TU WIEN Informatics

Abfrage von Graphdatenbanken mit Ontologien

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Data Science

eingereicht von

Nikola Dragovic, BSc
Matrikelnummer 01528986

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.techn. Maria Magdalena Ortiz de la Fuente, MSc

Wien, 19. Mai 2022

Nikola Dragovic	Maria Magdalena Ortiz de la Fuente

TU WIEN Informatics

Querying Property Graphs with Ontologies

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Data Science

by

Nikola Dragovic, BSc
Registration Number 01528986

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.techn. Maria Magdalena Ortiz de la Fuente, MSc

Vienna, 19th May, 2022

Nikola Dragovic

Maria Magdalena Ortiz de la Fuente

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Nikola Dragovic, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Mai 2022

Nikola Dragovic

Danksagung

Zunächst möchte ich meiner Betreuerin Magdalena Ortiz meinen aufrichtigen Dank dafür aussprechen, dass ich diese Arbeit realisieren konnte. Ich bin dankbar, dass sie sich nicht nur die Mühe gemacht hat, ihre Zeit zu investieren, um meine Ergebnisse und Gedanken zu diskutieren, sondern auch dafür, dass sie meine Forschung mit ihrem aufschlussreichen Feedback und ihren Kommentaren sorgfältig begleitet hat.

Mein besonderer Dank gilt Elem, Emre und Virtual Vehicle für ihre Geduld und Unterstützung bei der Einordnung meiner Forschungsarbeit in einen industriellen Kontext. Sie gaben mir wertvolle Hinweise zur Verbesserung meiner wissenschaftlichen Kommunikationsfähigkeiten und Einblicke in den Prozess der Erstellung von Wissensgraphen, was mir das Selbstvertrauen gegeben hat, diese in Zukunft einzusetzen.

Ich schätze mich glücklich, dass ich unglaubliche Freunde habe, die mich während meines Studiums und während dieser Arbeit ständig unterstützt haben. Es gibt nur wenige Worte, die ausdrücken können, wie sehr ich die Ermutigung schätze, die sie mir weiterhin entgegenbringen. Dazu gehören auch all die unglaublichen Kollegen, mit denen ich in den letzten Jahren zusammengearbeitet habe.

Diese Arbeit wäre ohne die bedingungslose Unterstützung und Liebe meiner Familie unmöglich gewesen. Sie haben mir die Möglichkeit gegeben, mein Studium durchzuführen, und ich werde immer dankbar sein für alles, was sie für mich getan haben.

This work was conducted using the Protégé resource, which is supported by grant GM10331601 from the National Institute of General Medical Sciences of the United States National Institutes of Health.

Acknowledgements

First, I would like to express my sincerest gratitude to my supervisor, Magdalena Ortiz, who allowed me realize this thesis. I am thankful that she not only went through the effort of investing her time to discuss my findings and thoughts, but also for carefully guiding my research with her insightful feedback and comments.

Special thanks to Elem, Emre and Virtual Vehicle, for their patience and assistance in putting my research into an industrial context. Giving me valuable pointers to improve my science communication skills and insights into the process of knowledge graph creation has equipped me with the confidence to use these in the future.

I consider myself lucky for my incredible friends, who have constantly supported me during my studies and throughout this thesis. Few words can express the how much I value the encouragement they continue to show me. This also includes all the incredible colleagues I have worked with in the last years.

This thesis would have been impossible without the unconditional support and love from my family. You gave me the opportunity to pursue my studies, and I will always appreciate everything you have done for me.

This work was conducted using the Protégé resource, which is supported by grant GM10331601 from the National Institute of General Medical Sciences of the United States National Institutes of Health.

Kurzfassung

Bei Ontology-Mediated Querying fragen NutzerInnen Daten mit Hilfe einer Ontologie ab. Eine Ontologie bietet nicht nur ein Mittel, Daten aus heterogenen Quellen miteinander zu verbinden, sondern auch die Möglichkeit, bei unvollständigen Daten Schlüsse zu ziehen. Moderne Systeme erlauben NutzerInnen ihre Daten mit SPARQL 1.1 abzufragen, wobei davon ausgegangen wird, dass diese in einem relationalen Schema gespeichert sind. Obwohl diese Systeme von Graph-strukturierten Daten ausgehen, ermöglichen sie keine Abfragen mit Navigationsfunktionen. Unser Ziel ist es, NutzerInnen die Möglichkeit zu geben, eine Neo4j Property-Graph Datenbank mit Ontologien und einer Abfragesprache mit Navigationsfunktionen abzufragen. In unserer Arbeit diskutieren wir die Unterschiede in der Semantik zwischen SPARQL 1.1 und der Property-Graph Abfragesprache Cypher. Abfragen in unserem Framework sollen in Bezug auf eine gegebene Ontologie umschreibbar, und ihre Auswertung hinsichtlich der Datenkomplexität praktisch ausführbar sein. Wir schlagen ein Framework für die Abfrage von Neo4j Property-Graphen mit Ontologien vor. Darüber hinaus definieren wir Bedingungen, die sicherstellen, dass die Antworten in den betrachteten Abfragesprachen übereinstimmen. Unsere Arbeit umfasst auch eine neue Umschreibetechnik für Abfragen in unserem Framework. Weiters zeigen wir, dass unsere Umschreibung vollständig und korrekt ist und dass die Beantwortung von Abfragen praktisch umsetzbar ist. Abschließend stellen wir eine Implementierung unserer Umschreibung vor und diskutieren die Realisierbarkeit unseres Ansatzes anhand eines Anwendungsfalls aus dem Bereich des autonomen Fahrens, der von der Virtual Vehicle Research GmbH bereitgestellt wurde. Unsere Ergebnisse weisen darauf hin, dass die Abfrage von Property-Graphen mit Ontologien in der Praxis realisierbar ist. Darüber hinaus zeigt sich, dass wir Property-Graph Datenbankmanagementsysteme nutzen können, um Navigationsabfragen in Bezug auf Ontologien zu beantworten.

Abstract

In ontology-mediated querying, users query their data by the means of an ontology. Not only does an ontology provide a means to connect data from heterogeneous sources together, but also a way to reason about incompleteness in the data. State of the art systems allow users to query their data with SPARQL 1.1, which is assumed to be stored in a relational schema. Despite the fact that these systems assume graph-structured data, they do not facilitate querying with navigational features. We aim to enable users to query a Neo4j property graph database with ontologies and navigational features in the query language. In our work, we discuss the differences in semantics between SPARQL 1.1 and the property graph query language Cypher. Finally, queries in our framework should be rewritable with respect to an input ontology, and evaluating them should be tractable in data complexity. We propose a framework for querying Neo4j property graphs with ontologies. Further, we define conditions which ensure that the answers given by the query languages under consideration coincide. Our work also includes a novel rewriting technique for queries in our framework. In addition, we show that our rewriting is complete and correct, and query answering is feasible. Finally, we present an implementation of our rewriting and discuss the viability of our approach based on a use case from the autonomous driving industry, provided by Virtual Vehicle Research GmbH. Our results indicate that querying of property graphs with ontologies is viable in practice. Furthermore, it shows that we can make use of the property graph database management system to answer navigational queries with regard to ontologies.

Contents

Kurzfassung **xi**

Abstract **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 1
1.3 Research Questions . . . 2
1.4 Solution Approach . . . 4
1.5 Thesis Structure . . . 4

2 Background and State Of the Art **5**
2.1 Property Graphs . . . 5
2.2 Ontologies . . . 7
2.3 Graph Querying . . . 12
2.4 Ontology-Mediated Querying . . . 19
2.5 Ontology-Mediated Querying in Practice . . . 20

3 Query Language Design **27**
3.1 Preliminaries . . . 27
3.2 Query Language . . . 30

4 Ontology-Mediated Querying by Rewriting **33**
4.1 A Naïve Rewriting . . . 33
4.2 Termination of Rewriting . . . 41
4.3 Correctness of Rewriting . . . 42
4.4 Query Answering with Rewriting . . . 47

5 Implementation **51**
5.1 Object Representation . . . 52
5.2 Query Parsing . . . 57
5.3 Rewriting . . . 57
5.4 Cypher Translation . . . 59
5.5 Project Structure, Dependency Management and Building . . . 60

5.6 Example Rewriting 61
5.7 Summary 61

6 A Real-World Use Case **65**
6.1 Dataset Description 66
6.2 VVR Property Graph Description 67
6.3 Adding Domain-Specific Knowledge 71
6.4 Rewriting 71
6.5 Query Evaluation 74
6.6 Summary 74

7 Conclusion and Future Work **77**
7.1 Research Questions 77
7.2 Future Work 79

List of Figures **81**

List of Tables **83**

List of Algorithms **85**

Bibliography **87**

CHAPTER 1

Introduction

1.1 Motivation

The ontology-mediated querying (OMQ) paradigm has garnered much attention in the last years from both academia as well as industry [Bie16]. Ontologies encapsulate domain-specific semantic knowledge of the data. Instead of using the vocabulary of the data schema, users query the data through the ontology with the vocabulary they are familiar with. An OMQ system can integrate data from multiple sources with different schemata, but users can still formulate queries relying on a unified, global view of the data. The answers to user queries are enriched with implied knowledge from the ontology, adding results from incomplete data. The combination of an ontology with data is referred to as a *knowledge base*.

Despite the fact that OMQ assumes and is even tailored for graph-structured data, existing systems lack crucial functionalities present in all query languages for graphs. Even if modern ontology-based data access (OBDA) systems such as Ontop [XLK+20] implement a large portion of SPARQL 1.1 [SH13], the property path functionality is not supported. In essence, property paths enable the users to query whether two objects in the knowledge graph are connected by a user-specified path of possibly arbitrary length. Such query language features are also referred to as *navigational features*. Moreover, such systems use relational database management systems (RDBMS) for query evaluation, which are not optimized for query evaluation in graph databases. This is an obstacle to users who want to leverage the advantages of OBDA with property graph databases.

1.2 Problem Statement

Currently, there is no framework for OMQ which assumes that the underlying data the users are querying through the ontology are stored and accessed as graph databases.

Even if graph schemata can be abstracted to relational schemata, enabling the use of current state of the art OMQ systems, we lose the capabilities of the native graph database management system to efficiently evaluate navigational queries over the sources. A framework for OMQ of property graphs consists of a query language and an ontology language. In addition, queries should be answered with regard to an ontology over the plain data.

Our work focuses on answering queries where the data is stored with Neo4j's proprietary property graph model. Neo4j is one of the world's most popular graph database management systems. The graphs of a Neo4j database can be queried with the Cypher query language [FGG+18]. However, the semantics of Cypher are different to the semantics of SPARQL 1.1 and ontology queries. Cypher has *isomorphism-based* semantics, whereas SPARQL 1.1 and ontologies follow *homomorphism-based* semantics. Therefore, any framework that that aims to answer OMQs over Neo4j property graph databases must ensure an adequate semantics for OMQs with Cypher and that its relationship with the usual homomorphism-based semantics is understood. Moreover, Cypher only supports a subset of the property path fragment of the SPARQL 1.1 query language. As a consequence, we must ensure that the query language in our framework is powerful enough to capture conjunctive queries with navigational features in addition to being implementable in Cypher and SPARQL 1.1.

Finally, we want to enable query answering using standard query engines, and evaluating queries over the plain data. The *query rewriting* methodology is a well-known methodology within OMQ [CGL+07, BOS15, BOS15] to achieve this. In a nutshell, given a query q and an ontology $\mathcal{T}$, the query q should be reformulated to a query q' which encapsulates the knowledge in $\mathcal{T}$ and is evaluated directly over the graph data. In a complete rewriting, the answers of q' coincide with the answers of q over the given data including inferred relations and objects from the combination of $\mathcal{T}$ and the data.

1.3 Research Questions

Our work is guided towards answering the following research questions:

RQ1: What is an appropriate way to enable OMQ of property graphs?

An OMQ framework description should include a definition of an ontology and query language. The query language should allow for conjunctive queries with navigational features. In addition, the semantics of the queries should be well-defined for both Cypher and SPARQL 1.1. Users should be able to query their data using the vocabulary of an ontology, and the answers should consider the implicit knowledge of the ontology. Moreover, the queries should be evaluated without adding the implied facts from the ontology to the data. Answering queries with regard to ontologies should be *tractable* in *data complexity*.

We show the viability of our approach on a concrete use case, provided by Virtual Vehicle GmbH, one of Europe's largest research centers for autonomous driving.

RQ2: How do Cypher semantics affect the answers of navigational queries?

By using Cypher as a target language, we are required to define cases where the answers returned by a Cypher query coincide with the answers of a query language whose answers are based on homomorphisms. Such a definition is necessary to ensure that a rewriting of an input query with regard to an ontology returns the correct answers when executed as a Cypher query. A proper semantics for OMQ with property graphs has not yet been proposed.

In other words, we must investigate investigate which restrictions of paths or query language properties are necessary to achieve this. We will show that we can define a set of syntactic restrictions on the knowledge base such that the answers coincide for Cypher and SPARQL 1.1 semantics. Furthermore, we show that we can define a set of syntactic restrictions on the data and the ontology for which the answers coincide on the canonical model of the knowledge base.

RQ3: What is a suitable rewriting strategy using Cypher as a target language?

The primary objective of a rewriting is to avoid materialization of the implied facts of a knowledge base. Depending on the ontology, the size of the materialization of the knowledge base may be infinite, in which case query evaluation is infeasible.

Therefore, we require a rewriting procedure for our ontology and query language. We will present such a rewriting procedure and show that it can be used with Cypher as a target language for query evaluation.

RQ4: How feasible are Cypher rewritings for OMQ of property graphs?

The feasibility of Cypher rewritings is dependent on the computational complexity of query evaluation in graph databases. Moreover, it is well-known that conjunctive query answering with navigational features is intractable in combined complexity. However, the complexity of query answering is often described in *data complexity*. For query languages with homomorphism-based semantics, the data complexity of conjunctive query answering with navigational features is tractable. Still, the semantics of Cypher can lead to intractability in data complexity.

Based on our rewritings, we show the data complexity of answering queries with Cypher as a target language.

1.4 Solution Approach

The main goal of this thesis is to develop a framework for ontology-mediated querying with navigational features for property graphs that is practical to use in a real-life use case. In our work, we follow the methodology of ontology-mediated querying [Bie16].

As a prerequisite, we introduce the Neo4j proprietary graph model and the ontology language DL-Lite. We define models for DL-Lite which can be interpreted as property graphs. Furthermore, we analyze the semantics of query languages and define sets of roles Ξ which are safe to use in paths. Based on this, we define the notion of Ξ-acyclicity and Ξ-compliance, which are syntactic restrictions on the knowledge base. The query language in our framework is a subset of C2RPQs, but can still express a useful variety of paths. For a knowledge base with Ξ-acyclic data and a Ξ-compliant ontology, we show that in the canonical model the answers under both semantics coincide for our query language.

Our main contribution is a rewriting procedure for our query language and DL-Lite ontologies. Using an ontology and a query as an input, it returns a set of queries which can be evaluated in Cypher. We also prove its termination and correctness, and present a query answering algorithm which uses our rewriting procedure as its core element. To show the feasibility of our approach, we show the computational complexity of query rewriting and query answering in our framework. For this, we leverage traditional methods from complexity theory.

1.5 Thesis Structure

We introduce the relevant concepts of ontology-mediated querying in Chapter 2. This includes the description of property graphs, ontologies and the notion of models in relation to property graphs. Furthermore, we define the semantics of the query languages we consider in our work and present the state of the art for OMQ and OBDA. In Chapter 3 we present the query language for our framework, where we also discuss the answers to queries in our query language under different semantics. There, we also present syntactic restrictions to ensure that the answers coincide in the canonical model of the knowledge base. We describe our rewriting algorithm for ontology-mediated querying in Chapter 4. Moreover, we prove its correctness under homomorphism-based semantics and show that using Cypher as a target language returns the expected answers. In addition, we discuss the computational complexity of an answering procedure based on our rewriting. Chapter 5 outlines our implementation of our rewriting algorithm, which produces a Cypher query given an ontology and a query. We present a use case for OMQ with property graphs in Chapter 6, where we used an ontology to bridge the gap between different datasets in the same Neo4j database. Finally, we summarize our work, answer the research questions, and give an outlook for future work in Chapter 7.

CHAPTER 2

Background and State Of the Art

The main goal of this thesis is to develop a framework for ontology-mediated querying of property graphs with navigational features. Before we can describe our approach, however, we must define each of the aspects this thesis is concerned with. Therefore, we must first describe what property graphs and ontologies *are*. We must line out the fundamental terminologies and definitions. While it may sound trivial at first glance, we must also introduce what queries are in this context. Most importantly, we must define how queries are interpreted, which amounts to describing their semantics. Our framework is designed for Neo4j property graphs. As a consequence, we must also introduce the Cypher query language and its relation to our defined semantics and ontologies. We must bridge the gap between data models, ontologies and query semantics. Finally, we also present examples, known results, and technologies from the realm of the semantic web ontology-mediated data access and querying systems, whose success we build our own work on.

2.1 Property Graphs

Graph databases have experienced an increase in popularity in the last decade. They can be categorized as NoSQL ("Not only SQL") databases, as they are inherently non-relational. *Nodes* and *relationships* are the core elements of a graph database compared to tables in relational databases. As such, nodes and relationships (edges) are often treated as first-class citizens.

Many domains can be naturally expressed with property graphs. For example, the International Consortium of Investigative Journalists organized the data from the Panama Papers leak in a property graph database and made it available for download to the public[1]. Other notable examples of use cases for property graphs include social network

[1] `https://github.com/ICIJ/offshoreleaks-data-packages`, accessed 17th May, 2022

analysis [DKT16], bioinformatics [LRS+16], and, more generally, knowledge graph creation [KBL+21]. Nevertheless, there does not yet exist a standard for property graphs. Moreover, the development of a standard graph query language analogous to SQL is still an ongoing project[2].

Our work is focused on the most popular graph database management system is Neo4j with its query language Cypher. Property graphs are the core data model underlying the Neo4j system. We first provide a brief overview of the Neo4j proprietary model.

2.1.1 Model

The Neo4j proprietary model described by Francis et al. [FGG+18] relies on the three disjoint, countably infinite sets of node identifiers $\mathcal{N}$, relationship identifiers $\mathcal{R}$ and property keys $\mathcal{K}$.

Definition 2.1.1 (Neo4j Property Graphs [FGG+18])**.** Let $\mathcal{L}$ be the countable set of node labels and $\mathcal{T}$ the countable set of relationship types. Then, a property graph G is a tuple $\langle N, R, \mathsf{src}, \mathsf{tgt}, \iota, \lambda, \tau \rangle$, where:

- $N \subseteq \mathcal{N}$ is a finite set of nodes in G.
- $R \subseteq \mathcal{R}$ is a finite set of relationships in G.
- $\mathsf{src} : R \to N$ is a function mapping each relationship to its source node.
- $\mathsf{tgt} : R \to N$ is a function mapping each relationship to its target node.
- $\lambda : N \to 2^{\mathcal{L}}$ is a function mapping each node to a finite set of labels.
- $\tau : R \to \mathcal{T}$ is a function mapping each relationship to a relationship type.
- $\iota : (N \cup R) \times \mathcal{K} \to \mathcal{V}$ is a function mapping a node or relationship and a property key to a value.

For illustration purposes we assume two base types: The set $\mathbb{Z}$ of integers and the set Σ^* of finite strings over the alphabet Σ. The set of values $\mathcal{V}$ is defined inductively as follows:

- Elements of $\mathcal{N}, \mathcal{R}$ are values.
- Elements of the base types $(\mathbb{Z}, \Sigma^*)$ are values.
- true, false, and null are values
- list() is a value. If $v_1, \ldots, v_n$ are values, then $\mathsf{list}(v_1, \ldots, v_n)$ is a value.

[2] https://www.gqlstandards.org/home, accessed 17th May, 2022

- map() is a value. If $v_1, \ldots, v_n$ are values and $k_1, \ldots k_n$ are distinct property keys, then $\mathsf{map}((k_1, v1), \ldots, (k_n, v_n))$ is a value.
- $\mathsf{path}(n)$ is a value if $n \in N$. If $n_1, \ldots, n_m$ are elements of N and $r_1, \ldots, r_m$ are elements of R, then $\mathsf{path}(n_1, r_1, n_2, \ldots r_m, n_m)$ is a value.

