypher transformation that go through the transformation steps required to reach database layer. In conclusion, we describe a custom serializer, which is the essential programming challenge in order to achieve generic result. The source code featured in this chapter is submitted on CD with this thesis.

Note: RedisGraph's implementation is only partially complete due to time constraints.

4.1 Programming Paradigm

Previously, in section 3.2, we realized about the software framework of the programming layer. This section aims to explain the OOP paradigm used to make components reusable, resulting in a system that is easy to scale, dynamic, and flexible.

The fine details of the software package components used for implementation are depicted in Fig. 4.1

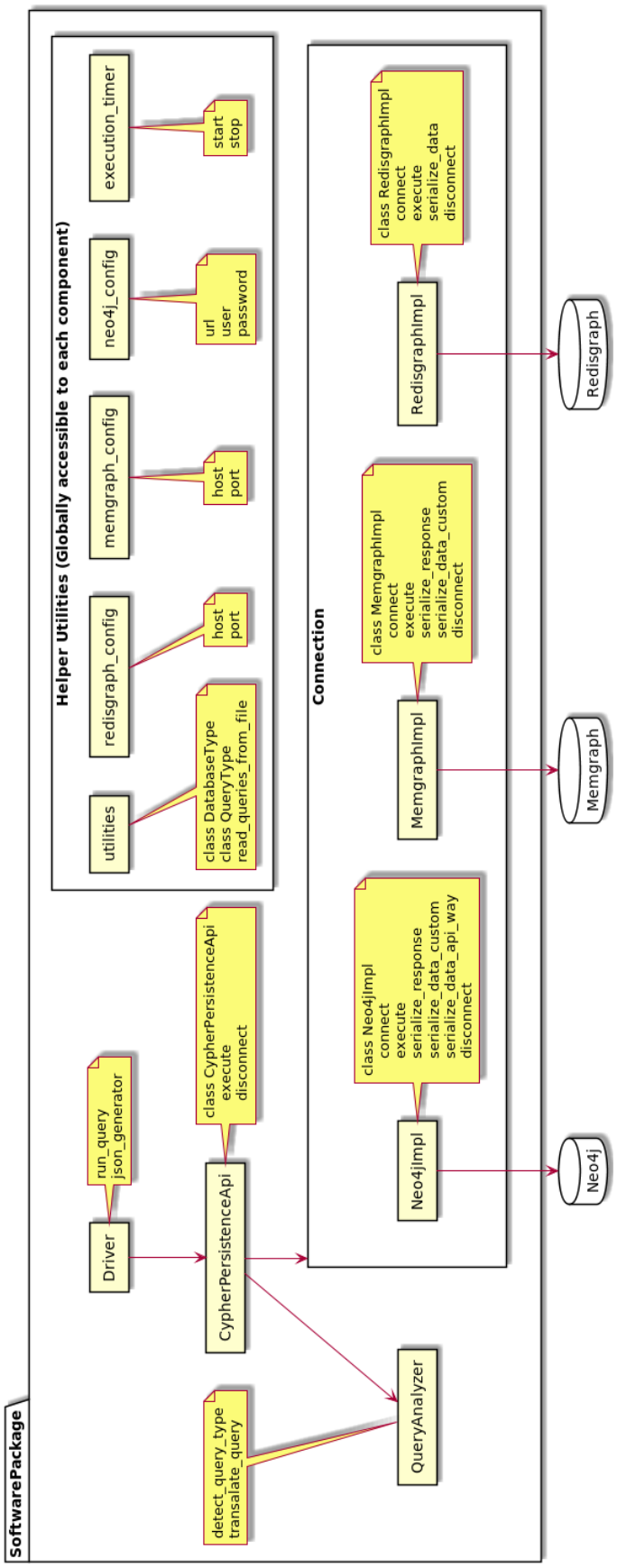


Figure 4.1: Package Diagram

Classes provide a means of bundling data and functionality together. In our system, each database implementation has its own set of classes and functions. For example: *class Neo4jImpl* has a set of functions to communicate with database and return a serialized response using *serialize_response()*. Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. The implementation classes for each database take *Connection class* as an *abstract class*.

An abstract class can be considered as a blueprint for other classes. It allows you to create a set of methods that must be created within any child classes built from the abstract class. A class which contains one or more abstract methods is called an abstract class. An abstract method is a method that has a declaration but does not have an implementation. While we are designing large functional units we use an abstract class. When we want to provide a common interface for different implementations of a component, we use an abstract class. Similarly, the *Connection class* in our package consists of execute method that establishes database connection based on the respective choice obtained from the Driver module. You can define a common Application Program Interface (API) for a set of sub-classes by defining an abstract base class. This capability is especially useful when a third-party will provide implementations, such as with plugins, but it can also help you when working in a large team or with a large code-base where remembering all classes is infeasible.

The *CypherPersistenceApi* class, which mediates choice and query analysis, includes execution and disconnection functionality through the use of a *database_type* object for each datastore. It also checks with the QueryAnalyzer for query type and query translations defined in the regexMap. This section of the program, however, is only partially implemented. Because Neo4j and Memgraph use pure cypher, major query translations are handled at the library level. In the case of Redisgraph, we make a minor change inside the constructor to specify the graph key to be used for querying.

The helper utilities component is a collection of configuration files, common utilities for query processing such as finding database and query type. Because each datastore has its own caveats for describing configuration, creating a separate configuration makes more sense. We can also add, remove, or modify openCypher-based datastores on the fly. This approach makes a component reusable and allows for some scalability.

4.2 Command-Line Utility for Testing

In the previous chapter, section 3.1.1, we mentioned an aspect of the design that is embedded in the system. Client applications that want to integrate with the uniform graph query system must first test a few system queries. A simple CLI program is introduced used to select the datastore, run queries, and provide appropriate feedback in order to accomplish this. If an error occurs, the developer can modify the query or make changes to the relevant software component.

The utility can be viewed as a starting point for developing client-side middleware that acts as an interface between the client and the uniform graph query system. It is indeed extremely useful for specific test cases where a client might need a different output format. This implementation runs for one persistence mode at a time.

4.3 Cypher Transformation - From Plain Cypher to JSON-like Result

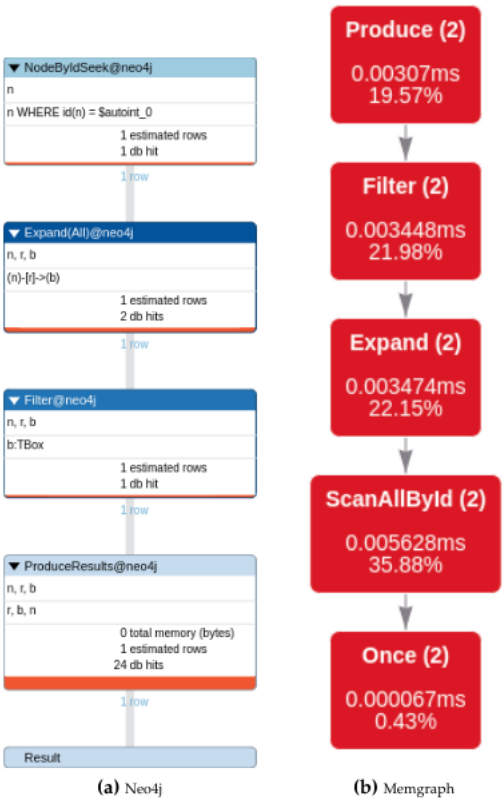
In this section, we will explain how a new query is received by the system, transformed into an execution plan, and returned to the client as a consumable result. In addition, a conversion of result will be described using short code snippets. Both taxonomies are split into 2 subsequent sub-sections.

4.3.1 Query Transformation

When a query is received by their respective driver (in the case of Neo4j) or adapter (in the case of Memgraph), it goes through a series of transformation steps before being executed by the underlying database. In this section, we will go over the fundamentals of the query transformation steps that apply to both of the system's graph datastores and explain them with the help of short code snippets.

The task of running a query is broken down into operators, each of which performs a specific piece of work. The operators are arranged in a tree-like structure known as an execution plan. Each operator in the execution plan is represented in the tree as a node. Each operator accepts zero or more rows as input and outputs zero or more rows. This means that the output of one operator is used as the input for the next. Joining two branches in the tree combines input from two incoming streams and produces a

single output. Fig. 4.2 illustrates the execution plans for a sample query run within the implemented system.



(a) Neo4j **(b)** Memgraph

Figure 4.2: Query Execution Plan

Neo4j driver exposes a Cypher channel over which database work can be carried out [6]. This work itself is organized into sessions, transactions and queries. Sessions are always associated with a single transaction context, which is typically a single database. Sessions, by utilizing the bookmark mechanism, also guarantee correct transaction sequencing, even when transactions occur across multiple cluster members. Fig. 4.3 depicts this phenomenon.

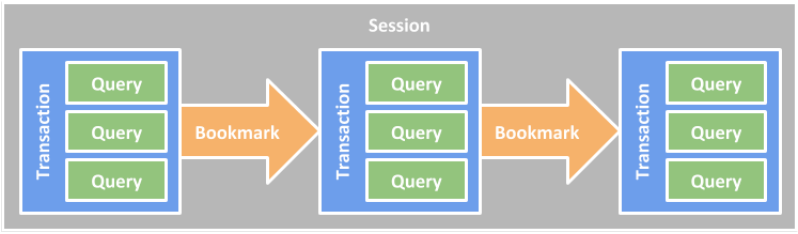


Figure 4.3: Neo4j's Cypher Workflow [6]

Neo4j makes use of *Session* class in the API. A Neo4j session is a logical container for any number of transactional units of work that are causally related. Within a clustered environment, sessions automatically provide guarantees of causal consistency, but multiple sessions can also be causally chained if necessary. Sessions serve as the highest level of containment for database activity. They can be configured in a number of different ways. This is carried out by supplying configuration inside the session constructor like the snippet below.

```
with driver.session(...) as session:
    # transactions go here
```

Listing 1: Neo4j's Session Class snippet

On the other hand, Memgraph has a *Cursor* class. Cursors are created by the *connection.cursor()* method and they are bound to the connection for the entire lifetime. Cursors created by the same connection are not isolated, any changes done to the database by one cursor are immediately visible by the other cursors [4]. A sample implementation of the *Cursor* class is shown below.

```
def execute(self, index,  query):
    print('executing query: ', index)
    # Create a cursor for query execution
    cursor = self.connection.cursor()
    # Execute the query
    cursor.execute(query)
```

Listing 2: Memgraph's Cursor Class snippet

The similarity between sessions and cursors is that neither is thread-safe.

The next fundamental step is about using transactions. Transactions are discrete units of work that include one or more Cypher Queries. Transactions may include read or write work and are typically routed to an appropriate server for execution, where they are completed in their entirety. In the event that a transaction fails, it must be retried from the beginning. The transaction manager is in charge of this. It requires minimal boilerplate code and allows for a clear separation of database queries and application logic. The following code snippet demonstrates how we use the transaction function in our software.

```
with driver.session() as session:
    # transactions go here - read or write.
    result = session.write_transaction(generate_select_query_function(query))
```

Listing 3: Neo4j's transaction function snippet

Before writing a transaction function it is important to ensure that it is designed to be idempotent. This is because a function may be executed multiple times if initial runs fail [25]. In contrast to Memgraph, which has two modes - explicit transactions and auto-commit transactions, Neo4j has three modes - explicit, auto-commit and managed (read or write) transactions. For Neo4j, explicit transactions support multiple statements and must be created with an explicit call.

For Memgraph, an explicit transaction is initiated by sending the BEGIN query or a specific message from the client side. By default, pymgclient sends that message before the first query, which initiates an explicit transaction. They must be explicitly committed, which is why we must call the commit (or rollback) method after completion. While tackling a challenge a Memgraph developer mentioned to me, "*We allow only pure cypher queries (MATCH, RETURN, CREATE...) in explicit (multicommand) transactions.*" As a result, we explicitly invoke the function once more, as we did previously for Neo4j.

```
def execute(query):
    # execution code ...
    self.connection.commit()
```

Listing 4: Memgraph's transaction function snippet

In summary, before performing transactions at the database layer, queries are transformed using the Cypher workflow, which allows for efficient execution.

4.3.2 Result Transformation

In this sub-section, we will go through the result transformation from database layer to generic output format with the help of short representational snippets.