From the definition of property graphs it follows that Neo4j property graphs are multi-valued property graphs [AAB+17]. However, multi-values are restricted to the labels of nodes only. Relationships are assigned exactly one relationship type, and each (node, key) or (relationship, key) pair is assigned to exactly one value (which can be null). Nevertheless, such a value can be a list.

2.2 Ontologies

Ontologies are a formalism to define domain knowledge in a structured manner. The formal basis of ontologies are Description Logics (DLs), fragments of first-order logic. DLs have shown to be useful in a variety of scenarios [BHLS17]. Description Logics are especially suited to reasoning about incomplete data. Moreover, they inherently assume that data is incomplete. Thus, it is no surprise that DLs have also become the foundation of reasoning within the Semantic Web.

For our intents and purposes, we will leverage the fact that DLs can reason about incomplete data for querying property graph data. The first step is, however, to formally define DLs and ontologies both syntactically and semantically. We present a DL suited for representing the domain knowledge in a property graph that has desirable computational properties when it comes to query answering.

2.2.1 Syntax

The core building blocks of DLs are basic concepts, roles, individuals and relationships. Basic concepts can be viewed as labels or classes for individuals. For example, the concept Person denotes people in our domain. Roles are binary relations between objects in the domain. To illustrate, we can use the role name friendOf to define pairs of people who are friends. Individuals are, as the name suggests, objects in our domain. Contrary to the common definition of the DL vocabulary, we extend it by a set of relationships. Because edges are *objects* in property graphs, we also define the set of relationship names in our vocabulary. This way, we interpret them as semantic objects.

Definition 2.2.1 (Vocabulary)**.** The DL vocabulary consists of the three countably infinite and mutually disjoint sets

- $\mathsf{N_C}$ of basic concept names,
- $\mathsf{N_R}$ of role names and,

- $\mathsf{N_I}$ of individual names and,
- $\mathsf{N_E}$ of relationship names.

In the case of role names, we are often also interested in the inverse of roles. For a role $r \in \mathsf{N_R}$, r^- denotes the inverse role. We use $\mathsf{N_R^\pm}$ to define the union of the symbols $\mathsf{N_R} \cup \{r^- \mid r \in \mathsf{N_R}\}$ for convenience.

Using the DL vocabulary, we can define concepts. Similar to basic concepts, these can be viewed as labels or classes for individuals. However, we can use concept constructors to define more complex structures than simple labels. The constructors we use in our description logic language are fairly simple.

Definition 2.2.2 (DL Concept Constructors)**.** Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be our sets of basic concept and role names. Then,

- every $A \in \mathsf{N_C}$ is a concept.
- $\exists s$ is a concept if $s \in \mathsf{N_R^\pm}$.

DL concepts can be used to build an ontology, which is a set of terminological axioms. Simply speaking, an axiom is an abstract representation of domain knowledge. An axiom expresses our beliefs by putting conditions on objects in our domain. Axioms are sometimes also referred to as *general concept inclusions* (GCIs). The general intuition behind this term is that an axiom consists of a left-hand side and a right-hand side. We interpret an axiom involving concepts as follows: If an object is described by the concept of the left-hand side, then it can also be described by the concept on the right-hand side (but not the other way around). Similarly, for axioms between roles, if two objects are in a relation described by the left-hand side, then their relation can also be described by the right-hand side. Put together, sets of axioms encapsulate all of our background knowledge.

One prominent family of DLs for which query answering is tractable in data complexity is referred to as DL-Lite in the literature [CGL$^+$07]. DL-Lite has been designed specifically with efficient query answering in mind and is used extensively in knowledge-based systems. As such, the types of concepts and axioms we define can be considered a part of the DL-Lite family.

Definition 2.2.3 (Axioms)**.** Let B and C be arbitrary constructed concepts. Then, $B \sqsubseteq C$ is a positive concept inclusion. Moreover, role inclusions are of the form $s \sqsubseteq t$ for $s, t \in \mathsf{N_R^\pm}$ are also axioms.

We also allow for negative inclusions of the form $B \sqsubseteq \neg C$ and $s \sqsubseteq \neg t$ for $s, t \in \mathsf{N_R^\pm}$.

Negative inclusions describe situations where we consider the objects represented by the concept on the left-hand side *disjoint* from the objects represented by the concepts on the right-hand side. A negative inclusion between roles defines that objects which are in the relation described by the left-hand side can not be in the relation described by the right-hand side. For example, a negative inclusion of the form $\mathsf{Person} \sqsubseteq \neg\mathsf{Machine}$ would describe that Person and Machine share no common objects. Similarly, we would not want objects that are related by friendOf to be also in a relation enemyOf, or for short $\mathsf{friendOf} \sqsubseteq \neg\mathsf{enemyOf}$.

The set of axioms is the terminological component of our DL knowledge base. Therefore, we also call it a *TBox* or *ontology* for short and denote it with $\mathcal{T}$. However, the ontology is only half of what we would consider a knowledge base. Up to now, we have only discussed how we express our conceptual view of a domain with Description Logics. What is usually more interesting is how our conceptual view behaves when we add data to it, because we ultimately want to query the data with our conceptual view.

We only have a limited number of assertions to consider. For one, there are *concept assertions* for objects in our domain. *Role assertions* describe which object domains are connected by an edge object. The edge object also determines the relationship type.

Definition 2.2.4 (Assertions)**.** Let $\mathsf{N_C}$ and $\mathsf{N_R}$ be our sets of concept and role names and $\mathsf{N_I}$ and $\mathsf{N_E}$ our sets of individual and relationship names. Then,

- $a : A$ for $a \in \mathsf{N_I}, A \in \mathsf{N_C}$ is a concept assertion.
- $e : r(a, b)$ for $a, b \in \mathsf{N_I}, e \in \mathsf{N_E}, r \in \mathsf{N_R}$ is a role assertion.

We refer to a set of assertions as an ABox $\mathcal{A}$. Additionally, we refer as $\mathsf{Ind}(\mathcal{A})$ to the set of individuals that occur in $\mathcal{A}$, and $\mathsf{Rel}(\mathcal{A})$ refers to the set of relationships that occur in $\mathcal{A}$.

In property graph terms, these assertions would be equivalent to two things: One, the individual names present in the concept assertions must be in our database. Two, these objects must be annotated with the labels specified in the assertions. Finally, two objects in the data are connected by an edge object that must be of one specific type.

A TBox $\mathcal{T}$ and an Abox $\mathcal{A}$ together form a DL knowledge base (KB) $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. Right now, a KB $\mathcal{K}$ has no *meaning*, because we have only defined the syntax of the KB. As it stands for now, KBs are just an assortment of symbols. Therefore, we must define the semantics of DL KBs so that we can *reason* about the elements therein.

2.2.2 Semantics

The semantics of a DL KB is defined by the notion of *interpretations*. This is, again, a slight variation of the literature standard for DL interpretations to accommodate our edge objects.

Definition 2.2.5 (Interpretations)**.** We consider an interpretation as a tuple $(\Delta_V^{\mathcal{I}}, \Delta_E^{\mathcal{I}}, \cdot^{\mathcal{I}}, \mathsf{src}, \mathsf{tgt}, \mathsf{role})$, where

- $\Delta_V^{\mathcal{I}}$ is a non-empty set called the *individual domain*, containing the objects in our domain.
- $\Delta_E^{\mathcal{I}}$ is a non-empty set called the *relationship domain*, containing the relationships between objects in our domain.
- $\mathsf{src} : \Delta_E^{\mathcal{I}} \to \Delta_V^{\mathcal{I}}$ is a total function that assigns each relationship its *source node*.
- $\mathsf{tgt} : \Delta_E^{\mathcal{I}} \to \Delta_V^{\mathcal{I}}$ is a total function that assigns each relationship its *target node*.
- $\mathsf{role} : \Delta_E^{\mathcal{I}} \to \mathsf{N_R}$ is a total function that assigns each relationship its role name.
- $\cdot^{\mathcal{I}}$ is the *interpretation function*.

The interpretation function $\cdot^{\mathcal{I}}$ maps:

- Each individual name $a \in \mathsf{N_I}$ to an element $a^{\mathcal{I}} \in \Delta_V^{\mathcal{I}}$
- Each relationship name $e \in \mathsf{N_E}$ to an element $e^{\mathcal{I}} \in \Delta_E^{\mathcal{I}}$
- Each basic concept name $A \in \mathsf{N_C}$ to $A^{\mathcal{I}} \subseteq \Delta_V^{\mathcal{I}}$
- Each role name $r \in \mathsf{N_R}$ to $r^{\mathcal{I}} = \left\{(v, w) \mid \exists u \in \Delta_E^{\mathcal{I}} : \mathsf{src}(u) = v, \mathsf{tgt}(u) = w, \mathsf{role}(u) = r\right\} \subseteq (\Delta_V^{\mathcal{I}} \times \Delta_V^{\mathcal{I}})$

In essence, we assign all individuals and relationships from our vocabulary to objects in the interpretation domains. In turn, basic concepts are interpreted as sets of objects in the domain $\Delta_V^{\mathcal{I}}$ and roles as sets of objects that contain the objects that are connected by a relationship object from $\Delta_E^{\mathcal{I}}$ with the appropriate role.

For a concept C, we call $C^{\mathcal{I}}$ the *extension* of C in $\mathcal{I}$. We can extend the interpretation function $\cdot^{\mathcal{I}}$ to general concepts and roles as follows:

- $(\neg A)^{\mathcal{I}} = \Delta_V^{\mathcal{I}} \setminus A^{\mathcal{I}}$
- $(s^-)^{\mathcal{I}} = \{(v, w) \mid (w, v) \in s^{\mathcal{I}}\}$
- $(\neg s)^{\mathcal{I}} = (\Delta_V^{\mathcal{I}} \times \Delta_V^{\mathcal{I}}) \setminus s^{\mathcal{I}}$
- $(\exists s)^{\mathcal{I}} = \{v \mid (v, w) \in s^{\mathcal{I}}\}$

With this definition of an interpretation in place, we can now define the semantics of DL TBoxes and ABoxes. However, we are usually not interested in interpretations that only satisfy part of the TBox. We want to focus on interpretations that satisfy all of the axioms in the TBox. Such interpretations are referred to as *models*.

Definition 2.2.6 (TBox Models)**.** An interpretation $\mathcal{I}$ satisfies a TBox inclusion $C \sqsubseteq D \in \mathcal{T}$ if $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, denoted $\mathcal{I} \vDash C \sqsubseteq D$. The interpretation is a model of $\mathcal{T}$ if it satisfies all inclusions in $\mathcal{T}$.

For ABoxes, we are again not particularly interested in interpretations that only partially cover the assertions made in the ABox. Therefore, we define models of ABoxes similar to models of TBoxes.

Definition 2.2.7 (ABox Models)**.** An interpretation $\mathcal{I}$ satisfies an ABox class assertion $a : A$ if $a^{\mathcal{I}} \in A^{\mathcal{I}}$ and a role assertion $e : r(a, b)$ if $\mathsf{role}(e^{\mathcal{I}}) = r$, $\mathsf{src}(e^{\mathcal{I}}) = a^{\mathcal{I}}$, and $\mathsf{tgt}(e^{\mathcal{I}}) = b^{\mathcal{I}}$. As a shorthand, we write $\mathcal{I} \vDash A(a)$ if $\mathcal{I}$ satisfies $a : A$, and $\mathcal{I} \vDash e : r(a, b)$ if it satisfies $e : r(a, b)$. $\mathcal{I}$ is a model of an ABox $\mathcal{A}$ if it satisfies all assertions in $\mathcal{A}$.

One and the same interpretation can satisfy a TBox, but not an ABox. Thus, when we define models of DL knowledge bases, we only want those interpretations that satisfy both the TBox and the ABox.

Definition 2.2.8 (KB Models)**.** An interpretation $\mathcal{I}$ is a model of a knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ if $\mathcal{I}$ is a model of $\mathcal{T}$ and a model of $\mathcal{A}$.

We call a knowledge base $\mathcal{K}$ *satisfiable* if there exists a model for $\mathcal{K}$. Otherwise, we call $\mathcal{K}$ *unsatisfiable*. For ontology languages in the DL-Lite family, one model holds a special position: The canonical (universal) model. One property that makes the canonical model useful in designing query answering algorithms for DL-Lite is that it can be homomorphically embedded into any other model of the KB [BHLS17].

Definition 2.2.9 (Canonical Models)**.** Let $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ be a KB of our ontology language. Then, we define the canonical model of the KB inductively. We start with the interpretation $\mathcal{I}_0$, obtained in the following way:

- $\Delta_V^{\mathcal{I}_0} = \mathsf{Ind}(\mathcal{A})$
- $\Delta_E^{\mathcal{I}_0} = \mathsf{Rel}(\mathcal{A})$
- $A^{\mathcal{I}_0} = \{a \in \mathsf{Ind}(\mathcal{A}) \mid a : A \in \mathcal{A}\}$
- $a^{\mathcal{I}_0} = a$
- $e^{\mathcal{I}_0} = e$
- For each role assertion $e : r(a, b) \in \mathcal{A} : \mathsf{src}(e) = a, \mathsf{tgt}(e) = b, \mathsf{role}(e) = r$

Then, starting with $\mathcal{I}_0$, the following three rules are exhaustively applied:

1. If $v \in B^{\mathcal{I}_i}, B \sqsubseteq C \in \mathcal{T}$, then add v to $C^{\mathcal{I}_{i+1}}$.

2. If $v \in B^{\mathcal{I}_i}, B \sqsubseteq \exists s \in \mathcal{T}$, then add a fresh element w to $\Delta_V^{\mathcal{I}_{i+1}}$, and a fresh element u to $\Delta_E^{\mathcal{I}_{i+1}}$. If s is the inverse of a role $r \in \mathsf{N_R}$ i.e., $s = r^-$, then define $\mathsf{src}(u) \coloneqq w$, $\mathsf{tgt}(u) \coloneqq v$, and $\mathsf{role}(u) \coloneqq r$. Otherwise, define $\mathsf{src}(u) \coloneqq v$, $\mathsf{tgt}(u) \coloneqq w$, and $\mathsf{role}(u) \coloneqq s$.

3. If $s \sqsubseteq t \in \mathcal{T}$, $(v, w) \in s^{\mathcal{I}}$, then add a fresh element u to $\Delta_E^{\mathcal{I}_{i+1}}$. If both s and t are role names or inverse roles, then define $\mathsf{src}(u) \coloneqq v$, $\mathsf{tgt}(u) \coloneqq w$, and $\mathsf{role}(u) \coloneqq t$. Otherwise, define $\mathsf{src}(u) \coloneqq w$, $\mathsf{tgt}(u) \coloneqq v$, and $\mathsf{role}(u) \coloneqq t$ if the inverse role is s, otherwise $\mathsf{role}(u) \coloneqq r$ for $r^- = t$.

We assume fairness of application of the rules i.e., any rule that can be applied, will eventually be applied. We set the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ (or $\mathcal{I}_{\mathcal{K}}$) to the limit of the sequence $\mathcal{I}_0, \mathcal{I}_1, \ldots$. Note that the canonical model can be infinite. It can be shown that $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ is a model if the knowledge base is satisfiable [BHLS17].

2.2.3 Relation Between Interpretations and Property Graphs

Recall the definition of property graphs from Definition 2.1.1 and interpretations from Definition 2.2.5. If we compare the two definitions, we can see that we can transform a property graph into an interpretation and vice-versa. This exemplifies the strong relation between interpretations of ontologies and graph-structured data (resp. property graphs).

To translate an interpretation into a property graph, we can set the set of nodes in the graph to $\Delta_V^{\mathcal{I}}$ and the set of relationships to $\Delta_E^{\mathcal{I}}$. The functions src and tgt remain largely unchanged, except for their domain and range. We can proceed similarly to define the mapping τ in the property graph from the function role of the interpretation. Finally, the function λ can be constructed from the extension of each of the basic concepts in $\mathsf{N_C}$.

Moreover, we can consider the interpretation $\mathcal{I}_0$ from Definition 2.2.9 as the *minimal model* of the ABox $\mathcal{A}$. The interpretation $\mathcal{I}_0$ contains exactly those elements that are also present in $\mathcal{A}$. With a slight abuse of notation, we set $\mathsf{db}(\mathcal{A}) = \mathcal{I}_0$ as the ABox $\mathcal{A}$ considered as a property graph. In the remainder of this thesis, we will focus on answering queries (with regard to ontologies) by reducing them to answering queries over $\mathsf{db}(\mathcal{A})$.

2.3 Graph Querying

Query languages for graphs have been studied long before the recent advent of graph databases and the semantic web. In the following, we will first describe the relevant types of queries for this thesis and their syntax. This includes the description of the different types of semantics, more specifically *homomorphism-based* and *no-repeated-edges* semantics. We will discuss the relation between these two types of semantics in the context of the Cypher query language [FGG+18, AAB+17, MNT20]. Moreover, we will present known results for the *complexity* of querying data under different semantics. Finally, we discuss how these results relate to querying data in the presence of ontologies.

2.3.1 Syntax

Any discussion of a query language for graph-structured data must begin with a description of a query *syntax*. We will describe the syntax of queries and relevant jargon by means of examples. In simple terms, we describe which queries are permitted. We define different types of query languages in logic notation. More specifically, we use *Datalog* notation for our queries. In this notation, every query has a *head* and a *body*. The head of the query contains the *free answer variables* i.e., the answers to our query. We refer to the number of variables in the head as the *arity* of a query. Queries of arity zero i.e., queries without answer variables, are called *Boolean*. Usually, the variables and individuals in a query are referred to as the *terms* of the query.

Conjunctive queries are some of the most-used queries in data access. As the name suggests, the body of conjunctive queries contains conjunctions of atoms with our answer variables and possibly additional ones. In the realm of graph query languages, CQs are often also referred to as *basic graph patterns*.

Example 2.3.1 (Conjunctive Queries)**.** Consider a database which contains sensor data from an autonomous vehicle. The data are organized in scenes for each driving scenario. Each Scene contains samples that are organized in time with a relation called next. The first Sample of a Scene is identified by the relation first from the Scene to the first Sample. The query

$$q(x) \leftarrow \mathsf{Scene}(y), \mathsf{first}(y, x), \mathsf{Sample}(x)$$

returns the first samples in the data. Note that the head of the query only contains one free variable, but the body has two variables. In Datalog notation, the variable y is implicitly existentially quantified. Hence, conjunctive queries are also a type of positive existential first-order queries.

Definition 2.3.1 (Conjunctive Queries)**.** A conjunctive query (CQ) has the form $q(\vec{x}) \leftarrow \exists \vec{y}.\phi$ where $\vec{x}$ and $\vec{y}$ are disjoint tuples of variables, and ϕ is a conjunction of atoms of the form

1. $A(t)$ where $A \in \mathsf{N_C}$ and $t \in \mathsf{N_I} \cup \vec{x} \cup \vec{y}$.
2. $r(t, t')$ where $r \in \mathsf{N_R}$ and $t, t' \in \mathsf{N_I} \cup \vec{x} \cup \vec{y}$

One limitation of conjunctive queries is that we can only express patterns of fixed size. For example, imagine we wanted to extract *all* the samples of a scene by only using the relations first and next. In short, we want to *navigate* the data along the relationships. This type of query is common in graph database settings. Navigation along relationships can be achieved by defining a regular expression over the relations in the graph. Hence, such queries are also referred to as *regular path queries* (RPQs). Furthermore, because these types of queries enable navigation in the graph, they are also referred to as *navigational graph patterns*.

Example 2.3.2 (Regular Path Queries)**.** Assume we have the same scenario as in Example 2.3.1. We can retrieve all scenes with their respective samples by the query

$$q(x, y) \leftarrow \mathsf{first} \cdot \mathsf{next}^*(x, y)$$

The regular expression $\mathsf{first} \cdot \mathsf{next}^*(x, y)$ denotes that x and y are connected by a path of relations that starts with first and ends with an arbitrary number of traversals of the next relation. Note that in an RPQ there are exactly two variables both in the head and body of the query.

Definition 2.3.2 (Regular Path Queries)**.** A navigational query (RPQ) has the form $q(t, t') \leftarrow \rho(t, t')$, where ρ is a regular expression over the alphabet $\mathsf{N_R}$.

A very simple and common extension of RPQs are 2RPQs. 2RPQs allow regular expressions over the alphabet $\mathsf{N_R^{\pm}}$ i.e., they can use role names and inverses.

Of course, we can combine conjunctive queries and (2)RPQs into a new query language called conjunctive relational path queries (C(2)RPQs). With C(2)RPQs, we gain access to new types of queries that can process more complex relationship structures along with additional restrictions on the variables in the query.

Example 2.3.3 (C2RPQs)**.** We extend the scenario from Example 2.3.1. Assume we wish to specify that the objects connected by first and next are Scene and Sample, respectively. We can express this in the following query, which is a combination of the previous two:

$$q(x, y) \leftarrow \mathsf{Scene}(x), \mathsf{first} \cdot \mathsf{next}^*(x, y), \mathsf{Sample}(y)$$

Definition 2.3.3 (C2RPQs)**.** A conjunctive 2-way conjunctive query (C2RPQ) has the form $q(\vec{x}) \leftarrow \exists \vec{y}.\phi$ where $\vec{x}$ and $\vec{y}$ are disjoint tuples of variables, and ϕ is a conjunction of atoms of the form

1. $A(t)$ where $A \in \mathsf{N_C}$ and $t \in \mathsf{N_I} \cup \vec{x} \cup \vec{y}$.
2. $\rho(t, t')$, where ρ is a regular expression over the alphabet $\mathsf{N_R^{\pm}}$

Finally, we define unions of queries in a generic way. We will use the letter "U" to denote unions of queries with the abbreviations we have introduced so far. For example, the term UCQ denotes a union of conjunctive queries.

Definition 2.3.4 (Unions of queries)**.** A union of queries is a set of queries q which have the same arity.

2.3.2 Semantics

Up to now, we have only defined the syntax of query languages. However, no discussion of query languages is complete without defining the *semantics* of queries. In the following, we will describe semantics for C2RPQs. Note that these semantics can also be applied to CQs and (2)RPQs, as they are a subset of C2RPQs.

There have been many different semantics proposed for graph query languages that have also been implemented in database systems [AAB+17]. We will focus on the two semantics which are relevant to this thesis: Homomorphism-based semantics and no-repeated-edges semantics. In both cases, we will assume *set* semantics. Homomorphism-based set semantics (h-semantics) are the standard semantics in ontology-mediated querying [BHLS17, CGL+07, BOS15, Bie16] and underlie the SPARQL 1.1 query language [SH13]. No-repeated-edges semantics (nre-semantics), also known as *trail* semantics [MNT20] are the backbone of the Cypher query language [FGG+18, AAB+17]. Even if Cypher assumes bag semantics, we can explicitly use set semantics, as we will show later. The distinction between bag and set semantics becomes relevant when we discuss the complexity of querying under different semantics, especially when we factor in extensions such as unions and projection.

We define answers to queries in interpretations. Since interpretations can be viewed as property graphs, we can cover the discussion of querying of property graphs at the same time. Furthermore, answers in OMQ rely on knowledge base models, which are interpretations as well.

Query answers in an interpretation $\mathcal{I}$ (or a property graph $\mathsf{db}(\mathcal{A})$) are defined by *matches*. Simply put, matches are mappings from the variables in the query to objects in the domain $v_1^{\mathcal{I}}, \ldots, v_n^{\mathcal{I}} \in \Delta_V^{\mathcal{I}}$ of an interpretation $\mathcal{I}$. Still, we can not define the semantics of queries before we define what a *path* exactly is. We base our definition of paths and their semantics on the work of Bienvenu et al. [BOS15]. Nevertheless, we modify the definition to account for the use of edge objects.

Definition 2.3.5 (Paths)**.** A sequence $u_1 u_2 \ldots u_n, n \geq 0$ is a path from v_0 to v_n in an interpretation $\mathcal{I}$ if all $u_i \in \Delta_E^{\mathcal{I}}$, $\mathsf{src}(u_1) = v_0$, $\mathsf{tgt}(u_n) = v_n$, and for every $1 \leq i \leq n$: $\mathsf{src}(u_{i+1}) = \mathsf{tgt}(u_i)$.

Sometimes it is more useful to talk about the roles occurring in a path rather than the individuals and relationships that constitute a path.

Definition 2.3.6 (Path Labels)**.** The label $\lambda(p)$ of a path $p = u_1 u_2 \ldots u_n$ in an interpretation $\mathcal{I}$ is the word $r_1 r_2 \ldots r_n$ where $r_i = \mathsf{role}(u_i)$ for all $i < 0 \leq n$. If $n = 0$, then $\lambda(p) = \epsilon$.

The *label* of a path can be considered as the "route" we take in an interpretation from the first object on the path to the last one.

For the discussion of semantics, it is helpful to imagine a query as a graph. As the name already suggests, h-semantics describe a semantics where the matches to queries are homomorphisms from the query to the underlying interpretation.

We begin the definition of C2RPQ h-semantics by defining the semantics for a regular path expression. The matches of path expressions are based on the labels of a path. For h-semantics, we are not concerned whether an edge is traversed twice.

Definition 2.3.7 (Homomorphism-Based Semantics of Path Expressions)**.** Let L be a regular language over $\mathsf{N}_\mathsf{R}^\pm$. Then, the semantics of L in an interpretation $\mathcal{I}$ is defined as follows:

$$L_h^\mathcal{I} = \{(v_0, v_n) \mid \text{there is a path } p = u_1 \dots u_n \text{ from } v_0 \text{ to } v_n \text{ such that } \lambda(p) \in L\}$$

Finally, we can define matches for queries in h-semantics.

Definition 2.3.8 (Homomorphism-Based Matches)**.** A homomorophism-based match (*h-match*) for a C2RPQ q in an interpretation $\mathcal{I}$ is a mapping π from the terms in q to elements in $\Delta_V^\mathcal{I}$ such that:

- $\pi(c) = c^\mathcal{I}$ for each $c \in \mathsf{N}_\mathsf{I}$,
- $\pi(t) \in A^\mathcal{I}$ for each atom $A(t)$ in q, and
- $(\pi(t), \pi(t')) \in L(\rho)_h^\mathcal{I}$ for each $\rho(t, t')$ in q.

From Defintion 2.3.8, we can see that *joins* are implicit in h-semantics. Colloquially speaking, a join is the combination of two single-atom queries with that share variables, similar to joins in relational algebra. Using another analogy from relational algebra, the answer variables are akin to selection.

No-repeated-edges semantics are a subset of homomorphism-based semantics. They are based on *isomorphisms*. Matches for queries must have an isomorphism from the edges in the query graph to the objects of $\Delta_E^\mathcal{I}$ of the interpretation. The *simple paths* semantics are closely related to nre-semantics, only that the isomorphism is between nodes of the query and the objects of $\Delta_V^\mathcal{I}$ of the interpretation. Similar to before, we first define the semantics of regular path expressions in nre semantics.

Definition 2.3.9 (No-Repeated-Edges Semantics of Path Expressions)**.** Let L be a regular language over N_R. Then, the no-repeated-edges semantics of L w.r.t. an interpretation $\mathcal{I}$ is defined as follows:

$$L_{nre}^\mathcal{I} = \{(v_0, v_n) \mid \text{there is a path } p = u_1 \dots u_n \text{ from } v_0 \text{ to } v_n \text{ such that each } u_i \text{ is distinct and } \lambda(p) \in L\}$$

For our framework, we can define a semantics which is equivalent to h-semantics except for the semantics of path expressions. In the next section, we will discuss how we can express queries in Cypher such that they follow exactly these semantics.

Definition 2.3.10 (No-Repeated-Edges Matches)**.** A no-repeated-edges match (*nre-match*) for a C2RPQ q in an interpretation $\mathcal{I}$ is a mapping π from the terms in q to elements in $\Delta_V^{\mathcal{I}}$ such that:

- $\pi(c) = c^{\mathcal{I}}$ for each $c \in \mathsf{N_I}$,
- $\pi(t) \in A^{\mathcal{I}}$ for each atom $A(t)$ in q, and
- $(\pi(t), \pi(t')) \in L(\rho)^{\mathcal{I}}_{nre}$ for each $\rho(t, t')$ in q.

It is easy to see that every nre-match is an h-match, while the converse is not true. Lastly, we define the notion of *answers* to queries.

Definition 2.3.11 (Homomorphism-Based Answers)**.** A tuple $(a_1, \ldots, a_n) \in \mathsf{N_I}$ is an *h-answer* to a query $q(\vec{x}) = \exists \vec{y} \phi$ in an interpretation $\mathcal{I}$ if there exists an h-match π such that $\pi(x_i) = a_i^{\mathcal{I}}$ for every $x_i \in \vec{x}$. We denote the set of h-answers for for a query q w.r.t. an interpretation $\mathcal{I}$ as $\mathsf{ans}_h(q, \mathcal{I})$.

Definition 2.3.12 (No-Repeated-Edges Answers)**.** A tuple $(a_1, \ldots, a_n) \in \mathsf{N_I}$ is an *nre-answer* to a query $q(\vec{x}) = \exists \vec{y} \phi$ in an interpretation $\mathcal{I}$ if there exists an nre-match π such that $\pi(x_i) = a_i^{\mathcal{I}}$ for every $x_i \in \vec{x}$. We denote the set of nre-answers for for a query q w.r.t. an interpretation $\mathcal{I}$ as $\mathsf{ans}_{nre}(q, \mathcal{I})$.

2.3.3 A Brief Introduction to the Cypher Query Language

The Cypher query language is used by Neo4j's graph databases [FGG+18]. It is tailored to the Neo4j property graph model. As a graph query language, it supports basic graph pattern matching in a "ASCII-art" graphical format. This feature enables users to visualize the pattern that should be matched in their query. Moreover, these graph patterns also allow users to specify an arbitrary length of a path between two nodes in the graph. Cypher also supports the creation of graphs (i.e., the command `CREATE`) and modifications of the graph (i.e., `DELETE`, `SET` or `MERGE`). However, we are focusing on the querying capabilities on Cypher in this work.

The basic structure of a Cypher query is shown in Figure 2.1. It consists of a `MATCH`, `WHERE`, and `RETURN` clause. The `MATCH` clause defines the query graph which should be matched. In the `WHERE` clause, we can add filters to the matches of the query, similar to the selection operator in relational algebra. Finally, the `RETURN` clause defines the variables which should be returned in the query. In this thesis, we will only use nodes as return objects. However, Cypher can return a wide range of values, such as property values [FGG+18] of a node or relation. Needless to say, Cypher also recognizes the `UNION` keyword for unions of queries.

```
MATCH ... [MATCH ...]*
WHERE ...
RETURN ...
```

Figure 2.1: Basic structure of a Cypher query

By default, Cypher follows no-repeated-edges bag semantics in the `MATCH` clause. However, there are a number of possibilities to bring Cypher "closer" to the behavior of h-semantics. Firstly, Cypher supports the explicit use of set semantics by including the `DISTINCT` keyword in the `RETURN` clause. Second, we can simulate h-semantics for CQs in Cypher by making the join operation, which is implicit in h-semantics, explicit. Cypher performs a join operation for each shared variable of the `MATCH` clauses in the query. Therefore, we can define a separate `MATCH` clause for each atom in a CQ. Because there are no navigational atoms in CQs, the semantics of a Cypher query constructed in such a way is the same as in h-semantics [AAB+17, CS17].

Nevertheless, it is widely known that Cypher does not support full 2RPQs or even RPQs [AAB+17, FGG+18]. Moreover, Cypher does not support inverses in disjunctions of path expressions. However, Cypher allows arbitrary-length expressions as shown in Example 2.3.2. Still, there is no way in Cypher to enable the use of h-semantics in RPQs. Therefore, we can set the semantics of Cypher as presented in Definition 2.3.10.

2.3.4 Complexity of Querying

For this section, we briefly introduce the following complexity classes which are relevant to our work:

$$\mathsf{AC}^0 \subset \mathsf{NL} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE}$$

Among the most famous complexity classes are P and NP. The class P describes problems which can be solved in polynomial time, while NP describes those that can be solved in non-deterministic polynomial time. At the time of the publication of this thesis, it is still unknown whether $\mathsf{P} \neq \mathsf{NP}$. The complexity class NL describes problems which can be solved in non-deterministic logarithmic space, wherein the complexity class AC^0 is contained. Problems that are contained in P (and by extension, in AC^0 and in NL), are called *tractable*. Finally, we introduce the class of problems which can be solved in polynomial space, PSPACE. NP and PSPACE represent *intractable* problems.

While many might be familiar with the notion of *combined complexity*, the notion of *data complexity* [Var82] is less known. Data complexity has received increased attention in the database community in recent times. It is more specific than combined complexity. In data complexity, we fix the queries and describe the complexity of querying with regard to database size.

The discussion of the query complexity for property graphs is based on the following decision problem: Given a query q and a tuple $\vec{t} \in \mathsf{Ind}(\mathcal{A})$, is $\vec{t}$ an answer to q in $\mathsf{db}(\mathcal{A})$

i.e., is $\vec{t} \in \mathsf{ans}(q, \mathsf{db}(\mathcal{A}))$? We will assume set semantics for all cases. It is known that bag semantics increases the complexity of querying, especially for C2RPQs [AAB+17].

It has been long-known that CQ answering is in AC^0 in data complexity and NP-complete in combined complexity [AHV95]. Moreover, these results hold for CQs with projection as well for both h- and nre-semantics. Because we only include projection, filtering and joins in CQs, the complexity does not change [AAB+17].

Evaluation of C2RPQs is inherently more difficult [AAB+17]. With h-semantics (in this context, the term *arbitrary paths semantics* is also often used), the data complexity jumps to NL. However, the combined complexity remains in NP. Most notably, these complexity bounds hold for SPARQL 1.1, if we restrict ourselves to projection, filtering and joins [AAB+17].

The complexity of evaluating C2RPQs with nre-semantics is even more difficult. Evaluating arbitrary (2)RPQs is already NP-complete, and therefore intractable, in data complexity. The combined complexity remains in NP. However, Martens et al. [MNT20] have described expressions for which RPQ evaluation is tractable with nre-semantics. Important for us, RPQs where each role in a Kleene star appears at most once in the regular expression, are tractable in data complexity [MNT20, Observation 3.7]. These are also referred to as *single-occurrence* RPQs.

Because we use Cypher with set semantics and in such a way that we use a `MATCH` clause for each atom in the query, complexity of evaluation of these queries is the same as in SPARQL, if we restrain navigational features to the ones described by Martens et al.

Nevertheless, we must be careful when we define the query language for our framework that we do not define queries which are not tractable in data complexity if we allow navigational atoms.

2.4 Ontology-Mediated Querying

In ontology-mediated querying (OMQ), we aim to combine the domain knowledge expressed in $\mathcal{T}$ with the assertions made in the ABox $\mathcal{A}$. The standard semantics for queries over ontologies and knowledge bases are homomorphism-based [CGL+07, BOS15].

In general, the answers to a query depend on the model of the knowledge base $(\mathcal{T}, \mathcal{A})$ that we are considering. Some tuples might be answers w.r.t. an interpretation $\mathcal{I}$, but not another interpretation $\mathcal{J}$. One common way to extract the answers the users are looking for is to only return those answers that have a match in *all of the models*. We refer to them as certain answers.

Definition 2.4.1 (Homomorphism-Based Certain Answers)**.** A tuple $(a_1, \ldots, a_n) \in \mathsf{N_I}$ is a *certain h-answer* to a query $q(\vec{x}) = \exists \vec{y} \phi$ in a KB $(\mathcal{T}, \mathcal{A})$ if there exists an h-match π such that $\pi(x_i) = a_i^{\mathcal{I}}$ for every $x_i \in \vec{x}$ in every model of the KB.

We denote by $\mathsf{cert}_h(q, \mathcal{T}, \mathcal{A})$ the certain answers to the query q under homomorphism-based semantics. It is widely known that $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ can be embedded into any model of a DL-Lite KB. Additionally, it can be shown that homomorphism-based C(2)RPQ matches are preserved under homomorphisms. Therefore, if an h-match is present in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, it is present in every model of a KB [BOS15, CGL+07]. Hence, $\mathsf{ans}_h(q, \mathcal{I}_{\mathcal{T},\mathcal{A}}) = \mathsf{cert}_h(q, \mathcal{T}, \mathcal{A})$. However, we can show that nre-matches are *not* preserved under homomorphisms. This is an important aspect that we further address in Chapter 3.

Lemma 2.4.1. *nre-matches are not preserved under homomorphisms.*

Proof. Let $\mathcal{A} = \{a : C\}, \mathcal{T} = \{C \sqsubseteq \exists r, \exists r^- \sqsubseteq C\}$. Then, the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ consists of a chain of C's connected via r-relations. Consider the interpretation $\mathcal{I}$:

$$\begin{aligned}
\Delta_V^{\mathcal{I}} &= \{a, v\},\\
\Delta_E^{\mathcal{I}} &= \{u_1, u_2\},\\
\mathsf{src} &= \{u_1 \mapsto a, u_2 \mapsto v\},\\
\mathsf{tgt} &= \{u_1 \mapsto v, u_2 \mapsto a\},\\
\mathsf{role} &= \{u_1 \mapsto r, u_2 \mapsto r\},\\
C^{\mathcal{I}} &= \{a, v\},\\
r^{\mathcal{I}} &= \{(a, v), (v, a)\},
\end{aligned}$$

which is a model of $(\mathcal{T}, \mathcal{A})$. Let $q(x) = \exists y.r \cdot r \cdot r(x, y)$ be a query. Then, there are infinitely many nre-matches in the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, but none in the model $\mathcal{I}$. □

Nevertheless, we can still present known complexity bounds for ontology-mediated querying for DL-Lite. The discussion of the query complexity for ontologies is based on the following decision problem: Given a query q and a tuple $\vec{t} \in \mathsf{Ind}(\mathcal{A})$, is $\vec{t}$ an answer to q in every model of $(\mathcal{T}, \mathcal{A})$ i.e., is $\vec{t} \in \mathsf{cert}_h(q, \mathcal{T}, \mathcal{A})$?

CQ answering is AC^0 in data complexity and NP-complete in combined complexity [CGL+07]. For C(2)RPQs, the data complexity increases to NL, while the combined complexity is in PSPACE [BOS15]. Note that in the case of CQs, the worst-case complexities are in-line with the complexity of query answering over plain graphs. However, the complexity of C(2)RPQ answering increases compared to the case where we do not have to consider a TBox.

2.5 Ontology-Mediated Querying in Practice

Ontologies can be considered a part of the *semantic web technologies* stack. They form the core of reasoning and establishing a shared vocabulary in practice. The recommendations and standards issued by the World Wide Web Consortium (W3C) have been paramount in enabling the development of tools and systems with ontologies. Most important for us

are the Resource Description Framework (RDF), the Web Ontology Language OWL 2, and the query language for RDF, SPARQL 1.1.

RDF data can be seen as graph-structured data [CLW14]. It is a more simple scheme than property graphs, as data are organized in sets of triples. Each triple consists of a subject, predicate and object. Using property graph terminology, the subject and object are interpreted as nodes, and the property is the label of the directed relation between the subject and object. Subjects are either IRIs or blank nodes (similar to anonymous objects in the canonical model of an ontology). Objects are either IRIs, blank nodes or literals, describing data values.

Much of the work we describe in the following is based on an RDF representation of the data, upon which an ontology is added. To emphasize, the OBDA paradigm includes a layer where data in a database is mapped to an RDF graph. Moreover, the underlying data are queried by the users with SPARQL rather than the native query language of the data store. We also discuss how ontologies can be integrated into queries in an efficient manner by introducing query rewriting. Finally, we give a brief overview of the work that has been done in the intersection between the semantic web and property graphs, focusing on Neo4j and Cypher specifically.