Query results are typically consumed as a stream of records [25]. The Neo4j driver provide a way to iterate through that stream. Only one result stream can be active at any given time

within a session. As a result, if the result of one query is not completely consumed before the result of another query is executed, the remainder of the first result is automatically buffered within the result object. It was quite challenging working with Neo4j's result as we had to manually serialize results which will be explained in detail in section 4.4. Memgraph, on the other hand, had built-in *Cursor* class methods that helped us achieve the desired result. In this case, no additional processing was required.

```
# Result recieved from the database

record  0 : <Record n=<Node id=71 labels=frozenset({'TBox'}) properties={'admin': '', 'comment': '', 'description': 'remove', 'html_info': '', 'identifier': '',
'namespace': 'car', 'ontology_level': 'lower', 'pl': '', 'sing': '', 'title': 'BMProcessor', 'unique': '', 'uri': '', 'url': '', 'version': 'v6.0', 'xsd_type': ''}>>


# Result converted to JSON-like format ustilizing the serializer.

{"result": [{"_id": 71, "_properties": {"admin": "", "comment": "", "description": "remove", "html_info": "", "identifier": "", "namespace": "car", "ontology_level":
"lower", "pl": "", "sing": "", "title": "BMProcessor", "unique": "", "uri": "", "url": "", "version": "v6.0", "xsd_type": ""}, "_labels": ["TBox"]}]}
```

Figure 4.4: Neo4j's result transformation

Transformation from Neo4j's result for a sample query - `MATCH (n) WHERE ID(n) = 71 RETURN n` to JSON-like format can be seen in Fig.4.4

```
# Result recieved from the database

{'admin': '', 'comment': '', 'description': 'remove', 'html_info': '', 'identifier': '', 'namespace': 'car', 'ontology_level': 'lower', 'pl': '', 'sing': '', 'title':
'BMProcessor', 'unique': '', 'uri': '', 'url': '', 'version': 'v6.0', 'xsd_type': ''}



# Result converted to JSON-like format ustilizing the serializer.


{"result": [[{"admin": "", "comment": "", "description": "remove", "html_info": "", "identifier": "", "namespace": "car", "ontology_level": "lower", "pl": "", "sing": "",
"title": "BMProcessor", "unique": "", "uri": "", "url": "", "version": "v6.0", "xsd_type": ""}]]}
```

Figure 4.5: Memgraph's result transformation

Transformation from Memgraph's result for a sample query - `MATCH (n) WHERE ID(n) = 71 RETURN n` to JSON-like format can be seen in Fig.4.5

4.4 Custom Serializer

Serializing records returned by the query was one of the challenges in generating a generic output. In section 3.3.2, we discussed a design decision that explains why JSON-like documents are the best output format; in this section, we will continue to explain granular details of the serializer by first explaining the result serializer and then the response serializer in subsequent sub-sections. The source code featured is part of the system's core implementation and is submitted on CD with this Thesis.

4.4.1 Result Serializer

The Listing 5 code snippet illustrates a function code required to manually transform the record class obtained from the object in Neo4j. This customized implementation will assist us in achieving true *generic output*. Same programmatic approach is used to serialize other databases present in the system. Python's built-in functions, in contrast to the principles, are widely used, and the fundamentals will be covered during the elucidation.

```
def serialize_result_custom(self, index, record):
    print('record ', index, ':', record)  # console print statement
    # Create an empty dictionary
    graph_data_type_list = {}
    # Iterate over the list of records also enumerating it.
    for j, graph_data_type in enumerate(record):
        # Check if the record has string or integer literal.
        if isinstance(graph_data_type, str) or
                isinstance(graph_data_type, int):
            """ Return the keys and values of this record as a dictionary
            and store it inside graph_data_type_dict. """
            graph_data_type_dict = record.data(j)
        else:
            """ Manually convert them into dictionary with __dict__ """
            graph_data_type_dict = graph_data_type.__dict__
            # Remove unnecessary _graph key from the record.
            from the record.
            if '_graph' in graph_data_type_dict:
                del graph_data_type_dict['_graph']
            # Add a _start_node key from the record.
            if '_start_node' in graph_data_type_dict:
                graph_data_type_dict['_start_node'] =
                    graph_data_type_dict['_start_node'].__dict__
                # Add a _labels key of start node from the record.
                if '_labels' in graph_data_type_dict['_start_node']:
                    frozen_label_set = graph_data_type.start_node['_labels']
                    graph_data_type_dict['_start_node']['_labels'] =
                    [v for v in frozen_label_set]
```

```
29                  # Remove unnecessary _graph key from the record.
30                  if '_graph' in graph_data_type_dict['_start_node']:
31                      del graph_data_type_dict['_start_node']['_graph']
32              # Add a _start_node key from the record.
33              if '_end_node' in graph_data_type_dict:
34                  graph_data_type_dict['_end_node'] =
35                  graph_data_type_dict['_end_node'].__dict__
36                  # Add a _labels key of start node from the record.
37                  if '_labels' in graph_data_type_dict['_end_node']:
38                      frozen_label_set = graph_data_type.start_node['_labels']
39                      graph_data_type_dict['_end_node']['_labels'] =
40                      [v for v in frozen_label_set]
41                  # Remove unnecessary _graph key from the record.
42                  if '_graph' in graph_data_type_dict['_end_node']:
43                      del graph_data_type_dict['_end_node']['_graph']
44              # Add other labels for representation from frozenset()
45              if '_labels' in graph_data_type_dict:
46                  frozen_label_set = graph_data_type_dict['_labels']
47                  graph_data_type_dict['_labels'] =
48                  [v for v in frozen_label_set]
49              # print(graph_data_type_dict) # test statement
50          graph_data_type_list.update(graph_data_type_dict)
51      return graph_data_type_list
```

Listing 5: Source code for custom result serializer for Neo4j

To begin, an empty dictionary is created, and then we iterate through the cursor's list of records one by one. The isinstance() function determines whether the object (first argument) is an instance or subclass of the classinfo class (second argument), which in our case is of the type string or integer. The following lines are nested conditions that remove unnecessary metadata and extract important blob fields such as textit *_start_node* and *_end_node*, which are critical fields of the result. By applying the *__dict__*attribute to a class object and obtaining the dictionary. In Python, all objects have the attribute *__dict__* , which is a dictionary object containing all of the attributes defined for that object. A dictionary is created by mapping attributes to their values. Furthermore, due to the

nature of *frozenset()*, some labels are inaccessible. A frozen set is nothing more than an immutable version of a Python set object. While elements of a set can be changed at any time, elements of a frozen set do not change after they are created. As a result, frozen sets can be used as dictionary keys or as elements of another set. However, it is not ordered in the same way that sets are (the elements can be set at any index). As a result, meticulous manual extraction is required. Finally, we update the empty dictionary, which was empty at the beginning of the iteration, and return the updated list to the response serializer.