2.5.1 OWL2 QL

The W3C recommends the OWL 2 Web Ontology Language to define ontologies in interchangable formats, most notably in the W3C's RDF [PSPM12]. OWL 2 provides a DL-based semantics that is referred to as the *Direct Semantics*. The ontologies described in the direct semantics form the class of OWL 2 DL. OWL 2 DL is far too powerful to facilitate reasoning in polynomial time, as it describes the DL $\mathcal{SROIQ}$. For this DL, even consistency checking is already beyond PSPACE [BHLS17].

However, there are three profiles within OWL 2 DL which have been designed with tractable reasoning in mind: OWL 2 RL, OWL 2 QL and OWL 2 EL [HMG+12]. Of these three profiles, OWL 2 QL can express the ontology language we described in Section 2.2.1. We will not go into the details of all available axioms in OWL2 QL. However, we do mention that OWL 2 QL is can express a more powerful version of the DL-Lite variant we have decscribed. For example, the ontology language can be extended with functional relations, which describes DL-Lite$_{\mathcal{F}}$ [CGL+07].

2.5.2 Query Rewriting

The success of ontology-mediated querying has been reliant on the fact that queries in lightweight ontologies such as DL-Lite can be *rewritten* into (sets of) queries that can be evaluated directly over the ABox. Simply speaking, queries q are reformulated with regard to a TBox $\mathcal{T}$ into a new query $q_{\mathcal{T}}$, which incorporates the domain knowledge in $\mathcal{T}$. This way, answering queries can be delegated to the underlying database management system (DBMS), and make use of the optimizations therein.

For CQs, the *first-order-rewritablility* of DL-Lite has been paramount in the development of OMQ in practice. Many popular OMQ systems assume that $\mathsf{db}(\mathcal{A})$ is a relational database e.g, Ontop [XLK+20]. Any CQ can be rewritten with regard to a DL-Lite TBox $\mathcal{T}$ into a UCQ Q such that

$$\mathsf{cert}_h(q, \mathcal{T}, \mathcal{A}) = \bigcup_{q \in Q} \mathsf{ans}_h(q, \mathsf{db}(\mathcal{A})).$$

As a consequence, OMQs can be evaluated directly over relational DBMS [CGL+07, BHLS17].

The first such algorithm for conjunctive query rewriting in DL-Lite was the aptly named PerfectRef procedure proposed by Calvanese et al. [CGL+07]. In a nutshell, the PerfectRef procedure generates all possible ways of replacing query atoms by another query atom that is implied by it. The new query is written as a union of conjunctive queries. In the rewriting, we have to take note of the fact that query atoms can not be rewritten arbitrarily. It is essential for the correctness of the algorithm that shared variables matter, as well as which variables are unbound. An unbound variable is not an answer variable, not a constant, and does not occur in more than one atom. In addition, rewritings create new variables in the query, some of which have to be unified so that a new replacement might be applicable.

Naturally, the PerfectRef procedure did not increase the computational complexity of query answering in DL-Lite. CQ answering with PerfectRef is in AC^0 in data complexity and in NP in combined complexity. However, the number of queries contained in the UCQ generated with this method is exponential in the size of the TBox $\mathcal{T}$. As a consequence, query evaluation could potentially take a lot of time.

In the years since, a number of different rewriting techniques for CQs and DL-Lite have been proposed. Some of these approaches such as the combined approach [KLT+10], included adding individuals to the ABox to facilitate a more efficient rewriting. However, it has been shown that there exists no polynomial-time algorithm for *pure rewriting* of CQs in DL-Lite [KKZ11] without changing the data. Hence, the number of queries generated by PerfectRef is worst-case optimal.

Nevertheless, a more practical pure rewriting has been developed by Kikot et al. [KKZ12] and is known as tree witness rewriting. To understand tree witness rewriting, it is helpful to consider the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ of a DL-Lite knowledge base as a tree with the individuals in the ABox as roots. The general idea is to generate queries for each axiom with an existential on the right-hand side. Only these axioms can generate new objects in the canonical model of the knowledge base. The second step is then combining compatible tree witnesses and replacing the atoms which can be implied by each witness.

As is the case with PerfectRef, the UCQ generated by this procedure has exponential size in the worst case. On the other hand, the tree witness rewriting has polynomial-time rewritings for many real-world ontologies. It has been shown that, in practice, very few tree witnesses have to be generated. In other words, the ontologies of real-life systems are quite simple.

2.5.3 Ontology-Based Data Access

The ontology-based data access (OBDA) paradigm for query answering has been very successful since its inception more than a decade ago [CGL+07, PLC+08]. The OBDA approach combines reasoning over incomplete data in description logics (DLs) with, possibly remote, heterogeneous data sources. The use of ontologies enables the users to enrich their queries with domain knowledge. Early adapters of OBDA designed their systems for querying relational data with SQL [PLC+08]. Recent developments have lead to adding non-relational sources such as MongoDB to OBDA systems [BCC+19].

From a user's standpoint, OBDA enables querying the data with a more user-friendly language than SQL i.e., SPARQL. The vocabulary of the query is designed by domain experts and ontology engineers. Another advantage is *transparency*: An OBDA system can integrate data from multiple sources, but users see the data as one centralized source. OBDA has been used in a variety of scenarios: oil and gas [KHS+17], healthcare [RLT+14], maritime security [BBXK16], and machine monitoring [KKM+16].

An OBDA system can be viewed as a three-layered architecture [XCK+18]. The top layer is the ontology $\mathcal{O}$. It encapsulates the domain-specific knowledge of the users in a DL as a set of axioms. The DL of choice should have some desirable properties in terms of complexity of query answering for the OBDA system to be useful in practice. One such DL is DL-Lite, which was specifically designed with tractable query answering in mind. The current industry standard for expressing ontologies is OWL 2 [PSPM12]. Queries are usually expressed in SPARQL [SH13].

The mapping layer $\mathcal{M}$ links the ontology layer $\mathcal{O}$ to the database schema $\mathcal{S}$. Mappings translate the data from the sources to objects in our ontology. The objects in the ontology form a graph with edges between the objects to signify a relationship. Translations have to be handled with care due to the *impedance mismatch* between data and objects and vice-versa. The World Wide Web Consortium recommends R2RML [DCS12] as a standard to define mappings. However, some systems have implemented their own mapping language, such as Ontop [XLK+20].

Finally, the data source schema $\mathcal{S}$ defines the shape of our data. This is usually a relational schema with tables consisting of rows of columns with primary and foreign keys and constraints. However, note that this high-level definition also allows for non-relational or NoSQL schemata. The three layers $\mathcal{O}, \mathcal{M}$ and $\mathcal{S}$ together form the OBDA specification $\mathcal{P} = \langle \mathcal{O}, \mathcal{M}, \mathcal{S} \rangle$. Queries are answered over a concrete data instance $\mathcal{D}$ that conforms to the schema $\mathcal{S}$.

We provide an intuition of how these OBDA systems work in practice by presenting an example. Assume we wish to extract all persons from a movie domain. We know that the domain contains actors and directors, which are both of type `Person`. This knowledge can be encapsulated with the following DL axioms:

$$\mathsf{Actor} \sqsubseteq \mathsf{Person}$$
$$\mathsf{Director} \sqsubseteq \mathsf{Person}$$

In the relational source schema, the actors and directors are stored in two different tables with their social security numbers as the primary key e.g.,

$$\begin{aligned} \mathit{Actor}: &\quad \{\underline{\text{ssn} : \text{integer}}, \text{name} : \text{string}, \dots\} \\ \mathit{Director}: &\quad \{\underline{\text{ssn} : \text{integer}}, \text{name} : \text{string}, \dots\} \end{aligned}$$

Objects from a database conforming to the source schema are retrieved by the mappings

$$\begin{aligned} \mathsf{Actor}(x) &\rightsquigarrow \text{SELECT ssn FROM Actor} \\ \mathsf{Director}(x) &\rightsquigarrow \text{SELECT ssn from Director} \end{aligned}$$

Using this OBDA specification, we can populate our ontology from the sources. In addition, we can add the fact that Actor and Director are Person to all our objects. This enables us to answer our query

$$q(x) \leftarrow \text{Person}(x)$$

However, when post our query, we are not forced to materialize the objects from the sources if we use an appropriate DL language. For example, we can encapsulate the axioms from our ontology by replacing the Person atom with Actor and Director, respectively:

$$\begin{aligned} q_a(x) &\leftarrow \text{Actor}(x) \\ q_d(x) &\leftarrow \text{Director}(x) \end{aligned}$$

For these two queries q_a and q_d, we know how to obtain the objects from the mappings and, therefore, from the sources directly. Unsurprisingly, the possibility of not having to materialize the ontology has lead to the notion of *virtual knowledge graph* (VKG) systems. An OBDA specification is also referred to as a VKG specification in the literature [XCK+18].

Most OBDA systems implement a form of query rewriting for these *conjunctive queries* we presented here. For example, Ontop uses tree witness rewriting to answer CQs over the data [KKZ12]. More often than not, these systems also cover more aspects of SPARQL 1.1 such as filtering, optional, or aggregation [XLK+20].

As of 2013, the query language SPARQL includes *property paths* as part of its query language [SH13]. In a nutshell, property paths allow us to query which tuples are in a relation expressed as a regular expression. Queries of this type are RPQs. As an example, assume that we extend our movie domain with relations between actors and directors. A relation $\mathsf{friendOf}$ denotes that x is a friend of y. The simple RPQ

$$q(x, y) \leftarrow \mathsf{friendOf}(x, y)$$

returns all pairs of objects that are friends. This query is both a conjunctive query and a RPQ. Still, RPQs allow us to traverse the relationship more than one time i.e.,

$$q(x, y) \leftarrow (\mathsf{friendOf})^+(x, y)$$

With this, we can retrieve all pairs that are connected by a path along the friendOf relation. This second query is not expressible as a conjunctive query because we can not express a path of arbitrary length.

However, popular ontology-based data access (OBDA) systems such as Ontop [XLK+20] or Mastro [CDGL+11] do not support navigational features. While an implementation of property paths is possible, a translation to SQL is cumbersome, and, more importantly, expensive.

2.5.4 Property Graphs and the Semantic Web

Adding ontological knowledge to property graphs has become an emerging field of interest in both the academic as well as the industrial community. There have also been numerous efforts to translate SPARQL queries into Cypher, though to our knowledge no full translation engine has been formulated yet [MR20, ASM22]. As we previously discussed, the semantics of SPARQL and Cypher are inherently different, so any translation must take these into account. This is especially true if property paths should be translated to Cypher. However, we also know that Cypher only supports a fragment of RPQs.

Nevertheless, we want to mention some efforts which are not only relevant to this thesis, but could also be considered in further work to implement a full OBDA system for Cypher such as Ontop has done for relational databases.

Mapping Property Graphs to RDF

A recent effort to execute SPARQL queries in a property graph database has been undertaken by Fathy et al. [FGBH20]. The main motivation for this work was to query the underlying data with a common vocabulary, an advantage we have discussed in the previous section. As a mapping language, the authors used xR2RML, an extension of the R2RML standard for NoSQL databases and query languages [MDFZM17]. First, SPARQL queries are translated into a graph query algebra, and normalized. Then, similar to the approach described above, the mappings are unfolded into the query to generate a Cypher query.

While the authors tested a number of queries, none of them covered the case where a relation would have to be traversed twice. Moreover, only conjunctive queries were tested, for which a semantically equivalent translation always exists. Hence, for the sixteen queries they tested, all of them returned the same results as an RDF triple store. Nevertheless, they demonstrated the use of mappings to make the property graph accessible to SPARQL.

Extensions of RDF have been proposed which could bring property graphs and RDF closer together. One such extension is RDF* (RDF-star), which allows for nesting of triples in the subject or object position of a triple [Har17]. RDF* enables a compact representation of metadata in this way. Still, RDF* should be understood as syntactic sugar on top of RDF. Any RDF* statement can be expressed in RDF.

Querying with Concept Hierarchies

Similar to our work, Cysneiros and Salgado proposed a query rewriting for concept hierarchies and property graphs [CS16]. Again, Neo4j and Cypher was used to demonstrate the viability of their approach. Concept hierarchies are a simple type of ontologies where all axioms are of the type $A \sqsubseteq A_1$, where A, A_1 are basic concept names from $\mathsf{N_C}$. Different to the OBDA approach however, their approach included storing the basic concept names in the graph database as nodes and connecting all objects of a concept to the ontology object in the database with a special role name. This allowed the implementation of a simple rewriting algorithm, but also put constraints on how to formulate the query. For example, the query $q(x) \leftarrow \mathsf{Food}(x)$ has to be formulated as $q(x) \leftarrow \mathsf{instanceOf}(x, y), \mathsf{Food}(y)$. As we can see, this approach is orthogonal to our proposed approach, where we assume the node labels are the classes of the objects in the ABox. However, the rewriting was quite simple: For each node label that occurs in the ontology, a reasoner is used to derive all subclasses. The subclasses are then added to the Cypher query with a disjunction in the `WHERE` clause on the label of the queried node(s). For this, a reasoner from the OWLAPI was used [Hor11].

Neosemantics

Recently, Neo4j has added the `neosemantics` (n10s) [Neo21a] plugin to its distribution. The main functionality of n10s is the import and storage of data in RDF format. The data can then be queried with Cypher. In addition, n10s can also use SHACL to validate the data [KK17]. Moreover, Neo4j property graphs can be exported as RDF triples with n10s.

Because ontologies can be expressed in RDF, n10s also allows importing ontologies. The ontology is then also stored as nodes and relations in the same property graph as the other RDF triples. Similar to the previous case, the semantic objects are connected to the concepts they belong to with special relations in the database. However, n10s only allows for select ontology axioms to be imported into Neo4j. Most notably, axioms with existentials on the right hand side can not be imported. As a consequence, the reasoning capabilities of n10s are also limited to hierarchies. However, n10s allows inference on both concept and role hierarchies.

CHAPTER 3

Query Language Design

The query language is the interface of our framework: Users interact with our system by posting queries and interpreting the answers. Hence, it is important that the query language is powerful enough to express the information desires of our users. In particular, we want to be able to formulate *conjunctive queries with navigational features* i.e., C(2)RPQs.

We define our query language in logic notation. In addition, the query language should be expressible in common graph query languages for the semantic web and property graphs. Specifically, we want the query language to be expressible in SPARQL 1.1 [SH13] and Cypher [FGG+18]. In addition, we would wish that we get the same answers regardless of whether we use SPARQL 1.1 or Cypher. However, we have already shown that the different semantics of path expressions in SPARQL 1.1 and Cypher are not compatible. More specifically, we have provided a proof that no-repeated edges semantics *matches* are not preserved under homomorphisms. This means that we can not use the certain answer semantics that are usually applied in this context [PLC+08, Bie16, BOS15]. Still, we can show that the certain answers under h-semantics and nre-semantics coincide in the canonical model if we place specific restrictions on the query language and the knowledge base.

3.1 Preliminaries

Before we present our query language for ontology-mediated querying with property graphs, we must define some additional terminology. As we have discussed in the previous section, we can define queries in Cypher where only the paths follow nre-semantics. Other than that, the semantics are equivalent to h-semantics, and SPARQL 1.1 semantics, respectively. Our main goal is therefore to identify cases where path nre-path semantics and h-path semantics coincide. However, we must also keep in mind that the navigational

features we allow do not lead to intractability in data complexity when we evaluate the queries with Cypher as a target language.

We can use the paths to define syntactic restrictions on sets of roles in the KB. The general idea is that for sets of roles for which we can guarantee no cycle occurs in the canonical model, nre- and h-semantics for paths overlap. For this, we introduce *role clusterings*, Ξ-*acyclic structures* and Ξ-*compliant TBoxes*. First, we define the set of roles that occur in a path.

Definition 3.1.1 (Path Roles)**.** Let $p = u_1 u_2 \dots u_n, n \geq 0$ be a path from v_0 to v_n in an interpretation $\mathcal{I}$. Then, $\mathsf{roles}(p) = \bigcup_{i=1}^{n} \{r \mid \mathsf{role}(u_i) = r\}$ is the set of roles occurring in $\lambda(p)$ If $p = \epsilon$, then $\mathsf{roles}(p) = \varnothing$.

In the following, we will define which roles can be used in navigational atoms. Role clusterings enumerate the relevant sets of role names from all possible combinations of roles.

Definition 3.1.2 (Role Clustering)**.** A role clustering $\Xi \subseteq 2^{\mathsf{N_R}}$ is a subset of the power set of the role names $\mathsf{N_R}$.

If we wish to ensure that no cycles occur in the canonical model, acyclicity in the underlying ABox of the knowledge base is a prerequisite.

Definition 3.1.3 (Ξ-Acyclic Structures)**.** A structure $\mathcal{I}$ is Ξ-acyclic if there is no path $p = u_1 u_2 \dots u_n, n > 0$ such that $\mathsf{src}(u_1) = \mathsf{tgt}(u_n)$ and $\mathsf{roles}(p) \subseteq \xi$ for any $\xi \in \Xi$.

We also refer to $\mathcal{A}$ as a Ξ-acyclic ABox if the property graph representation $\mathsf{db}(\mathcal{A})$ of $\mathcal{A}$ is Ξ-acyclic. Moreover, we also have to keep in mind that our target query language, Cypher, can only express a subset of RPQs. Our ontology language DL-Lite allows for inverses to appear in subrole axioms. As a consequence, a path that only uses role names can be implied by a path that uses inverses. However, Cypher can not express such navigational patters without raising the data complexity to an intractable level. Therefore, we must also place syntactic restrictions on the TBox to ensure no such path with inverses can be implied by the data and ontology.

Definition 3.1.4 (Ξ-Compliant TBoxes)**.** A TBox $\mathcal{T}$ is Ξ-compliant if for all positive role inclusions $s \sqsubseteq t \in \mathcal{T}$ and all $\xi \in \Xi$:

- If $t \in \xi$, then $s \in \xi$, and
- If $t^- \in \xi$, then $s^- \in \xi$

Finally, we show that the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ preserves Ξ-acyclicity if $\mathcal{A}$ is Ξ-acyclic and $\mathcal{T}$ is Ξ-compliant.

Lemma 3.1.1. *Let Ξ be a role clustering, $\mathcal{A}$ a Ξ-acyclic ABox and $\mathcal{T}$ a Ξ-compliant TBox such that $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is satisfiable. Then, the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ of the KB $\mathcal{K}$ is Ξ-acyclic.*

Proof. Assume that Ξ is a role clustering, $\mathcal{A}$ a Ξ-acyclic ABox and $\mathcal{T}$ a Ξ-compliant TBox. Let $\mathcal{I}_0, \mathcal{I}_1, \ldots, \mathcal{I}_n$ be the sequence of interpretations used in the construction of $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. We show by induction on i that the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ of the KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ is Ξ-acyclic. For the induction start, $\mathcal{I}_0$ is Ξ-acyclic because it contains exactly those assertions made in $\mathcal{A}$, and $\mathcal{A}$ is Ξ-acyclic. For the induction step, assume that $\mathcal{I}_i$ is Ξ-acyclic. To show that $\mathcal{I}_{i+1}$ is Ξ-acyclic, we make a case distinction by the rule that was applied to obtain $\mathcal{I}_{i+1}$ from $\mathcal{I}_i$.

1. If $v \in B^{\mathcal{I}_i}, B \sqsubseteq C \in \mathcal{T}$, then add v to $C^{\mathcal{I}_{i+1}}$.

   This rule does not add any new paths to $\mathcal{I}_{i+1}$. Hence, $\mathcal{I}_{i+1}$ is Ξ-acyclic.

2. If $v \in B^{\mathcal{I}_i}, B \sqsubseteq \exists s \in \mathcal{T}$, then add a fresh element w to $\Delta_V^{\mathcal{I}_{i+1}}$, and a fresh element u to $\Delta_E^{\mathcal{I}_{i+1}}$. If s is the inverse of a role $r \in \mathsf{N_R}$ i.e., $s = r^-$, then define $\mathsf{src}(u) \coloneqq w$, $\mathsf{tgt}(u) \coloneqq v$, and $\mathsf{role}(u) \coloneqq r$. Otherwise, define $\mathsf{src}(u) \coloneqq v$, $\mathsf{tgt}(u) \coloneqq w$, and $\mathsf{role}(u) \coloneqq s$.