In addition, a Neo4j official driver API-inspired approach is made available within the Python script. However, it does not produce the desired results.

4.4.2 Response Serializer

The response serializer now formats a serialized data object received from the data serializer. The function definition used in the implementation is depicted in Listing 6. The end-result is a complex object *result* that the driver will use to generate a JSON-document.

```
1 def serialize_response(self, result):
2         return {'result': [self.serialize_data_custom(index, record)
3         for index, record in enumerate(result)]}
```

Listing 6: Source code for custom response serializer for Neo4j

The above snippet illustrates function call used to serialize response post processing the record class. This implementation is the final step before it is sent to JSON serializer as a complex object.

5 Evaluation

Software evaluation is critical in any software system for verifying the system's quality as claimed by the developers. The goal of this chapter is to assess the implemented system using qualitative and quantitative methods. To begin, we will define the experimental setup that is used during the evaluation process. The dataset used in the system is then presented. After which we explain how we conduct experiments and gain insights through integration, observations, and calculations. To conclude the chapter, we will discuss the system flaws discovered during the evaluation.

5.1 Experimental Setup

We decided to start with the software architecture and get the results in Python, a simple yet powerful programming language. The implementation was completed using the JetBrains PyCharm Community IDE to simplify the programming work.

Note: Because of the partial implementation, we decided to exclude RedisGraph's results from the evaluation, as stated previously in Chapter 4.

The computer specification used for implementation is as follows:

- **Model:** Lenovo ThinkPad T430
- **Operating System:** Manjaro Linux 21.1.0
- **Processor:** Intel® Core™ i5-3320M CPU @ 2.60GHz × 4
- **RAM:** 12GB
- **Memory:** 256GB SSD

The following steps are followed to achieve the results:

1. The repository includes Dockerized backend databases Neo4j, Memgraph, and RedisGraph. They can be instantiated with the `$ docker-compose up` command. Memgraph should now begin seeding the database with the configured entry-point script. Neo4j and RedisGraph must take few additional steps in order to load datasets in their respective datastores.

2. At this point, seperate docker containers are created for each database. There is no Graphical User Interface (GUI) available for Memgraph; however Neo4j's remote interface is available at *http://localhost:7474/*.

3. When all of the software components and datasets are complete, the query system is ready to consume queries from a plaintext file called `queries.txt` inside input directory.

4. The `driver.py` script can then be executed to begin query execution process. The console will then display a summary of the connection status and datastore selection.

5. Eventually, once the query and result transformation processes are completed, we obtain a JSON-document in the output directory with the filename `output.json` Hereafter, the client can read the file and consume the results. In addition, the underlying libraries of the respective databases provide us with a console summary of results per query. The time elapsed during the full execution cycle of the plain Cypher queries is also included in the summary.

5.2 Dataset - Cattle Activity Recognition

We utilize a dataset from Md Abdullah Al Mahmud's thesis titled **Improving the Visualization of an Interactive Knowledge Graph Control System for IoT Machine Learning Model Management** for evaluation purposes. This dataset is a part of the KMS domain model v6, which is featured in the Future-IOT Project [9].

5.2.1 Specification

Here is the brief specification for the dataset available at the time of writing:

1. The schema-graph and the instance-graph are both contained within the same graph.

2. Regarding schema-graph nodes labeled as "TBox" (this is all about types, there are no instances):
    - They have one of the following tags: class, property and namespace.
3. There are twelve categories of relations, from which five are listed below:
    - :"has_namespace" (class –has_namespace–> namespace)
    - :"optional_property" (class –optional_property–> property)
    - :"required_property" (class –required_property–> property)
    - :"subclass_of" (class –subclass_of–> class)
    - :"object_property" (class –[:object_property title: "relationName"]–> class)
    - An example : User –[:object_property title: "works_in"]–> Organization
4. For instance-graph nodes labeled as "ABox", there are three things:
    - **Tag:** each node has one tag (ex :"User") that is the name of class that this node is a type of.
    - **Relation:** a node has a relation (:"type") to the class it is a type of (ex. "John" –type-> User)
    - Finally instance nodes have relations between them that have been defined in the schema (ex. "John" -studiesAt-> "UniBamberg").

5.2.2 Graph Schema

A visual representation of the graph will help us better understand the dataset. Fig 5.1 represents the graph schema obtained from Neo4j datastore of the system. This schema is also utilized by the remaining datastores.

In general, there can be *n* number of nodes with the defined specification.

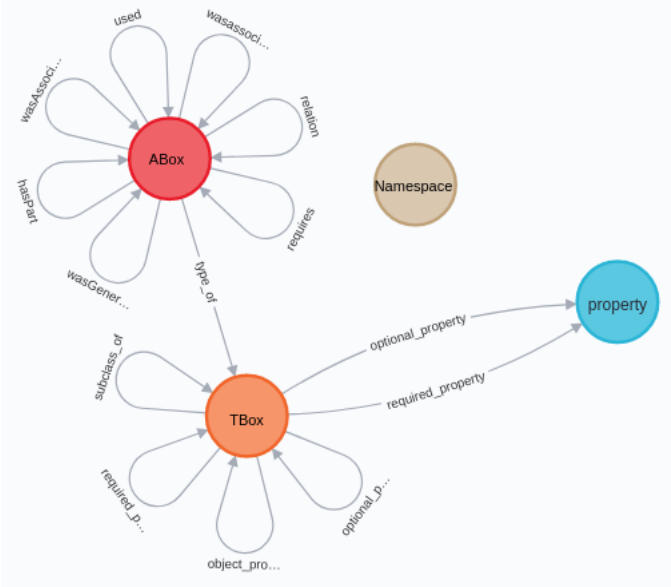


Figure 5.1: Graph schema of the dataset

5.3 Qualitative Evaluation

Qualitative evaluation allows us to gain a thorough understanding of a program or process. It allows for a more in-depth examination of issues of interest and the exploration of nuances. This section aims to present the qualitative evaluation results obtained in collaboration with Md Abdullah Al Mahmud's thesis titled **Improving the Visualization of an Interactive Knowledge Graph Control System for IoT Machine Learning Model Management** *to be submitted*. We decided to perform a simple system integration test that involves the process of assembling the constituent parts of a system in a logical, cost-effective manner, thoroughly testing system execution (all nominal and exceptional paths), and incorporating a full functional check-out.