   Assume that this rule creates a path $p = u_1 u_2 \ldots u_n, n > 0$ such that $\mathsf{src}(u_1) = \mathsf{tgt}(u_n)$ and $\mathsf{roles}(p) \subseteq \xi$ for a $\xi \in \Xi$. If $\mathcal{I}_i$ is Ξ-acyclic, then the addition of v and u to $\mathcal{I}_i$ has created a cycle in $\mathcal{I}_{i+1}$. It follows that u is an element of the path p. Therefore, $\mathsf{role}(u) \in \mathsf{roles}(p)$. If u is an element of p, $\mathsf{role}(u) \in \mathsf{roles}(p)$ and $\mathsf{roles}(p) \subseteq \xi$, then $\mathsf{role}(u) \in \xi$. By definition, $\Xi \subseteq 2^{\mathsf{N_R}}$. Hence, $\xi \subseteq \mathsf{N_R}$. If $\mathsf{role}(u) \in \xi$ and $\xi \subseteq \mathsf{N_R}$, then $\mathsf{role}(u) \in \mathsf{N_R}$. Thus, $\mathsf{src}(u) = v$, $\mathsf{tgt}(u) = w$, and $\mathsf{role}(u) = s$.

   Furthermore, if u is an element of the path, then p must pass through v and w. However, w is a fresh element in $\Delta_V^{\mathcal{I}}$ and there is no $u' \in \Delta_E^{\mathcal{I}}$ such that $\mathsf{src}(u') = w$. Hence, w can not be part of any path $p = u_1 u_2 \ldots u_n, n > 0$ such that $\mathsf{src}(u_1) = \mathsf{tgt}(u_n)$ and $\mathsf{roles}(p) \subseteq \xi$ for a $\xi \in \Xi$. Therefore, $\mathcal{I}_{i+1}$ is Ξ-acyclic.

3. If $s \sqsubseteq t \in \mathcal{T}$, $(v, w) \in s^{\mathcal{I}}$, then add a fresh element u to $\Delta_E^{\mathcal{I}_{i+1}}$. If both s and t are role names or inverse roles, then define $\mathsf{src}(u) \coloneqq v$, $\mathsf{tgt}(u) \coloneqq w$, and $\mathsf{role}(u) \coloneqq t$. Otherwise, define $\mathsf{src}(u) \coloneqq w$, $\mathsf{tgt}(u) \coloneqq v$, and $\mathsf{role}(u) \coloneqq t$ if the inverse role is s, otherwise $\mathsf{role}(u) \coloneqq r$ for $r^- = t$.

   Assume that this rule creates a path $p = u_1 u_2 \ldots u_n, n > 0$ such that $\mathsf{src}(u_1) = \mathsf{tgt}(u_n)$ and $\mathsf{roles}(p) \subseteq \xi$ for a $\xi \in \Xi$. If $\mathcal{I}_i$ is Ξ-acyclic, then the addition of u to $\mathcal{I}_i$ has created a cycle in $\mathcal{I}_{i+1}$. It follows that u is an element of the path p. Therefore, $\mathsf{role}(u) \in \mathsf{roles}(p)$. If u is an element of p, $\mathsf{role}(u) \in \mathsf{roles}(p)$ and $\mathsf{roles}(p) \subseteq \xi$, then $\mathsf{role}(u) \in \xi$. By definition, $\Xi \subseteq 2^{\mathsf{N_R}}$. Hence, $\xi \subseteq \mathsf{N_R}$. If $\mathsf{role}(u) \in \xi$ and $\xi \subseteq \mathsf{N_R}$, then $\mathsf{role}(u) \in \mathsf{N_R}$.

   If $(v, w) \in s^{\mathcal{I}}$, then there must exist a $u' \in \Delta_E^{\mathcal{I}_i}$ such that $\mathsf{src}(u') = v$, $\mathsf{tgt}(u') = w$, and $\mathsf{role}(u') = s$. From the Ξ-acyclicity of $\mathcal{T}$ it follows that if $\mathsf{role}(u) \in \xi$, then

$\mathsf{role}(u') \in \xi$. Therefore, $\mathcal{I}_i$ can not be Ξ-acylic, since there must exist a path $q = u_1 u_2 \dots u_n, n > 0$ in $\mathcal{I}^i$ such that $\mathsf{src}(u_1) = \mathsf{tgt}(u_n)$ and $\mathsf{roles}(p) \subseteq \xi$ for a $\xi \in \Xi$, where u' is a part of p. This contradicts our initial assumption that $\mathcal{I}^i$ is Ξ-acyclic. Therefore, $\mathcal{I}_{i+1}$ is Ξ-acyclic.

□

3.2 Query Language

The query language for our framework is a modification of conjunctive regular path queries as defined by Bienvenu et al. [BOS15]. Most notably, we only allow arbitrary length expressions over elements of Ξ if the ABox $\mathcal{A}$ is Ξ-acyclic and the TBox $\mathcal{T}$ is Ξ-compliant. Moreover, the application of Kleene stars is restricted to unions of the sets of roles in Ξ.

Definition 3.2.1 (Ξ-Restricted C2RPQs)**.** Let Ξ be a role clustering. A Ξ-*restricted C2RPQ* has the form $q(\vec{x}) \leftarrow \exists \vec{y}.\phi$ where $\vec{x}$ and $\vec{y}$ are disjoint tuples of variables, and ϕ is a conjunction of atoms of the forms:

1. $A(t)$ where $A \in \mathsf{N_C}$ and $t \in \mathsf{N_I} \cup \vec{x} \cup \vec{y}$.
2. $(s_1 \cup s_2 \cup \dots s_n)(t, t')$, where $n \geq 1$, $s_i \in \mathsf{N_R^{\pm}}$ and $t, t' \in \mathsf{N_I} \cup \vec{x} \cup \vec{y}$
3. $\rho(t, t')$, where ρ is a concatenation of $r_i \in \xi$, $\xi \in \Xi$ the following forms:

$$(r_1 \cup \dots \cup r_n) \quad \textit{or} \quad (r_1 \cup \dots \cup r_n)^*,$$

$n \geq 1$, $t, t' \in \mathsf{N_I} \cup \vec{x} \cup \vec{y}$

In the remainder of this thesis we write Ξ-*restricted queries* when we refer to Ξ-restricted C2RPQs. For all atoms with two terms, we refer to them as *binary atoms*. The order of the atoms in the body of the query has no influence on the semantics. Hence, we can view the body of a query as a *set* of atoms. Moreover, we will also consider unions of Ξ-restricted queries as sets of queries.

Our query language is strictly more expressive than conjunctive queries, which are covered by 1. and 2. in our definition. Moreover, we allow for disjunction of arbitrary role names and their inverses. They allow for more succinct rewritings, as we will show in the following chapter.

We allow for *path* atoms in the last point of the description of our query language. Furthermore, they are a special type of binary atoms we denote as *arbitrary length atoms*, because they can describe paths that can be of length zero or more. Still, arbitrary length atoms are subject to restrictions on the ABox and TBox. We do note however, that Ξ-acyclicity of ABoxes and Ξ-compliance are strictly syntactic restrictions.

We have shown previously in Lemma 2.4.1 that nre-matches are not preserved under homomorphisms. Therefore, certain answer semantics do not seem adequate for nre-semantics. Nevertheless, we can still use certain answers for homomorphism-based semantics. If we restrict our queries to Ξ-acyclic KBs, then we get the same answers under h-semantics and nre-semantics.

Lemma 3.2.1. *For any interpretation $\mathcal{I}$ that is Ξ-acyclic and any Ξ-restricted C2RPQ $q(\vec{x})$, it holds that π is an h-match iff π is an nre-match.*

Proof. Let Ξ be an arbitrary role clustering, $\mathcal{I}$ a Ξ-acyclic interpretation, and q a Ξ-restricted CRPQ. If q is a CQ without a regular path expression, then any match π is an h-match iff π is an nre-match because the definitions of the matches are identical in this case. If q contains a regular path expression, then the definition of h-matches and nre-matches only differ in the definition of mappings for regular expressions $\rho(t, t')$.
If-direction: Let π be an arbitrary nre-match. Then, π is also an h-match because $L^{\mathcal{I}}_{nre} \subseteq L^{\mathcal{I}}_{h}$.
Only-If-direction: Let π be an arbitrary h-match and $\rho(t, t')$ a path expression in q. Since π is an h-match, we know that $(\pi(t), \pi(t')) \in L^{\mathcal{I}}_{h}$. Hence, there exists a path $p = u_1 \dots u_n, n > 0$ in $\mathcal{I}$ such that $\lambda(p) \in L(\rho)$. However, we also know that $\mathcal{I}$ is Ξ-acyclic and that ρ is a regular expression using role names from one $\xi \in \Xi$. It follows that for every $i \neq j$, $u_i \neq u_j$ holds because there is no path $p' = u'_1 u'_2 \dots u'_m, m > 0$ in $\mathcal{I}$ such that $\mathsf{src}(u'_1) = \mathsf{tgt}(u'_m)$ and $\mathsf{roles}(p) \subseteq \xi$. Therefore, $(\pi(t), \pi(t')) \in L^{\mathcal{I}}_{nre}$. Finally, we have that π is an nre-match. □

As an extension of Lemma 3.2.1, our framework ensures that for the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, the h-answers coincide with the nre-answers if the ABox $\mathcal{A}$ and the TBox $\mathcal{T}$ are Ξ-acyclic, resp. Ξ-compliant.

Lemma 3.2.2. *Let Ξ be a role clustering, $\mathcal{A}$ a Ξ-acyclic ABox, $\mathcal{T}$ a Ξ-compliant TBox and q a Ξ-restricted C2RPQ. Then, $\mathsf{ans}_{\mathsf{h}}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}}) = \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$.*

Proof. From Lemma 3.1.1 it follows that $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ is Ξ-acyclic. Because $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ is a Ξ-acyclic interpretation, and q is a Ξ-restricted CRPQ, any match π is an h-match iff π is an nre-match by Lemma 3.2.1. Hence, $\mathsf{ans}_{\mathsf{h}}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}}) = \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. □

As a consequence of Lemma 3.2.2, it follows that $\mathsf{cert}_{h}(q, \mathcal{T}, \mathcal{A}) = \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. In other words, any certain answer to q under h-semantics is an nre-answer in the canonical model. Because nre-answers are not preserved under homomorphisms, we deem it adequate to define the nre-answers as those tuples that have an nre-match in the canonical model.

Definition 3.2.2 (No-repeated-Edges Certain Answers)**.** A tuple $(a_1, \dots, a_n) \in \mathsf{N_I}$ is a *certain nre-answer* to a query $q(\vec{x}) = \exists \vec{y} \phi$ in a KB $(\mathcal{T}, \mathcal{A})$ if $(a_1, \dots, a_n) \in \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$.

This is not a rare occurrence: Alternative semantics to queries have been proposed in multiple domains. Most notably, in the case of queries with aggregates, the certain answers can be empty under standard open-world semantics [CKNT08]. As an example, the query from the proof of Lemma 2.4.1 has (a) as an answer in our framework.

The coincidence of answers in the canonical model for both semantics has another consequence we wish to mention. Because our path atoms are quite simple, we can break up the concatenation into single binary atoms with fresh variables. For example, a path expression like $r(s \cup t)^* r(x, y)$ is rewritten as $r(x, z_1), (s \cup t)^*(z_1, z_2), r(z_2, y)$ with z_i as fresh variables in q. It is easy to see that under h-semantics, both expressions have the same matches for x and y. Not only will we make use of this fact in rewritings, but it also means that our arbitrary length atoms are all of the form $(r_1 \cup r_2 \cup \ldots r_n)^*(x, y)$, which are tractable in data complexity under nre-semantics (and h-semantics).

Finally, we must discuss the usability of navigational atoms in our query language. At first glance, it might seem restrictive that path atoms can only be directed. However, we also note that paths which include inverse roles can also be included by "splitting up" the path atom. With conjunctive queries a path like $rs^- t^*(x, y)$ can be simulated by writing $r(x, z_1), s^-(z_1, z_2), t^*(z_2, y)$, even if the roles r, s and t form a cycle in $\mathcal{A}$. Still, this comes at the cost that the semantics of the two expressions are not equivalent, as the second expression can potentially have more matches. On the other hand, Cypher also does not allow for using inverses in arbitrary length expressions. In addition, navigational features are often used in simple ways such as $r_1 \cdot r_2^*$ or r^*, as was shown from the analysis of SPARQL query logs [BMT17, BMT19]. Moreover, Martens et al. [MNT20] report that 99.8% of the RPQs contained in the SPARQL query logs obtained by Bonifati et al. [BMT17] were single-occurrence RPQs. While not all single-occurrence RPQs can be captured in our query language, we can still cover a wide range of them.

CHAPTER 4

Ontology-Mediated Querying by Rewriting

The success of OMQ systems relies on the efficient rewriting of queries posed to the knowledge base $\mathcal{K} = (\mathcal{T}, \mathcal{A})$ as a query over the source. Intuitively, a rewriting injects our domain knowledge expressed in the TBox $\mathcal{T}$ into the query. This way, we can make use of the optimization techniques employed by the database management system (DBMS) for query answering.

We present a simple rewriting technique based on the PerfectRef algorithm for DL-Lite [CGL+07]. The rewriting algorithm transforms a query in our query language to a union of queries in our query language, which can also be expressed in Cypher. We also prove that the rewriting is correct for nre-semantics and Ξ-compliant TBoxes. Moreover, the Cypher rewritings return the correct answers when they are executed over the plain data. Finally, we analyze the computational complexity of query rewriting.

4.1 A Naïve Rewriting

For Ξ-restricted queries, we extend the PerfectRef algorithm [CGL+07] for path expressions and atoms with inverse roles. We present our rewriting algorithm Rewrite in Algorithm 4.1. Steps (a) and (b) in the algorithm are identical to the original formulation of the PerfectRef algorithm. However, to account for atoms with disjunction and arbitrary length atoms, we need to make some adjustments to the `Replace` function, which is also present in PerfectRef. Furthermore, the functions `SaturatePaths`, `Concatenate`, `Merge` and `Drop` are new additions to the algorithm.

The function τ is marks all occurrences of unbound variables as such. In the following, we will use '_' to denote unbound variables. However, we assume that we record a unique

name for each unbound variable. We need unique names for variables in the rewriting of arbitrary length atoms.

An arbitrary length expression can only contain role names from $\mathsf{N_R}$. Note that for a Ξ-compliant TBox, there is no axiom in $\mathcal{T}$ such that a rewriting could introduce an inverse role in an arbitrary length expression. Moreover, the answers to Ξ-restricted queries with path atoms coincide under both nre- and h-semantics only if the TBox $\mathcal{T}$ is Ξ-compliant and the ABox $\mathcal{A}$ is Ξ-acyclic. Hence, our rewritings are only applicable if the TBox is Ξ-compliant, and only returns the correct answers for queries with path expressions if the ABox is also Ξ-acylic. We also wish to mention that our query language allows inverse roles for binary atoms that are not of arbitrary length. For any atom $(s_1 \cup \cdots \cup s_n)(x, y)$, the atom $(s_1^- \cup \cdots \cup s_n^-)(y, x)$ is semantically equivalent. As a result, we consider them to be equivalent when we describe a query as a *set* of atoms. The rules we describe in this section are purely on a *syntactic* level i.e. there is no *reasoning* involved. Hence, for any operation that includes atoms with inverse roles, we consider both "versions" of the atom.

4.1.1 Path Saturation

We begin our rewriting of Ξ-restricted queries by first *splitting up* path atoms into single atoms with fresh variables. For example, a path expression like $r(s \cup t)^* r(x, y)$ is rewritten as $r(x, z_1), (s \cup t)^*(z_1, z_2), r(z_2, y)$ with z_i as fresh variables in q. As a result, all the atoms in the query are of the following three forms:

- $A(x)$
- $(s_1 \cup \cdots \cup s_n)(x, y)$
- $(r_1 \cup \cdots \cup r_n)^*(x, y)$

Note that we have "eliminated" concatenations of relations from our query. We then *saturate* the roles occurring in each binary atom.

Definition 4.1.1 (Roles of Binary Atoms)**.** Let $\alpha = \rho(x, y)$ be a binary atom and $s_1, \ldots s_n$ the roles from $\mathsf{N_R}^{\pm}$ occurring in ρ. Then, $\mathsf{roles}(\alpha) = \{s_1, \ldots, s_n\}$.

Definition 4.1.2 (Saturation of Binary Atoms)**.** Let $\alpha = \rho(x, y)$ be a binary atom. Then, the *saturation* S of $\mathsf{roles}(\alpha)$ w.r.t. a TBox $\mathcal{T}$ is inductively defined in the following way:

- If $s \in \mathsf{roles}(\alpha)$, then $s \in S$.
- If $s \in S$, and $s_1 \sqsubseteq s \in \mathcal{T}$, then $s_1 \in S$.
- If $s \in S$, and $s_1 \sqsubseteq s^- \in \mathcal{T}$, then $s_1^- \in S$.

The *saturation* of α is then $(s_1 \cup \cdots \cup s_n)^*(x, y)$ for $s_i \in S$ if α was an arbitrary length atom, and $(s_1 \cup \cdots \cup s_n)(x, y)$ otherwise.

Algorithm 4.1: Rewrite

Input: A Ξ-restricted query q, and a Ξ-compliant TBox $\mathcal{T}$
Output: Q a set of Ξ-restricted queries

$Q \leftarrow \{\tau(\texttt{SaturatePaths}(q, \mathcal{T}))\}$;
$Q' \leftarrow \varnothing$;
while $Q \neq Q'$ **do**
  $Q' \leftarrow Q$;
  foreach $q \in Q'$ **do**
    (a) **foreach** *atom* $\alpha \in q$ **do**
      foreach *PI* $I \in \mathcal{T}$ **do**
        if I *is applicable to* α **then**
          $Q \leftarrow Q \cup \{\tau(\texttt{Replace}(q, \alpha, I))\}$;
      end
    end
    (b) **foreach** *pair of atoms* $\alpha_1, \alpha_2 \in q$ **do**
      if α_1 *and* α_2 *unify* **then**
        $Q \leftarrow Q \cup \{\tau(\texttt{Reduce}(q, \alpha_1, \alpha_2))\}$;
    end
    (c) **foreach** *pair of path atoms* $\alpha_1, \alpha_2 \in q$ **do**
      if α_1 *and* α_2 *can be concatenated* **then**
        $Q \leftarrow Q \cup \{\tau(\texttt{Concatenate}(q, \alpha_1, \alpha_2))\}$;
    end
    (d) **foreach** *pair of path atoms* $\alpha_1, \alpha_2 \in q$ **do**
      if α_1 *and* α_2 *can be merged* **then**
        $Q \leftarrow Q \cup \{\tau(\texttt{Merge}(q, \alpha_1, \alpha_2))\}$;
    end
    (e) **foreach** *atom* $\alpha \in q$ **do**
      if α *is an arbitrary-length path atom with an unbound variable* **then**
        $Q \leftarrow Q \cup \{\tau(\texttt{Drop}(q, \alpha))\}$
    end
  end
end
return Q;

Note that s and s_1 can be role names or inverse roles, but in the case of arbitrary length expressions they are only role names because of the Ξ-compliance of $\mathcal{T}$. As an example, if our arbitrary length atom is $(r \cup s)^*(x, y)$ and $t \sqsubseteq r \in \mathcal{T}$, then we write $(r \cup s \cup t)^*(x, y)$. Both the path splitting operation and the saturation of the roles of the binary atoms is done by `SaturatePaths`, shown in Algorithm 4.2. In the main body, the binary atoms are split and each new atom that was obtained by splitting is saturated. Unary atoms remain unchanged in `SaturatePaths`. The function `Saturate` performs saturation as described in Definition 4.1.2, with a binary atom and a TBox as an input.

Algorithm 4.2: SaturatePaths

Input: A Ξ-restricted query q, and a Ξ-compliant TBox $\mathcal{T}$
Output: q' a Ξ-restricted query

```
1  q' ← ∅;
2  foreach atom α ∈ q do
3      if α is a binary atom of the form ρ_1 ... ρ_n(x, y) then
4          left ← x;
5          foreach concatenation element ρ_i do
6              if ρ_i = ρ_n then
7                  right ← y;
8              else
9                  right ← fresh variable not occurring in q or q';
10             q' ← q' ∪ Saturate(ρ_i(left, right), T);
11         end
12     else
13         q' ← q' ∪ α;
14 end
15 return q';
```

4.1.2 Replacement of Query Atoms

The main idea behind *replacing* an atom α with an atom α' in a query q is to generate a query q' for which a match for q' implies that there is a match for q. In essence, it could be that the match for α was added to the model of the knowledge base to satisfy an axiom in the TBox $\mathcal{T}$.

Example 4.1.1 (Replacement of Query Atoms)**.** Let $q(x) \leftarrow A(x)$ be a query and $\mathcal{K} = (\{A_1 \sqsubseteq A\}, \{A_1(a)\})$. Then, for any interpretation $\mathcal{I}$ to be a model, $\mathcal{I} \vDash A(a)$. Therefore, (a) must be a certain answer to the query q.

Example 4.1.1 shows that in our rewriting algorithm, we should add a query $q(x) \leftarrow A_1(x)$ to our set of queries. We present the replacement rules for each possible combination of query atom and concept inclusion in Table 4.1. We say that a concept inclusion or axiom is *applicable* to a query atom if they appear in Table 4.1. In Algorithm 4.1, applying

the replacement of atoms is done by the function `Replace`. Note that arbitrary length atoms can not be replaced.

Axiom	Query Atom	Rewriting
$A_1 \sqsubseteq A$	$A(x)$	$A_1(x)$
$\exists s \sqsubseteq A$	$A(x)$	$\texttt{Saturate}(s(x,\_),\mathcal{T})$
$A \sqsubseteq \exists s$	$(s \cup \ldots)(x,\_)$	$A(x)$
$\exists s_1 \sqsubseteq \exists s$	$(s \cup \ldots)(x,\_)$	$\texttt{Saturate}(s_1(x,\_),\mathcal{T})$
$A \sqsubseteq \exists s^-$	$(s \cup \ldots)(\_,x)$	$A(x)$
$\exists s_1 \sqsubseteq \exists s^-$	$(s \cup \ldots)(\_,x)$	$\texttt{Saturate}(s_1(x,\_),\mathcal{T})$

Table 4.1: PerfectRef rewritings with disjunctions, "_" denote unbound variables

4.1.3 Unification

If two atoms α_1 and α_2 in a query q *unify*, then the result is a new query q' where the most general unifier of α_1 and α_2 has been applied to q. As is the case for PerfectRef, each occurrence of '_' must be considered a separate variable. Moreover, the most general unifier replaces each occurrence of '_' with the corresponding variable in the other atom. Example 4.1.2 shows the unification for two binary atoms with unbound variables.

Example 4.1.2 (Unification of Atoms)**.** Let $q(x) \leftarrow r(x,\_), r(\_,y)$ be a query. Then, the unification of the atoms $r(x,\_)$ and $r(\_,y)$ results in the query $q(x) \leftarrow r(x,y)$.

If the corresponding variable is also unbound, the result is also '_'. In Algorithm 4.1, unification of atoms in q is performed by `Reduce`.

4.1.4 Concatenation of Arbitrary Length Atoms

Regarding rules for arbitrary length atoms, there are a number of cases we have to consider. First, there are cases where we can rewrite an arbitrary length atoms by appending it to the start or the end of the other atom i.e., performing a *concatenation*.

Example 4.1.3 (Concatenation of Arbitrary Length Atoms)**.** Consider the query $q(x) \leftarrow (r \cup s \cup t)^*(x,y), r(x,z)$. Then, the query $q(x) \leftarrow (r \cup s \cup t)^*(z,y), r(x,z)$ is the result of concatenating the second atom to the start of the first atom.

Note that by applying this operation, the variable z turned from an unbound variable to a bound variable. To add, the atom $r(x,z)$ has remained unchanged. We describe this rewriting rule in the following definition.

Definition 4.1.3 (Concatenation of Arbitrary Length Atoms)**.** Let α_1 be a binary atom and α_2 an arbitrary length atom such that $\mathsf{roles}(\alpha_1) \subseteq \mathsf{roles}(\alpha_2)$. There are two cases:

- Assume $\alpha_1 = \rho(x, z)$ and $\alpha_2 = \rho^*(x, y)$. Then, the rewriting of α_2 is $\rho^*(z, y)$. The *concatenation* of α_1 and α_2 is then $\rho(x, z), \rho^*(z, y)$.

- Assume $\alpha_1 = \rho(x, y)$ and $\alpha_2 = \rho^*(z, y)$. Then, the rewriting of α_2 is $\rho^*(z, x)$. The *concatenation* of α_1 and α_2 is then $\rho(x, y), \rho^*(z, x)$.

For clarification, we want to mention that there is no way to concatenate a binary atom with inverse roles to an arbitrary length atom. By definition, the roles occurring in an arbitrary length atom can not be inverse roles for a Ξ-restricted query and a Ξ-compliant TBox $\mathcal{T}$. Therefore, if a binary atom contains role names *and* inverse roles, the condition that the roles of the atom are a subset of the arbitrary length atom it should be concatenated to can never be fulfilled. However, if the binary atom contains *only* inverse roles, then we have to consider the semantically equivalent "inverse" version of the atom. In Algorithm 4.1, this operation is done by `Concatenate`.

4.1.5 Merging of Binary Atoms

Second, we add a rule for *merging* binary atoms. In a nutshell, merging of binary atoms is akin to finding the most common denominator of two binary atoms.

Example 4.1.4 (Merging of Binary Atoms)**.** Let $q(x, y) \leftarrow (r \cup r_1 \cup r_2)(x, \_), (r \cup r_3 \cup r_4)^*(\_, y)$. If we merge the two atoms in the query, we get $q(x, y) \leftarrow r(x, y)$.

Definition 4.1.4 (Merging of Binary Atoms)**.** Let α_1 and α_2 be binary atoms such that $\mathsf{roles}(\alpha_1) \cap \mathsf{roles}(\alpha_2) \neq \varnothing$ and the terms of α_1 and α_2 are unifiable. We define the binary atoms α_1' and α_2' as the result of replacing the roles of α_1 and α_2 respectively with $\mathsf{roles}(\alpha_1) \cap \mathsf{roles}(\alpha_2)$. Moreover, if either of α_1 or α_2 is not an arbitrary length atom, then both α_1' and α_2' are not arbitrary length atoms. Then, the query q' obtained as the result of *merging* α_1 and α_2 is obtained by replacing them with α_1' and α_2' in q, and applying the most general unifier of α_1' and α_2' to q.

We want to mention that because the roles in both α_1 and α_2 are saturated, the intersection of the roles of the atoms is also saturated. In other words, the roles of the merged atom is closed under the subrole hierarchy. We must address one small issue with the definition of merging. For atoms with inverse roles, it can happen that there are two ways of merging the atoms. Therefore, we have to consider the inverses of binary atoms that are not of arbitrary length.

Example 4.1.5 (Merging of Binary Atoms with Inverses)**.** Let $q() \leftarrow (r \cup s^-)(x, y), (r \cup s)(x, z)$. Then, there are two possible ways of merging the binary atoms in q: $q() \leftarrow r(x, y)$ and $q() \leftarrow s(x, x)$.

4.1.6 Dropping Arbitrary Length Atoms

Finally, we *drop* arbitrary length path atoms from the query if one of the variables in the expression is unbound. As the name implies, the function `Drop` applies this operation to q.

For the new queries that are created in the reformulation of the initial query, the rules for reformulation can again be applied. We illustrate this by a simple example.

Example 4.1.6. Let $\mathcal{T} = \{A \sqsubseteq \exists r^-, \exists r \sqsubseteq \exists s^-, \exists s \sqsubseteq \exists t^-\}$ and $\mathcal{A} = \{a : A\}$. Assume we wish to rewrite the Ξ-restricted query $q() \leftarrow ts^*r(x, y)$ over the KB $\mathcal{K} = (\mathcal{T}, \mathcal{A})$. We go over the execution of Rewrite one step at a time. First, we rewrite the query as

$$q() \leftarrow t(\_, z_1), s^*(z_1, z_2), r(z_2, \_)$$

Then, the procedure enters the main loop. In the first step of the main loop, the axiom $\exists s \sqsubseteq \exists t^-$ is applicable. Hence, the query

$$q() \leftarrow s(z_1, \_), s^*(z_1, z_2), r(z_2, \_)$$

is added to the output set queries. This rule application makes merging of the first two binary atoms in the query possible, therefore adding

$$q() \leftarrow s(\_, z_2), r(z_2, \_)$$

to the set of queries to rewrite. However, we can also perform concatenation of $s(z_1, \_)$ and $s^*(z_1, z_2)$. As a consequence, the query

$$q() \leftarrow s(\_, x), s^*(x, z_2), r(z_2, \_)$$

is also a result of applying a rewriting to the second query. Note that x has turned from an unbound variable to a bound variable, and vice-versa for z_1. Now, we can replace the atom $s(\_, x)$ by $r(x, \_)$ because the axiom $\exists r \sqsubseteq \exists s^-$ is applicable. Thus, the query

$$q() \leftarrow r(x, \_), s^*(x, z_2), r(z_2, \_)$$

will be added after one loop of the procedure. The atoms $r(x, \_)$ and $r(z_2, \_)$ can be unified, which leads to

$$q() \leftarrow r(x, \_), s^*(x, x)$$

to be added to the set of queries. In one of the previous loops of the algorithm, the procedure will have added the query

$$q() \leftarrow r(x, \_), s(x, x)$$

because the binary atoms $s(\_, x)$ and $s^*(x, z_2)$ from the fourth query can be merged. Furthermore, the query

$$q() \leftarrow r(\_, \_)$$

was also added in an iteration of the main loop because the axiom $\exists r \sqsubseteq \exists s^-$ is applicable to $s(\_, z_2)$ from the third query. Finally, the last query which is in the output of Rewrite is

$$q() \leftarrow A(\_)$$

as a result of applying the axiom $A \sqsubseteq \exists r^-$ to the previous query. In essence, we unravel the path in the implied part of the canonical model in the query. This small example already shows that the unraveling of paths creates many different rewritings, all of which but $A(\_)$ return an empty result.

The following example highlights the use of path unification and rewriting with other atoms in a Ξ-restricted query with multiple atoms.

Example 4.1.7. Let $\mathcal{T} = \{C \sqsubseteq \exists t, \exists t^- \sqsubseteq \exists s, \exists s^- \sqsubseteq B, s \sqsubseteq r, t \sqsubseteq r\}$. We want to rewrite the Ξ-restricted query $q() \leftarrow A(x), r^*(x, y), B(y)$ with $\mathcal{T}$. Similar to the previous example, we will discuss the rewriting step-by-step. First, we rewrite the query as

$$q() \leftarrow A(x), (r \cup s \cup t)^*(x, y), B(y)$$

by saturating the arbitrary length atom in the input query with regard to the ontology. In the first iteration of the main loop, only the replacement of $B(y)$ is done by the algorithm. The axiom $\exists s^- \sqsubseteq B$ is applicable, hence

$$q() \leftarrow A(x), (r \cup s \cup t)^*(x, y), s(\_, y)$$

is added to the set of queries. On this new query, both merging and concatenation can be performed. We first show the result of concatenating the binary atoms, which generates the query

$$q() \leftarrow A(x), (r \cup s \cup t)^*(x, z), s(z, \_)$$

that is added to the output. The result of merging is the query

$$q() \leftarrow A(x), s(x, \_)$$

on which we can once again apply an axiom. This time, the axiom in question is $\exists t^- \sqsubseteq \exists s$, which leads to

$$q() \leftarrow A(x), t(\_, x)$$

being added to our set of queries. Furthermore, merging is applicable to the third query, therefore

$$q() \leftarrow A(x), s(x, x)$$

is added to the rewriting. However, we can not apply any axioms to this query. Still, we have a query in our set of queries on which an axiom can be applied. More specifically, the axiom $\exists t^- \sqsubseteq \exists s$ can be applied to the third query. Hence, we can add

$$q() \leftarrow A(x), (r \cup s \cup t)^*(x, z), t(\_, z)$$

to our set of queries. Similar to before, we can perform both merging and concatenation on this query. The result of concatenation is

$$q() \leftarrow A(x), (r \cup s \cup t)^*(x, z), t(z, \_)$$

and the result of merging is

$$q() \leftarrow A(x), t(x, \_)$$

on which we can both finally apply the axiom $C \sqsubseteq \exists t$. If we apply the axiom to the first of these two queries, we get that

$$q() \leftarrow A(x), (r \cup s \cup t)^*(x, z), C(z)$$

is a part of our rewritten query. Applying the axiom to the second of these two queries yields

$$q() \leftarrow A(x), C(x)$$

to be added to our rewriting. Only one more rule can be used for rewriting, which is merging the atoms in the last query we generated from concatenation. Finally, the query

$$q() \leftarrow A(x), t(x, x)$$

is generated and the rewriting terminates.

4.2 Termination of Rewriting

A fundamental property for our algorithm Rewrite to be viable for query answering is that it must terminate for a Ξ-restricted query q and a Ξ-compliant TBox $\mathcal{T}$. Our argument is based on the proof of termination for PerfectRef [CGL+07].

Lemma 4.2.1 (Termination of Rewrite)**.** *Let q be a Ξ-restricted query, and $\mathcal{T}$ a Ξ-compliant TBox. Then, the algorithm* Rewrite$(q, \mathcal{T})$ *terminates.*

Proof. The first step of the algorithm, `SaturatePaths`, returns a query with n atoms. It is easy to see that this first step terminates. We base our proof for termination of Rewrite on the following observations:

1. For any of the steps (a)-(e) in the procedure, the number of atoms is bounded by n. In other words, no step adds an atom to the query. `Replace` and `Concatenate` replace an atom in the query by another atom. The steps `Reduce`, `Merge`, and `Drop` either combine two atoms into one atom, or remove an arbitrary length atom.

2. The number of distinct terms occurring in the queries generated by Rewrite is also bounded by n. More specifically, it is bounded by the number of terms in the query generated by `SaturatePaths` plus the symbol '_' used for unbound variables. Fresh unbound variables can only be introduced by `Replace` in binary atoms.

Because we consider binary atoms to be equal (on a syntactic level) if they have the same sets of roles and the same terms, the atom created by `Replace` can only be added once to each query where the atom in question has not been replaced yet. Even if concatenation can turn an unbound variable into a bound one, we record a unique name for each one of them. As a consequence, the concatenation operation can be executed only once for each binary atom that contains an unbound variable that may become bound this way.

3. If k is the number of times a fresh unbound variable can be introduced into the query, then the number of distinct atoms generated by Rewrite is bounded by $m \cdot (n+k)^2$, where m is the number of role names and concept names.

4. We record the queries generated by Rewrite in a set. Queries are only added to this set, and never removed.

The first three points imply that the number of queries that Rewrite can generate is bounded. The last point dictates that no query is generated twice by the algorithm. Therefore, the algorithm is bound to terminate. For each possible replacement of a query atom a new query is generated. Hence, every combination of replacements is generated by Rewrite, and the number of distinct queries is exponential in the number of atoms contained in the query. □

4.3 Correctness of Rewriting

We now show that Rewrite is a complete and correct rewriting for Ξ-restricted queries q over Ξ-compliant TBoxes and Ξ-acyclic ABoxes in nre-semantics. In nre-semantics, we defined the certain answers of queries with regard to ontologies as the answers in the canonical model. However, the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ may be infinite. Still, we can show that Rewrite can add all implied knowledge into the rewriting of an input query. In turn, the rewritten query can be evaluated over the property graph representation of the ABox.

For this, we have to show that the union of answers obtained from evaluating each query $q' \in \mathsf{Rewrite}(q,\mathcal{T})$ over the database $\mathsf{db}(\mathcal{A})$ are exactly the certain answers of q over the knowledge base $(\mathcal{T},\mathcal{A})$ i.e., $\bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})) = \mathsf{cert}_{nre}(q, (\mathcal{T},\mathcal{A}))$. Therefore, showing that $\mathsf{Rewrite}(q,\mathcal{T})$ is a complete and correct rewriting is equivalent to proving $\bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})) = \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. Our proof will consist of two Lemmas, one for showing completeness and one for showing correctness.

Lemma 4.3.1 (Correctness of Rewriting)**.** *Let q be a Ξ-restricted query, $\mathcal{T}$ a Ξ-compliant TBox, and $\mathcal{A}$ a Ξ-acyclic ABox such that $(\mathcal{T},\mathcal{A})$ is satisfiable. Then,*

$$\bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})) \subseteq \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}}).$$

Proof. In order to prove our claim, it suffices to show $\mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})) \subseteq \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$ holds for any $q' \in \mathsf{Rewrite}(q, \mathcal{T})$. Moreover, this is an immediate consequence of showing that $\mathsf{ans}_{nre}(q', \mathcal{I}_{\mathcal{T},\mathcal{A}}) \subseteq \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$.

We start our proof with the first step of the algorithm, which is reformulating all path expressions in q. `SaturatePaths` first splits up path atoms into single atoms with fresh variables. Let q' be the query obtained from q in this step, and $\vec{a} \in \mathsf{ans}_{nre}(q', \mathcal{I}_{\mathcal{T},\mathcal{A}})$ an answer to q' in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. We assumed that $\mathcal{A}$ is Ξ-acyclic and $\mathcal{T}$ is Ξ-compliant. Hence, $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ is also Ξ-acyclic (Lemma 3.1.1). It follows that there are no cycles using the roles in Ξ. Thus, any nre-match that satisfies the new binary atoms in q' also satisfies the binary atoms in q. Hence, $\vec{a}$ must also be an element of $\mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. Note that if $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ were not Ξ-acyclic, then splitting up the path atoms could add answers where path atoms do not have an nre-match (cf. the example from Lemma 2.4.1).

Then, all binary atoms are saturated. Let q'' be the query obtained from q' in this step, and $\vec{a} \in \mathsf{ans}_{nre}(q'', \mathcal{I}_{\mathcal{T},\mathcal{A}})$ an answer to q'' in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. From the Ξ-compliance of $\mathcal{T}$ it follows that if $r \in \mathsf{N_R}$ occurs in an arbitrary length atom, there is no $s^- \sqsubseteq r$ or $r \sqsubseteq s^-$ in $\mathcal{T}$. Hence the query q' generated in this step only uses $r \in \mathsf{N_R}$ for arbitrary length atoms, and is still a Ξ-restricted query. It is easy to see that $\vec{a} \in \mathsf{ans}_{nre}(q', \mathcal{I}_{\mathcal{T},\mathcal{A}})$. Therefore, for the query q'' obtained from $\mathtt{SaturatePaths}(q, \mathcal{T})$, we have that $\mathsf{ans}_{nre}(q'', \mathcal{I}_{\mathcal{T},\mathcal{A}}) \subseteq \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$.

We now move on to the main body of the Rewrite algorithm, for which we will show that for any query q_{i+1} obtained from q_i obtained in any of the steps, $\mathsf{ans}_{nre}(q_{i+1}, \mathcal{I}_{\mathcal{T},\mathcal{A}}) \subseteq \mathsf{ans}_{nre}(q_i, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. Let $\vec{a} \in \mathsf{ans}_{nre}(q_{i+1}, \mathcal{I}_{\mathcal{T},\mathcal{A}})$ be an answer to q_{i+1} in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. Assume that q_{i+1} was obtained from any of the following steps in the main body of the algorithm:

(a) Assume that q_{i+1} was obtained from q_i by applying one of the rules in Table 4.1 by $\mathtt{Replace}(q, \alpha, I)$. Let I be the concept inclusion $A_1 \sqsubseteq A$ that was used in this step i.e., q_{i+1} was obtained from q_i by replacing the atom $A(x)$ with $A_1(x)$. If $\vec{a}$ is a match for q_{i+1}, then there must exist a $v \in \Delta_V^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$ such that $\pi(x) = v$ and $v \in A_1^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$. From the definition of the construction of $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ it follows that the first rule is applicable such that $v \in A^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$. Hence, $\vec{a} \in \mathsf{ans}_{nre}(q_i, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. The proofs for the other cases in Table 4.1 are analogous.