Collecting qualitative evaluation data requires the use of different tools. Interviews, focus groups, document/material review, and ethnographic participation/observation are some of the most commonly used data collection methods for qualitative data [26]. After completing some general tasks required to achieve the expected results specified in our respective problem statements, we used a form of review.

In the table 5.1, we provide a brief informative report that we formed to better explain the conducted review. Md Abdullah Al Mahmud's Visualization Control System serves as the client for our system.

No.	Evaluation Criteria	Description	Result
1.	System Interface	An interface service end-point capable of communicating with both systems.	Yes
2.	Query Input	The client system generates a generic input format. For instance, a plain text document.	Yes
3	Execution Interruption	A client request to pause or terminate the current query execution process.	No
4.	Result Output	The query results are produced, and the uniform graph query system generates a generic output format. It is a JSON document in this case.	Yes
5.	Integration Test	As a single coherent software system, complete integration of both systems is possible.	Partial

Table 5.1: Qualitative Evaluation Report

The review was divided into five criteria, with specific evaluation items included in the report. A brief description enables understanding of the context of each item, which is followed by a conditional result received in a short answer by us.

1. **System Interface:** Both systems were initially tested for communication and exposing relevant configuration using a single system end-point point. We were able to connect without any major technical challenges because the visualization control back-end system previously ran on Python's Flask framework.

2. **Query Input:** A generic input is generated in this criteria in order to feed our system with plain Cypher queries. This was accomplished by writing a Python script that instructed the front-end component to directly write a `queries.txt` file in the input source of our software.

3. **Execution Interruption:** The system was then specifically engineered to interrupt a query execution cycle by the client program. That did not go as planned, as we lost system connectivity during the test.

4. **Result Output:** We created a generic file that was used as an input to our query system in Query Input; similarly, in this criteria, we tested whether the client could consume the results generated by our system. The outcomes were positive, and the generic format (JSON document) made it easier for the client to perform reads.

5. **Integration Test:** In the final criterion, we integrated a module from the visualization control back-end system as a native component and experimented with a few different Memgraph configurations. As a result, some design methodologies did not work with our software system, resulting in a partial outcome.

Using the aforementioned criteria, we successfully complete a qualitative evaluation of our system.

5.4 Quantitative Evaluation

In the previous section, we conducted a qualitative review of our system, and now we will evaluate it quantitatively. This section aims to achieve one of the objectives stated in chapter 1, section 1.4. Quantitative evaluation is an evaluation method that generates numerical indices primarily from formal (objective) data collection methods, systematic and controlled observation, and a predetermined research design.

We decided to conduct an experiment to determine how much overhead our programming layer added over native database query execution. The purpose of this brief experiment is to assess the impact of additional processing steps performed by the graph query system. The results obtained will aid us in understanding how to improve the system further. An experiment plan is intended to capture the controlled conditions that will be considered during the experimentation. Several scenarios were developed during the experiment and will be discussed in the following subsections, along with the results obtained.

5.4.1 Plan

A set of 30 queries was provided as input to the systems, and they were run 'x' number of times. These queries were written with the dataset's Create Read Update Delete (CRUD) operations in mind. The native database system was designed to run queries directly on the database layer, without the need for any pre or post processing steps that were required in the Uniform Graph Query System. In other words, native database systems would represent the bare-metal performance of the driver-adapter combination and underlying Python libraries. We chose to run the set of queries for 1, 10, and 50 iterations respectively for all scenarios.

5.4.2 Scenarios

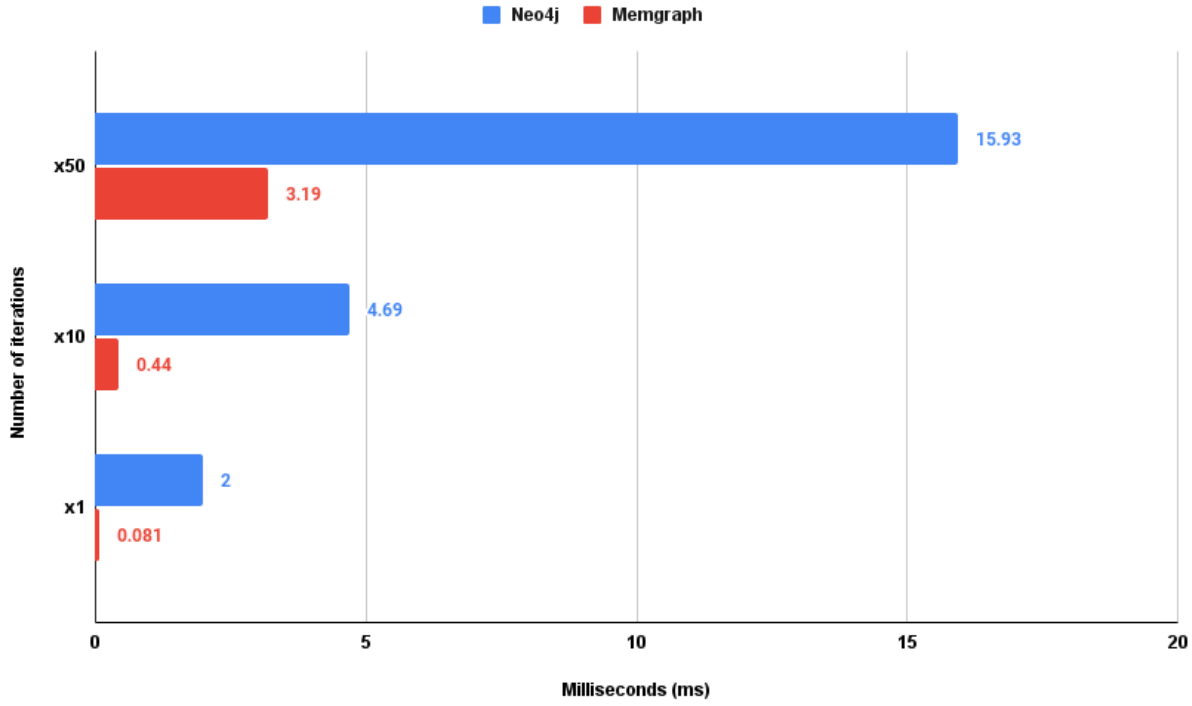
To better understand scenarios, we must first understand the concept of query caching in graph datastores. This phenomenon is shared by both datastores under consideration. The database is in charge of transferring data files from disk to the Page Cache. When you run a query, if the data is not already in cache, it is loaded from disk. The data can be read from cache the next time the query is run, resulting in a faster query. Any query that uses the same data will now benefit from the cached data. Caches are cleared automatically on container startup in our system. To clear caches, Neo4j, for example, uses the `db.clearQueryCaches()` configuration.