(b) Assume that $q_{i+1} = \tau(\mathtt{Reduce}(q_i, \alpha_1, \alpha_2))$, where α_1 and α_2 are atoms that unify. Let σ be the most general unifier of α_1 and α_2 and π' a match for $\vec{a}$ for q_{i+1} in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. We can construct a mapping $\pi(t)$ for all terms t in q_i by setting $\pi(t) = \pi'(t \cdot \sigma)$. Moreover, $\pi(x_i) = \pi'(x_i' \cdot \sigma)$ for any answer variable x_i and x_i' of q and q'. It is easy to check that π is a match for $\vec{a}$ for q_i in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. Thus, $\pi(\vec{x}) = \vec{a}$. Therefore, $\vec{a} \in \mathsf{ans}_{nre}(q_i, \mathcal{I}_{\mathcal{T},\mathcal{A}})$.

(c) Assume that $q_{i+1} = \tau(\mathtt{Concatenate}(q_i, \alpha_1, \alpha_2))$, where α_1 and α_2 are binary atoms that can be concatenated into one path. Let $\alpha_1 = \rho_1(x, z)$ and $\alpha_2 = \rho_2^*(x, y)$ such that $\mathsf{roles}(\rho_1) \subseteq \mathsf{roles}(\rho_2)$ with the result $\rho_1(x, z), \rho_2^*(z, y)$. We can see that a match for $\vec{a}$ for q_{i+1} in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, which must exist because $\vec{a}$ is an answer to q_{i+1}, is

also a match for q_i. It follows that $\vec{a} \in \mathsf{ans}_{nre}(q_i, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. The proof for the other forms of α_1, α_2 are analogous.

(d) Assume that $q_{i+1} = \tau(\texttt{Merge}(q_i, \alpha_1, \alpha_2))$, where α_1 and α_2 are binary atoms that can be merged into one binary atom. Let $\alpha_1 = \rho_1(x, y)$ and $\alpha_2 = \rho_2^*(x', y')$. The result of this operation is a new path atom which describes a path of length one that uses one of the role names described in the intersection of role names of ρ_1 and ρ_2. Moreover, the most general unifier of (x, y) and (x', y'), respectively, has been applied to q. Akin to the case for `Reduce`, we can construct a mapping π from the most general unifier σ. We can see that the obtained mapping is a match for $\vec{a}$ for q_i in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. As a result, $\vec{a} \in \mathsf{ans}_{nre}(q_i, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. The proofs for the other forms of α_1, α_2 are analogous.

(e) Assume that $q_{i+1} = \tau(\texttt{Drop}(q_i, \alpha))$, where α is an arbitrary length path atom that contains an unbound variable. It is easy to check that $\vec{a} \in \mathsf{ans}_{nre}(q_i, \mathcal{I}_{\mathcal{T},\mathcal{A}})$.

We can see that for any query q' obtained from the main body of $\mathsf{Rewrite}$, we have that $\mathsf{ans}_{nre}(q', \mathcal{I}_{\mathcal{T},\mathcal{A}}) \subseteq \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. Because $\mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})) \subseteq \mathsf{ans}_{nre}(q', \mathcal{I}_{\mathcal{T},\mathcal{A}})$ for a Ξ-acyclic ABox $\mathcal{A}$ and a Ξ-compliant TBox $\mathcal{T}$, we have that $\bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})) \subseteq \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. □

Lemma 4.3.2 (Completeness of Rewriting)**.** *Let q be a Ξ-restricted query, $\mathcal{T}$ a Ξ-compliant TBox, and $\mathcal{A}$ a Ξ-acyclic ABox such that $(\mathcal{T}, \mathcal{A})$ is satisfiable. Then,*

$$\mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}}) \subseteq \bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})).$$

Proof. The proof of the completeness of the rewriting follows if for any answer $\vec{a} \in \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$, there exists a query $q' \in \mathsf{Rewrite}(q, \mathcal{T})$ such that $\vec{a} \in \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A}))$. We will show this claim holds by inductively showing that in each rewriting step, we are "moving up" in the part of the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ that is induced by the axioms in $\mathcal{T}$. Note that the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ can be seen as a forest with roots as ABox assertions in $\mathcal{A}$. As such, we can define the depth n of each element $v \in \Delta_V^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$. If the mapping assigns a term to an element of the ABox, then the depth is zero.

Let $\vec{a} \in \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$ and π a mapping. If $\mathsf{depth}(\pi(y)) = 0$ for all variables y in the query, then we are done i.e., all variables in the query are mapped to elements of $\mathsf{Ind}(\mathcal{A})$. Otherwise, there must exist some variable y with $\pi(y) \in \Delta_V^{\mathcal{I}_{\mathcal{T},\mathcal{A}}} \setminus \mathsf{Ind}(\mathcal{A})$ such that there is no variable z in q, and $\pi(z)$ is a child of y. In short, we assume that $\pi(y)$ is the deepest element in a branch of the tree of $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ a query variable is mapped to. We will show that there is $q' \in \mathsf{Rewrite}(q, \mathcal{T})$ such that we can construct a mapping π' for $\vec{a}$ with $\mathsf{depth}(\pi'(y)) < \mathsf{depth}(\pi(y))$. By repeatedly applying this operation, we will eventually obtain a query $q^* \in \mathsf{Rewrite}(q, \mathcal{T})$ such that we can construct a mapping π^* where all query variables have depth zero i.e., are mapped to elements of $\mathsf{Ind}(\mathcal{A})$.

Assume that $\pi(y) = v \in \Delta_V^{\mathcal{I}_{\mathcal{T},\mathcal{A}}} \setminus \mathsf{Ind}(\mathcal{A})$ was added to $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ by the following rule: $w \in \Delta_V^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$ with $w \in A^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$ and $A \sqsubseteq \exists r \in \mathcal{T}$, where $A \in \mathsf{N_C}$ and $r \in \mathsf{N_R}$ (the proof for the claim in the other cases is analogous). Without loss of generality, we assume that q is the query generated from $\texttt{SaturatePaths}(q, \mathcal{T})$. This means that any binary atom is closed under the subrole hierarchy. Note that atoms of the form $(s_1 \cup s_2 \cup \cdots \cup s_n)(y, y')$ can occur in q even if $\pi(y)$ is the deepest element of a branch in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, because the s_i can be inverse roles. However, because we know that v was added to $\Delta_V^{\mathcal{I}_{\mathcal{T},\mathcal{A}}}$ by $A \sqsubseteq \exists r$, any path in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ must reach $\pi(y)$ via an r-relation. Therefore, for any binary atom that includes y and contains inverse roles, we can re-express it with y on the right-hand side. Then, for any such atom if $\rho(x, y) \in q$, then $r \in \mathsf{roles}(\rho)$. For the sake of clarity, we will assume for the remainder of this section that all such binary atoms contain r in the set of roles and y on the right side of the atom. There are two cases: Either y occurs only in arbitrary length atoms, or there is at least one atom of the forms $A(y)$, or $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ in q.

Assume that y only occurs in arbitrary length path atoms and q only contains arbitrary length path atoms. Arbitrary length atoms can not contain any inverse roles if the TBox $\mathcal{T}$ is $\exists$-compliant. Hence, arbitrary length atoms which contain y must be of the form $\rho^*(x, y)$ with $r \in \mathsf{roles}(\rho)$. If there are only arbitrary length atoms in q, then we can construct a mapping π' for $\vec{a}$ for q in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ where we map every variable to the same element in $\mathsf{db}\ (\mathcal{A})$ (arbitrary length atoms only implies that there is a match for $\vec{a}$ for q in $\mathsf{db}\ (\mathcal{A})$). It is simple to verify that π' is a match for $\vec{a}$ such that the depth of each mapped variable is zero.

Assume y only occurs in q in the form of $(r \cup r_1 \cup \cdots \cup r_n)^*(x, y)$. If y is bound in q, then there must be another arbitrary length atom $\rho(x', y)$ in q such that it can be merged with $(r \cup r_1 \cup \cdots \cup r_n)^*(x, y)$. Their intersection contains at least r and since π is a match for $\vec{a}$ for q, it implies that the terms of the atoms are unifiable. The result of merging these two arbitrary length atoms is another arbitrary length atom which contains r in its roles. For the resulting query q', we can construct a mapping π' such that π' is a match for $\vec{a}$ in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$. Hence, by repeatedly applying this operation, we can obtain a query q' such that y is unbound. If y is unbound in an arbitrary length path expression and $\pi(y)$ is the deepest element of a branch in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, then we drop the atom in q (y unbound in q in an arbitrary length atom implies that we can make a match for $\vec{a}$ that "does not go to y").

We now turn to the case where there is at least one atom of the forms $A(y)$, or $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ in q. Because $\pi(y) = v$, and v is a leaf in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, for all $B(y) \in q$, it must hold that $\mathcal{T} \models \exists r \sqsubseteq B$. It is easy to see that we can rewrite $B(y)$ into $(r \cup s_1 \cup \cdots \cup s_n)(\_, y)$ in a finite number of steps with the rules in Table 4.1. We have shown that we can rewrite q into a query q' where all atoms are binary atoms of the form $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ or $(r \cup r_1 \cup \cdots \cup r_n)^*(x, y)$ in a finite number of steps. Moreover, we can construct a match π' for $\vec{a}$ for q'.

If y is a bound variable, there must be at least one other atom in q' that uses r and has

y as a variable.

- Assume the other atom is of the form $(r \cup s'_1 \cup \cdots \cup s'_m)(x', y)$. Then, there are two operations that are possible. One, the atoms are unifiable i.e. contain exactly the same roles. The result is a new query q'' with either $(r \cup s'_1 \cup \cdots \cup s'_m)(x, y)$ or $(r \cup s'_1 \cup \cdots \cup s'_m)(x', y)$, depending on which of x or x' is bound or unbound.

  Otherwise, the atoms can be merged because their intersection contains at least r and since π' is a match for $\vec{a}$ for q', it implies that the terms of the atoms are unifiable.

  The resulting query q'' then contains the intersection of the roles, which is closed under the sub-role hierarchy, and either (x, y) or (x', y) as terms, depending on which of x or x' is bound or unbound. In both cases, we can construct a match π'' for $\vec{a}$ in the rewritten query q''. Moreover, y may become unbound as a result of the operations.

- Assume the other atom is of the form $(r \cup r_1 \cup \cdots \cup r_n,)^*(x', y)$. Again, there are two operations that are possible. First, the atom can be merged into a non-arbitrary length atom such that r is included in the roles. Similar to the case above, the resulting query q'' contains this atom with either (x, y) or (x', y) as terms, depending on which of x or x' is bound or unbound.

  If the roles of the atom $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ are not a subset of the arbitrary length atom, then it can not be concatenated to the arbitrary length atom. However, this implies that $\pi'(y)$ can only be reached by an r-relation, because π' is a match for $\vec{a}$. This case is covered if we merge the atoms as above.

  Otherwise, the atom $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ can be concatenated to the arbitrary length atom with the arbitrary length atom at the front. The result is a query q'' that contains the atoms $(r \cup r_1 \cup \cdots \cup r_n,)^*(x', x)$, $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$.

  In both cases, we can construct a match π'' for $\vec{a}$ in the rewritten query q''. Moreover, y may become unbound as a result of the operations.

By repeatedly merging, unifying and concatenating $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ with the other atoms in the query, we get that y becomes an unbound variable. Finally, we can apply the PI $A \sqsubseteq \exists r$ to the atom to obtain a query which replaces $(r \cup s_1 \cup \cdots \cup s_n)(x, y)$ with $A(x)$. Hence, we have mapped the query variables in the branch of $\pi(y)$ in $\mathcal{I}_{\mathcal{T},\mathcal{A}}$ one step closer to $\mathsf{db}(\mathcal{A})$. □

We have assumed nre-semantics in our correctness and completeness proofs. However, we can also show that our framework returns the certain answers under h-semantics. In Lemma 3.2.2, we have shown that $\mathsf{ans}_h(q, \mathcal{I}_{\mathcal{T},\mathcal{A}}) = \mathsf{ans}_{nre}(q, \mathcal{I}_{\mathcal{T},\mathcal{A}})$. Furthermore, we have defined the answers in our framework to be the answers obtained from the canonical model $\mathcal{I}_{\mathcal{T},\mathcal{A}}$, where the answers coincide for nre- and h-semantics if $\mathcal{T}$ is Ξ-compliant and

$\mathcal{A}$ is Ξ-acyclic. It remains to show that the execution of the rewritten query returns the intended answers.

Lemma 4.3.3 (Evaluation of Rewriting with Homomorphism-Based Semantics)**.** *Let q be a Ξ-restricted query, $\mathcal{T}$ a Ξ-compliant TBox, and $\mathcal{A}$ a Ξ-acyclic ABox such that $(\mathcal{T}, \mathcal{A})$ is satisfiable. Then,*

$$\bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_h(q', \mathsf{db}(\mathcal{A})) = \bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A})).$$

Proof. If $\mathcal{A}$ is Ξ-acyclic, then $\mathsf{db}(\mathcal{A})$ is Ξ-acyclic by definition. From Lemma 3.2.1 it follows that for any Ξ-restricted query q $\mathsf{ans}_h(q, \mathsf{db}(\mathcal{A})) = \mathsf{ans}_{nre}(q, \mathsf{db}(\mathcal{A}))$. As a consequence, $\bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_h(q', \mathsf{db}(\mathcal{A})) = \bigcup_{q' \in \mathsf{Rewrite}(q,\mathcal{T})} \mathsf{ans}_{nre}(q', \mathsf{db}(\mathcal{A}))$. □

Finally, we have established that we can use query rewriting for Ξ-restricted queries, Ξ-compliant TBoxes and Ξ-acyclic ABoxes. The answers are defined by the answers over the canonical model, which can be retrieved from a property graph database which uses nre-semantics for paths in its query language. Additionally, we have shown that our framework also returns the answers under the standard semantics for OMQ.

4.4 Query Answering with Rewriting

With a correct and complete rewriting procedure in place, we can now describe a method for answering Ξ-restricted queries. In our discussion of answering queries in our framework, we decouple checking that the restrictions in our framework hold from evaluating the queries.

4.4.1 Checking Framework Restrictions

The restrictions are dependent on the role clustering Ξ. For a system that uses our framework for query answering, there are two options for checking the restrictions of the framework. One option is that the role clustering Ξ is given by the users, and the other is inferring a minimal role clustering from the query. Still, checking that the TBox is Ξ-compliant and the ABox $\mathcal{A}$ is Ξ-acyclic can be done independently of how the role clustering Ξ was created.

If the role clustering is not given by the users, then we can generate the role cluster Ξ in the following way: For each path atom in the input query q, we collect the role names that occur in a set. Then, we saturate each set of roles, yielding a role clustering.

Should a role clustering be given, we also need to saturate each set in the clustering. From the definition of role clusterings (Definition 3.1.2), it follows that each set in the clustering is already saturated. However, if the users only define sets of roles, which are not necessarily role clusterings, we can still saturate the sets to generate a role clustering.

Asserting that $\mathcal{T}$ is Ξ-compliant and $\mathcal{A}$ is Ξ-acyclic is relatively simple. If an inverse is in the saturation of the roles, then $\mathcal{T}$ is not Ξ-compliant. Then, for each saturated set of roles, we check that they are acyclic in $\mathsf{db}(\mathcal{A})$. For a database that uses Cypher as a query language, this can be queried as shown in Figure 4.1. The query depicted in Figure 4.1 assumes that the roles that should be checked for acyclicity are r, s, and t. If there is a path using a combination of these roles that returns to the same node, then the query returns a result. Otherwise, the result of the query is empty. We note that the query is tractable in data complexity for both h-and nre set semantics because it falls into the category of single-occurrence RPQs [MNT20].

```
1 MATCH (x)-[:r|s|t *]->(x)
2 RETURN DISTINCT x
3 LIMIT 1
```

Figure 4.1: Cypher query to check acyclicity of roles

If this check fails, then our rewriting procedure is not guaranteed to return the correct results. How a system that implements our approach handles this case can be varied, for example an error message could be shown to the users that the query can not be answered correctly.

4.4.2 Query Answering

The algorithm Answer is based on the answering procedure for PerfectRef [CGL+07] and takes as input a Ξ-restricted query q, a TBox $\mathcal{T}$, and an ABox $\mathcal{A}$. We present this procedure in Algorithm 4.3.

First, the answering procedure must check whether the knowledge base (without considering the query) is satisfiable with the function `Consistent`. Calvanese et al. have shown that this consistency check can reduced to answering a UCQ over $\mathsf{db}(\mathcal{A})$ [CGL+07, Theorem 17]. If the knowledge base is unsatisfiable, then every tuple in $\mathsf{Ind}(\mathcal{A})$ of the same arity as q is an answer to the query. The function `AllTuples` returns all tuples in $\mathsf{Ind}(\mathcal{A})$ with the same arity as q. In Cypher, a query to retrieve all tuples is fairly straightforward to construct. Figure 4.2 shows a Cypher query which returns all tuples of arity two in the database.

```
1 MATCH (x), (y)
2 RETURN *
```

Figure 4.2: Cypher query to return all tuples of arity two

Then, we can call the rewriting procedure Rewrite with q and $\mathcal{T}$ to generate the UCQ Q that should be evaluated over the plain data $\mathsf{db}(\mathcal{A})$.

Algorithm 4.3: Answer

Input: A Ξ-restricted query q, a Ξ-compliant TBox $\mathcal{T}$, and a Ξ-acyclic ABox $\mathcal{A}$

```
1 if not Consistent(T, A) then
2 |   return AllTuples(q, A);
3 Q ← Rewrite(q, T);
4 return ⋃_{q_i ∈ Q} ans(q_i, db(A));
```

This answering procedure returns the correct answers, as the next lemma shows.

Lemma 4.4.1 (Correctness of Answer)**.** *$\vec{a}$ is a certain answer to a Ξ-restricted query q over $(\mathcal{T}, \mathcal{A})$ if and only if $\vec{a} \in \mathsf{Answer}(q, \mathcal{T}, \mathcal{A})$.*

Proof. The correctness of Answer in the case of a satisfiable knowledge base $(\mathcal{T}, \mathcal{A})$ is a direct consequence of the correctness and completeness of Rewrite shown in Lemmas 4.3.1 and 4.3.2. If the knowledge base is unsatisfiable, then it is easy to check that $\mathsf{AllTuples}(q, \mathcal{T}, \mathcal{A}) = \mathsf{cert}(q, \mathcal{T}, \mathcal{A})$. □

Finally, we can discuss the complexity of query answering with our framework. In general, the complexity of query answering in DL-Lite depends on the complexity of evaluating C(2)RPQs in a (graph) database.

Lemma 4.4.2 (Complexity of Ξ-restricted query answering in DL-Lite)**.** *The complexity of* Answer *is*

1. NL *in data complexity.*
2. NP*-complete in combined complexity.*

Proof. Our proof is based on the following two observations:

1. The check for consistency can be performed in non-deterministic logarithmic space in $|\mathcal{A}|$ [CGL$^+$07]. The procedure Rewrite constructs a set of C2RPQs where there are no concatenations of paths, which means that each role in a Kleene star appears at most once in an arbitrary length atom. Hence, the evaluation of a union of such queries is in NL in data complexity under both h-and nre set semantics.
2. Similar to the argument for PerfectRef [CGL$^+$07], we can formulate a version of Rewrite which doesn't generate a set of queries Q, but non-deterministically returns one $q' \in Q$. We know that the number of queries generated by Rewrite is polynomial in the size of the TBox (cf. Lemma 4.2.1). Therefore, we can generate one query q' in a polynomial number of steps from the initial query q. Confirming the consistency of the knowledge base is feasible in combined complexity [CGL$^+$07]. Therefore, for a boolean Ξ-restricted query, a non-deterministic version of Answer is

in NP. NP-hardness follows from the NP-hardness of evaluating C2RPQs in graph databases.