1. **Total query execution time on first startup and post caching - Uniform Graph Query System:** When a system is freshly loaded with the dataset for the first time, we record the time it takes to complete a full query execution in our system, including all mandatory steps as stated earlier in chapter 3.1. Then, as planned, we run queries for a predetermined number of iterations, but this time query caching in effect.

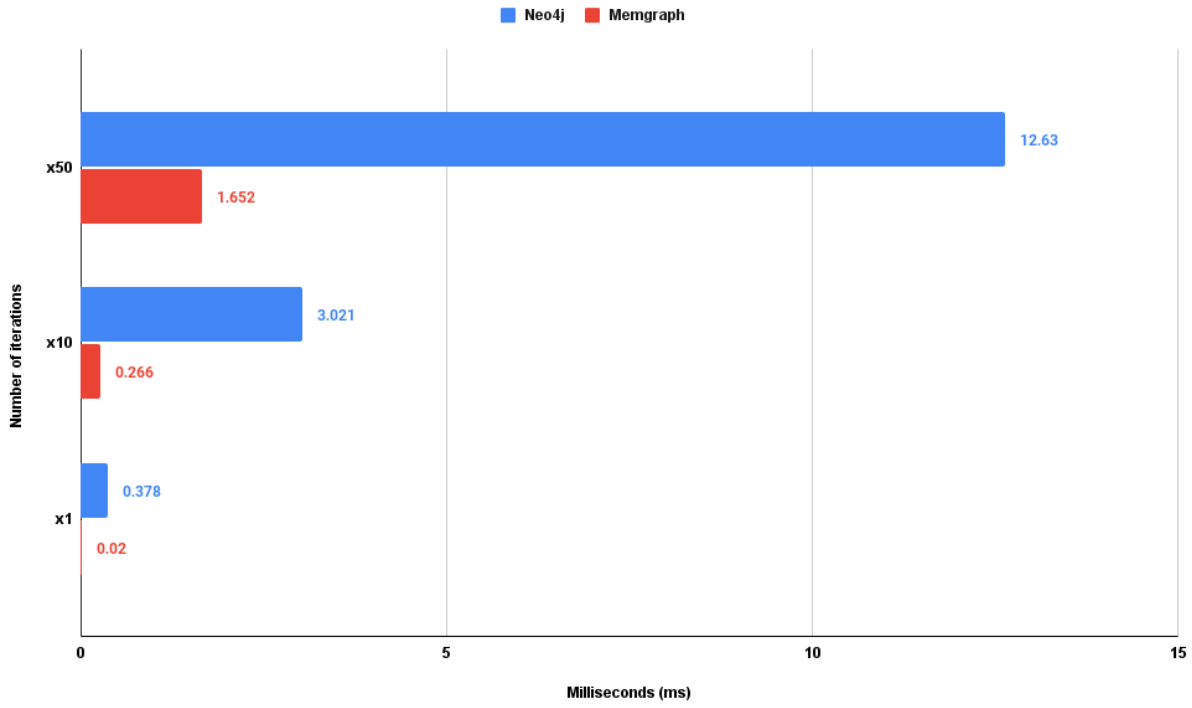
2. **Total query execution time on first startup and post caching - Native Database System:** When a system is first loaded with the dataset, we record the time it takes to complete a full query execution on the database layer without using the programming layer as an interface. We will run Cypher queries designed for both datastores seperately, so no database selection is required. Then, as planned, we run queries for a predetermined number of iterations, but this time we use query caching.

5.4.3 Results

This section illustrates the experiment's results, which are represented in an easy-to-understand graph format.

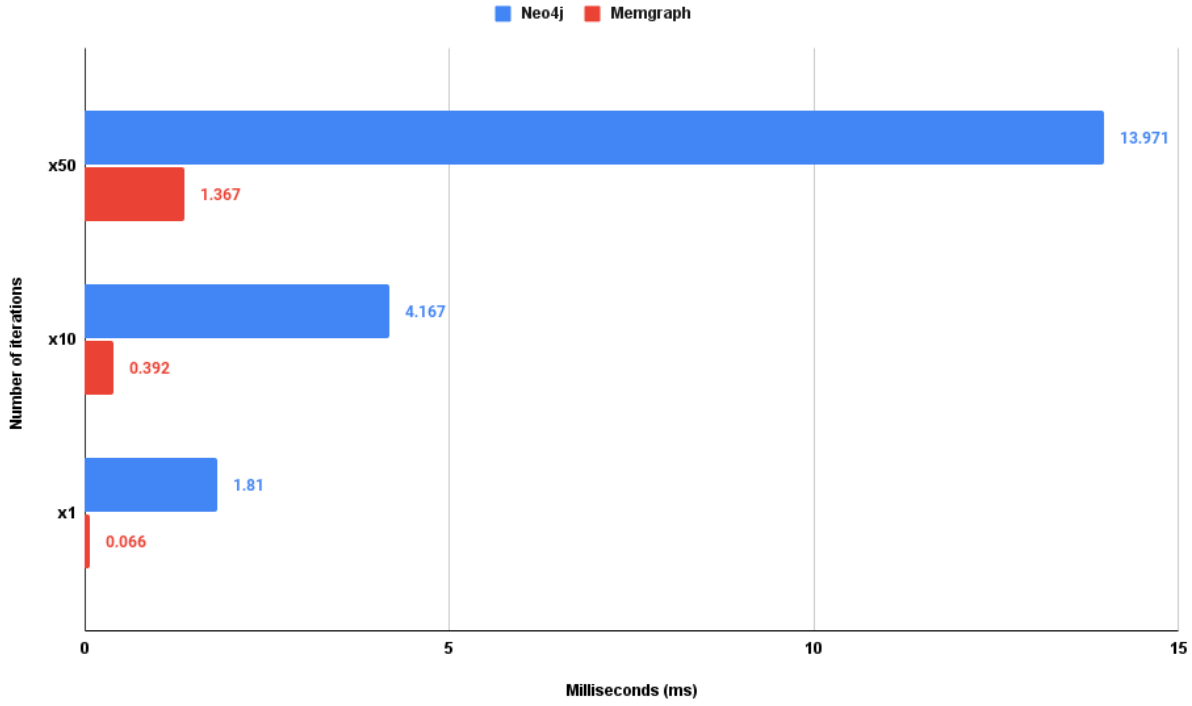


(a) Pre-caching

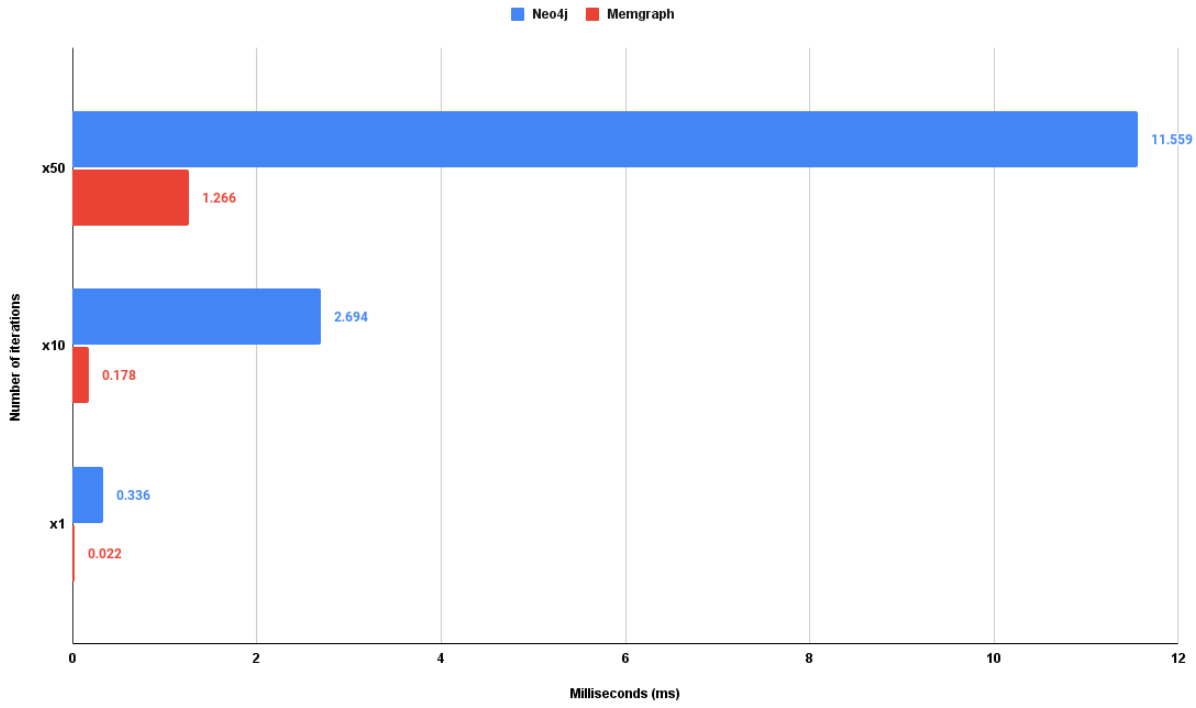


(b) Post-caching

Figure 5.2: Total query execution time on first startup and post caching - Uniform Graph Query System



(a) Pre-caching



(b) Post-caching

Figure 5.3: Total query execution time on first startup and post caching - Native Database System

5.4.4 Summary

After analyzing the results, we can conclude that queries take longer to execute on the first try than on subsequent attempts. Overall, Memgraph is extremely fast and efficient when dealing with small datasets such as ours. Neo4j, on the other hand, has a good number of features for dealing with large datasets; however, it did not perform well with our experiment.

At last, we successfully evaluated our system using qualitative and quantitative methodologies, and we discovered that our system does not under perform and integrates well with other systems.

5.5 Limitations

The architecture can be improved further before it can be delivered in a complete, end-to-end form. This section acknowledges and summarizes the weaknesses of this work. Due to a time constraint on the work, many non-trivial changes were not attempted.

In section 5.2 Dataset - Cattle Activity Recognition, the thesis describes the scenario dataset utilised in this work for the proof of concept implementation. The sample size, however, is small, and schema information is limited. If the schema representations after data import differ, the same Cypher queries will not represent the same operations across the underlying databases. In this case, query translations may be required to accommodate the schema difference. However, translation is a schema-dependent process rather than a universal one. As a result, for each difference in the schema, a new translation will be required. There cannot be a single translation that makes this system schema-independent. During this work, it was determined that appropriate data cleaning is a better approach than writing trivial translations for different schemas. The thesis describes the software design for query processing in section 3.2. Due to the system's schema-dependent nature, the module - Query Analyzer is not fully functional in the current implementation and only returns a predefined query type.

Furthermore, when we compare the additional steps involved in the Uniform Graph Query System, such as datastore selection, query serialization, and generic output format description, we observed no significant difference in performance.

6

Conclusion and Future Work

We proposed a uniform query system for graph databases based on openCypher Projects in this research, that can be regarded as a generic solution for integrating different graph datastores in the context of applications with Polyglot Persistence. The objective of this thesis was to gain a deeper understanding of the graph database querying system and design a "state of the art" system of uniform nature, as stated in section 1.4. To accomplish this, we presented an in-depth study of graph databases in the section 2.1, their data models, as well as usage, and then we explored some related systems with different ideas in chapter 2. This gave us a general understanding of how to design a system to achieve our goal. In section 3.1, we proposed the system architecture for our system, which included the main idea and design decisions made to achieve it. It is worth mentioning that the software architecture is modular and loosely coupled, and the design decisions allow us to query the datastores very efficiently. A proof of concept implementation in chapter 4 allows to select the best implementation and then integrate it into the overall infrastructure. This implies that it will be an essential component of the overall solution. Recommendations were supported by solid reasoning from the respective domains. Furthermore, in order to achieve a fair evaluation of our system, we use a real-world dataset - cattle activity recognition - to validate software. The results obtained during the evaluation process meet our goal of a functional system.

For the future, section 5.5 already sheds light on the missing pieces and disadvantages of this implementation that can be worked on. Besides that, the architecture can be improved further before it can be delivered in a complete, end-to-end form. To achieve better results, the software components and their specifications can be changed. A better evaluation model can further back the learnings obtained from the results.

Bibliography

[1] Amazon AWS. *What Is a Graph Database?* en-US. URL: https://aws.amazon.com/nosql/graph/.

[2] Jesús Barrasa. *RDF Triple Stores vs. Labeled Property Graphs: What's the Difference?* Aug. 2017. URL: https://neo4j.com/blog/rdf-triple-store-vs-labeled-property-graph-difference/.

[3] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://rfc-editor.org/rfc/rfc8259.txt.

[4] *Cursor class — pymgclient 1.0.0 documentation*. URL: https://memgraph.github.io/pymgclient/cursor.html.

[5] *Cypher Query Language Reference, Version 9*. URL: https://github.com/opencypher/openCypher/blob/master/docs/openCypher9.pdf.

[6] *Cypher workflow - Neo4j Python Driver Manual*. en. URL: https://neo4j.com/docs/python-manual/4.3/cypher-workflow/.

[7] Emil Elfrem. "Meet openCypher: The SQL for Graphs". Neo4j Graph Database Platform. In: (Oct. 21, 2015). URL: https://neo4j.com/blog/open-cypher-sql-for-graphs/.

[8] Thomas Frisendal. *Property Graphs*. URL: http://graphdatamodeling.com/Graph%20Data%20Modeling/GraphDataModeling/page/PropertyGraphs.html.

[9] *FutureIOT*. de-DE. URL: https://www.futureiot.de/.

[10] *Graph Modeling Guidelines - Developer Guides*. en. URL: https://neo4j.com/developer/guide-data-modeling/.

[11] Alastair Green et al. *openCypher: New Directions in Property Graph Querying*. type: dataset. 2018. DOI: 10.5441/002/EDBT.2018.62. URL: https://openproceedings.org/2018/conf/edbt/opencypher.pdf.

[12] Aidan Hogan et al. "Knowledge Graphs". In: *arXiv:2003.02320 [cs]* (Jan. 2021). arXiv: 2003.02320. URL: http://arxiv.org/abs/2003.02320.

[13] Erika Kalar. *What is Polyglot Persistence for Databases*. en-US. Section: Uncategorized. Apr. 2018. URL: https://www.objectrocket.com/blog/uncategorized/what-is-polyglot-persistence/.

[14] Jayant B. Karanjekar. "A Uniform Query Framework for Heterogenous Data Management Systems". In: 2018. DOI: 10.29042/2018-4441-4445.

[15] Michael D. Lieberman et al. "Parasol: An Architecture for Cross-Cloud Federated Graph Querying". In: *Proceedings of Workshop on Data Analytics in the Cloud*. DanaC'14. Snowbird, UT, USA: Association for Computing Machinery, 2014, pp. 1–4. ISBN: 9781450329972. DOI: 10.1145/2627770.2627771. URL: https://doi.org/10.1145/2627770.2627771.

[16] Justin Miller. "Graph Database Applications and Concepts with Neo4j". In: *SAIS 2013Proceedings* (May 2013). URL: https://aisel.aisnet.org/sais2013/24.

[17] neo4j. *What is a Graph Database? - Developer Guides*. en. URL: https://neo4j.com/developer/graph-database/.

[18] Fábio Roberto Oliveira and Luis del Val Cura. "Performance Evaluation of NoSQL Multi-Model Data Stores in Polyglot Persistence Applications". In: *Proceedings of the 20th International Database Engineering & Applications Symposium on - IDEAS '16*. the 20th International Database Engineering & Applications Symposium. Montreal, QC, Canada: ACM Press, 2016, pp. 230–235. ISBN: 978-1-4503-4118-9. DOI: 10.1145/2938503.2938518. URL: http://dl.acm.org/citation.cfm?doid=2938503.2938518.

[19] *openCypher · openCypher*. URL: https://www.opencypher.org/.

[20] Yuval Shavit. *What are federated databases?* en. URL: https://searchitchannel.techtarget.com/feature/What-are-federated-databases.

[21] *SIMUTOOL*. en. URL: https://github.com/simutool.

[22] Nigel Small and Stefan Plantikow. *A Deeper Dive into Neo4j 3.0 Language Drivers*. en. June 2016. URL: https://neo4j.com/blog/neo4j-3-0-language-drivers/.

[23] *SQL ... and now GQL · openCypher*. URL: https://www.opencypher.org/articles/2019/09/12/SQL-and-now-GQL/.

[24] *The Property Graph Model - Developer Guides*. en. URL: https://neo4j.com/developer/graph-database/#property-graph.

[25] *The session API - Neo4j Python Driver Manual*. en. URL: https://neo4j.com/docs/python-manual/4.3/session-api/.

[26] Washington State University. *Qualitative Evaluation | Project and Program Evaluation | Washington State University*. en-US. URL: https://ppe.cw.wsu.edu/qualitative-evaluation/.

[27] *What are Graph Data Models??* URL: https://web.stanford.edu/class/cs520/2020/notes/What_Are_Graph_Data_Models.html.

[28] Peter T. Wood. "Query Languages for Graph Databases". In: *SIGMOD Rec.* 41.1 (Apr. 2012), pp. 50–60. ISSN: 0163-5808. DOI: 10.1145/2206869.2206879. URL: https://doi.org/10.1145/2206869.2206879.

Declaration

I hereby declare that, pursuant to § 9 paragraph 12 APO, I wrote the preceding master's thesis independently and did not use any sources or means other than those indicated.

Place, Date Signature