□

Recall that C2RPQ answering for arbitrary path atoms is NL in data complexity and PSPACE-complete in combined complexity for DL-Lite [BOS15]. Indeed, our query language is less expressive than C2RPQs, but gains the advantage of lower combined complexity. Moreover, our query language pushes the limit of what we can express in Cypher, and is tractable in data complexity under no-repeated-edges set semantics.

CHAPTER 5

Implementation

We implemented the algorithm Rewrite that we described in Chapter 4 in an object-oriented way with Java as a simple CLI tool. The code for our project is publically available on Github[1]. Figure 5.1 depicts the workflow of the CLI: Upon staring the tool, the users are first asked to provide an ontology file and then a query using the vocabulary provided in the ontology. Then, the query is rewritten with regard to the ontology as laid out in Algorithm 4.1. Finally, the rewritten query is translated into a Cypher query that can be executed over a database. This query is copied to the users' clipboard, such that they can copy-and-paste it into a Neo4j Cypher shell or the Neo4j Desktop application.

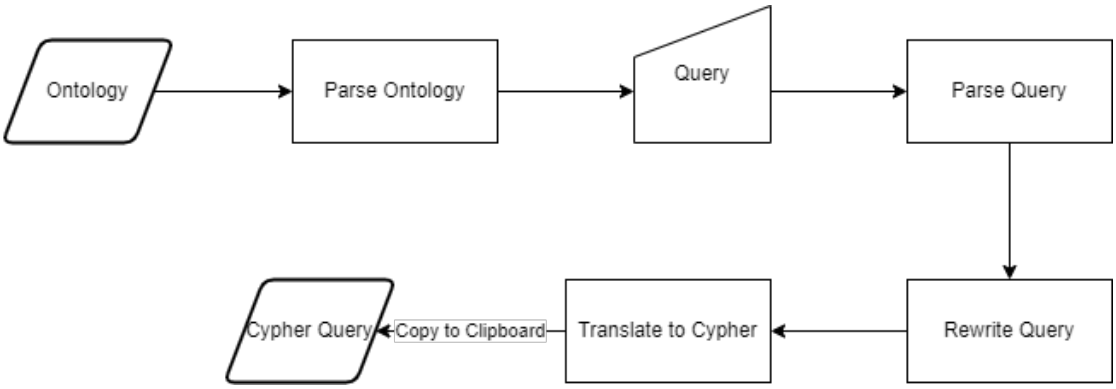

Figure 5.1: Implementation Workflow

First, we describe each object in our implementation and the corresponding concept from ontology-mediated querying. We will use serif to denote classes or functions in our implementation, and the standard font otherwise. Following that, we present the context-free ANTLR4 grammar used to define the query language we discussed in Chapter 3.

[1] https://github.com/nikdra/omq-cypher, accessed 17th May, 2022

In addition, we briefly outline how we used ANTLR4 to parse the query string into a query object our rewriter recognizes. The interface Rewriter and its lone implementation RewriterImpl contain the Java version of the rewrite procedure we defined in Algorithm 4.1. As the last step of our implementation, we show how we translated query objects and sets thereof into a Cypher query string. We illustrate the usage of our implementation by a simple example. Finally, we discuss the advantages and drawbacks of our implementation and extensions for future work.

5.1 Object Representation

Each conceptual object in OMQ is represented by a Java object, and therefore a specific class. For more general concepts, we used interfaces and abstract classes to capture shared traits.

5.1.1 Ontology

As the name suggests, the Ontology class represents an ontology. We used the OWL API [Hor11] for reading and storing OWL ontology axioms. More specifically, we store the OWLOntology object from the OWL API in a variable of our Ontology class. An OWLOntology object is a set of OWLAxioms, which is the pendant to our notion of a TBox $\mathcal{T}$. The constructor of the Ontology class accepts a file path string as an argument executes the following three steps:

1. Load the ontology specified by the file path, and assert that the ontology adheres to the OWL2 QL standard.
2. Generate a map classMap from the signature of the ontology with the basic concept names as the keys and the IRIs of the basic concepts as the values.
3. Generate a map propertyMap from the signature of the ontology with the role names as the keys and the IRIs of the roles as the values.

The maps classMap and propertyMap serve the purpose of connecting the concept and role names the users use in the query to IRIs in the ontology. We can then store the IRIs in the atoms' objects so that we can access them during reasoning with axioms, where only IRIs are used.

5.1.2 Terms

Recall that the terms of a query are variables or individuals. Moreover, a variable can be bound or unbound. In our implementation, we omit individuals. This can be simulated by using dedicated concept names. For example, we can assume that an assertion $a : \mathsf{N_a}$ in present in the ABox for each individual a, where $\mathsf{N_a}$ is a concept that does not occur anywhere else. Then, we can replace each occurrence of an individual a in a query with

a fresh variable x_a and add the atom $\mathsf{N_a}(x_a)$ to the query. Additionally, Neo4j does not recommend using its internal node identifiers in applications [Neo21b]. Rather, users should utilize application-generated identifiers and store them as properties. An ideal way for us to obtain unique names would therefore be from the mappings of a complete OBDA system for property graphs, or to use properties in querying. Both of these approaches are outside the scope of this thesis, thus we leave this topic for future work.

We decided to implement Terms as an interface. The classes UnboundVariable and BoundVariable implment this interface. Objects that implement the Terms interface can be used in objects of Atom. Figure 5.2 depicts an overview of the Terms interface. The function applySubstitution applies a substitution to a term and returns the term that results from the substitution. For convenience, we implemented the functions getName and getFresh, which return the name of the term and a new Term object with the same name, respectively.

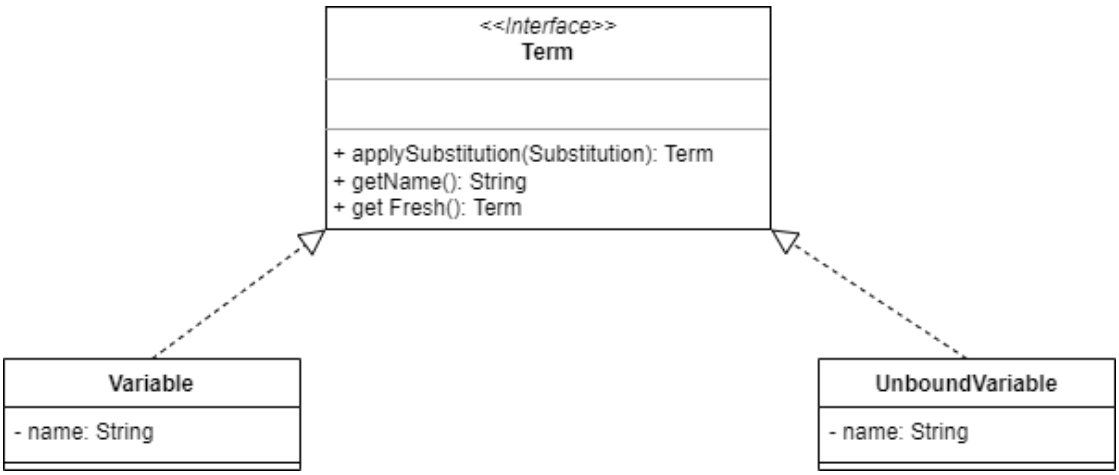

Figure 5.2: UML class diagram of Terms

5.1.3 Query

We defined an interface Query as an umbrella for the classes InputQuery and RewritableQuery. Both these classes have a variable head that is a set of Variable, and a variable body that is a set of Atom. The main difference between those classes is that RewritableQuery only has RewritableAtoms in its body that can be used in the main loop of the rewriting algorithm. In essence, the rewriter generates a RewritableQuery from an InputQuery in the first step of the rewriting, which is SaturatePaths.

5.1.4 Atom

Atoms are, both conceptually and in our implementation, the basic building blocks of the body of a query. An overview of the types of atoms we implemented is shown in Figure 5.3.

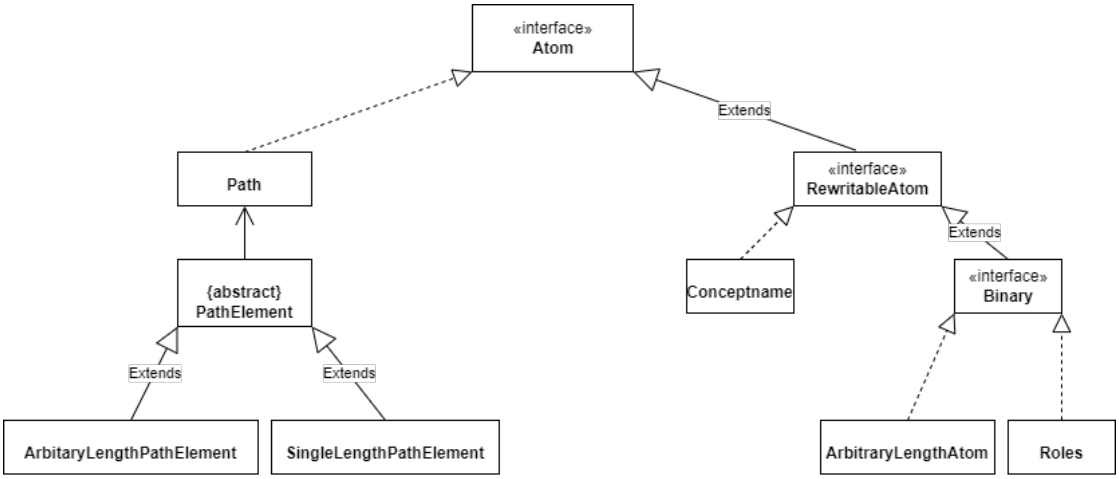

Figure 5.3: UML class diagram of atoms and related elements

Our object representation is a reflection of the different types of atoms in a Ξ-restricted query from Definition 3.2.1. The class Conceptname represents an atom of the form $A(x)$, while the class Role represents an atom of the form $(s_1 \cup s_2 \cup \dots s_n)(x, y)$. A path atom is a special case, because it contains a regular expression with concatenation, possibly with a Kleene star. We represent the elements of the concatenation with ArbitraryLengthPathElement and SingleLengthPathElement.

Path

Ignoring requirements on the TBox and ABox, a path atom contains concatenations of the form $(r_1 \cup \dots \cup r_n)$ or $(r_1 \cup \dots \cup r_n)^*$. We refer to each conjunct as a *path element*, reflected by the PathElement abstract class. In addition, a Path atom has two terms. Figure 5.4 shows an overview of paths and path elements with their fields and functions.

Path Elements

The abstract class PathElement defines that each subclass contains a set of role names, and can be converted to a Binary atom given two terms. In addition, the abstract class implements a function that facilitates saturation of the paths given an ontology.

A path element of the form $(r_1 \cup \dots \cup r_n)$ is a single length path element, denoted by the SingleLengthPathElement class. Conversely, a path element of the form $(r_1 \cup \dots \cup r_n)^*$ is an arbitrary length path element, represented by the ArbitraryLengthPathElement class.

Rewritable Atoms

Rewritable atoms are atoms of the forms that can occur in the main loop after the path atoms have been split up in Rewrite. Figure 5.5 provides an overview of the interfaces and classes that implement the RewritableAtoms interface.

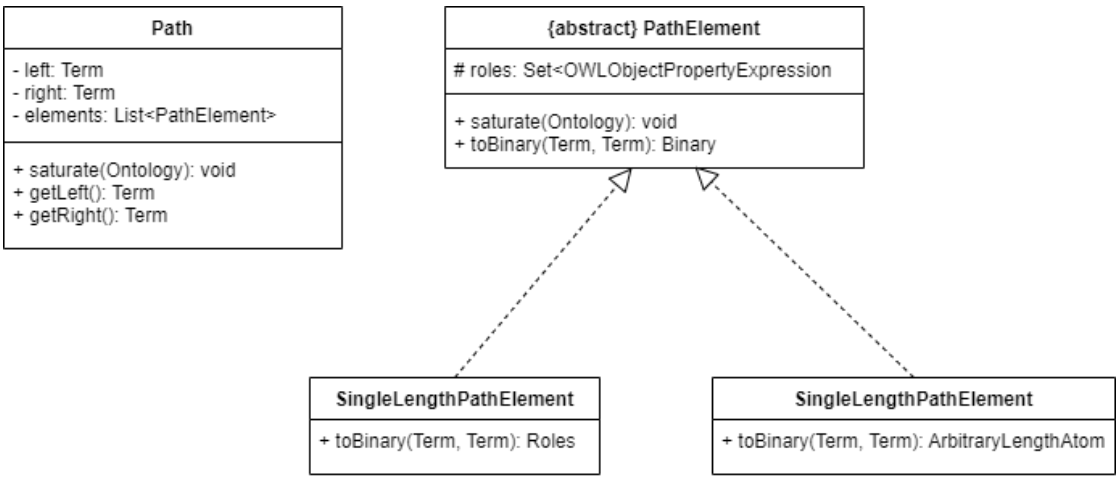

Figure 5.4: UML class diagram for paths and path elements

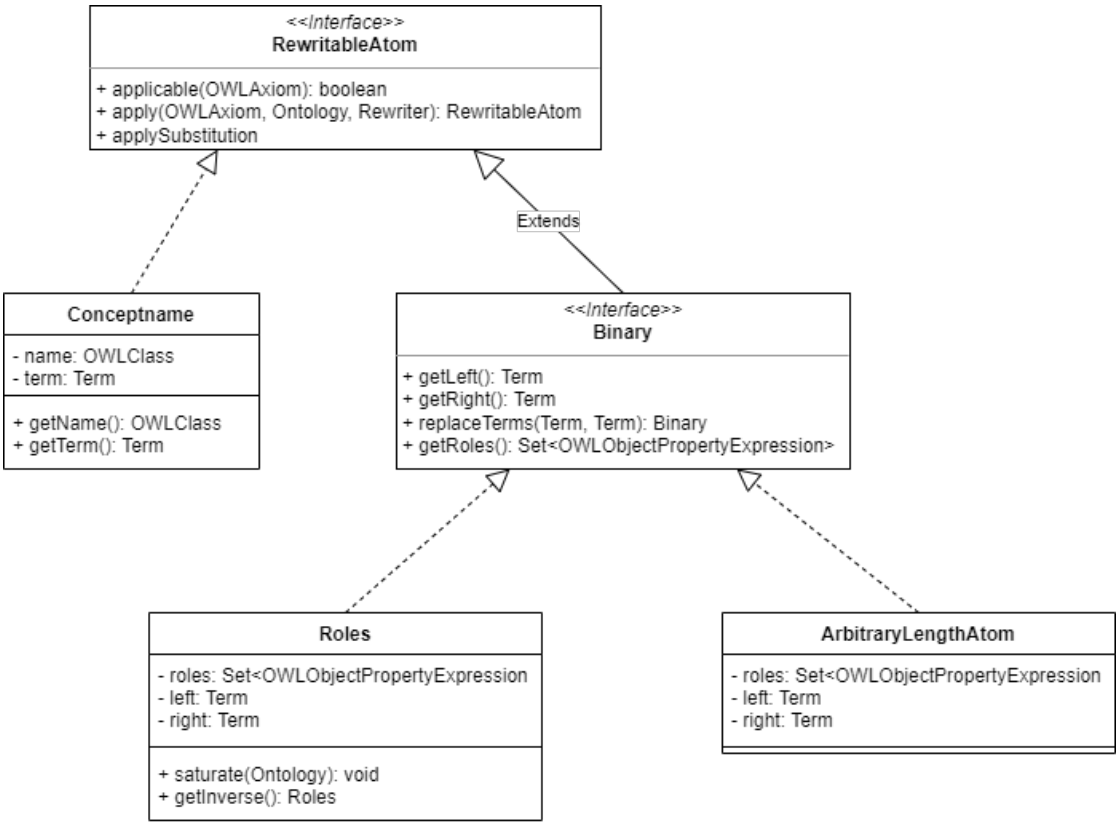

Figure 5.5: UML class diagram of rewritable atoms

All classes that implement the RewritableAtom interface implement the following three functions:

- applicable: Given an OWLAxiom object, determine whether the axiom is applicable to the atom as specified in Section 4.1.2. Note that for ArbitraryLengthAtom objects, this function must always return false.
- apply: Given an OWLAxiom object, an Ontology object and a Rewriter object, return the RewritableAtom that results from applying the given axiom to this atom. The precondition for correctness of this function is that applicable was called before and returned true. Because the result of apply might return a Role, we hand over the ontology as a parameter such that the atom can be saturated. The Rewriter object keeps track of the unbound variables that have been created in the rewriting process, hence we need it in case a new unbound variable is introduced.
- applySubstitution: Given a substitution, apply the substitution to the terms of the atom.

The Binary interface defines functions that we use for convenience in our implementation. Moreover, it implies that each binary atom contains a set of OWLObjectPropertyEpression objects, which are objects that represent roles or their inverses in the OWL API.

Conceptname

This class represents atoms of the form $A(x)$. We use the variable name to store the OWLClass of the atom.

Roles

Binary atoms of the form $(s_1 \cup \cdots \cup s_n)(x, y)$ are represented by the Roles class. Over the course of the query rewriting process, it might be necessary to saturate the atom with regard to the ontology. Therefore, Roles defines the function saturate with Ontology with a parameter.

Arbitrary Length Atoms

Arbitrary Length Atoms are of the form $(r_1 \cup \cdots \cup r_n)^*(x, y)$. In the rewriting, they can only be modified by merging or concatenation. There are no rules to replace an arbitrary length atom, therefore the functions applicable and apply required by the RewritableAtom interface return false and null, respectively.

5.1.5 Unifier and Substitution

The Unifier class takes as class constructor arguments two query atoms and computes the most general unifier, if one exists. We chose the algorithm described by Martelli and

Montanari [MM82] to determine the most general unifier. For the unifier, we chose to represent it as a list of Substitution objects and store it in the Unifier object. If no unifier exists, then the list is empty.

5.2 Query Parsing

We decided to use the Java ANTLR 4 library [Par13] to parse the users' queries to the CLI. ANTLR 4 provides the functionality to parse input defined by a context-free grammar. Moreover, it can automatically generate not only parser classes that can be used in Java directly, but also a parse tree visitor. In our implementation, we extended the visitor to transform the parse tree of our query into a InputQuery object with Atom objects. Figure 5.6 shows a Java code snippet which we use in our implementation to parse an input query in string form. The InputQueryBuilder class is our extension of the Visitor generated by ANTLR 4.

```
1 // parse query from string
2 CharStream cs = CharStreams.fromString(queryString);
3 QLexer lexer = new QLexer(cs);
4 QParser parser = new QParser(new CommonTokenStream(lexer));
5
6 ParseTree tree = parser.query();
7
8 InputQuery q = (InputQuery) new InputQueryBuilder(ontology).visit(tree);
```

Figure 5.6: Java example for parsing a query string with ANTLR

The syntax of the queries mostly adheres to the syntax we have used in our examples throughout this thesis. However, for path atoms, the concatenation symbol for the path elements is "/". We present the grammar we used for parsing Ξ-restricted queries in Figure 5.7.

5.3 Rewriting

The Rewriter interface defines the functions we used in the main loop of the algorithm Rewrite (cf. Algorithm 4.1). We implemented the interface in the class RewriterImpl, where the function rewrite is the function that should be called to rewrite an input query. No arguments are needed to instantiate a new RewriterImpl object. The main class of our implementation which defines the CLI, simply instantiates a new Rewriter and passes the input query to the rewrite function of the Rewriter object.

```
1  grammar Q ;
2
3  query : head ':-' body EOF ;
4
5  head : 'q(' (variable (',' variable)*)? ')' ;
6
7  variable : WORD ;
8
9  body : atom (',' atom )* ;
10
11 atom : conceptname | roles | path;
12
13 conceptname : words '(' variable ')' ;
14
15 roles : properties '(' left=variable ',' right=variable ')' ;
16
17 path : elements '(' left=variable ',' right=variable ')' ;
18
19 elements : pathElement ('/' pathElement)* ;
20
21 pathElement : arbitraryLengthPathElement | singleLengthPathElement ;
22
23 arbitraryLengthPathElement : rolename '*' | '(' rolename ('|' rolename)+ ')' '*';
24
25 singleLengthPathElement : rolename | '(' rolename ('|' rolename)+ ')' ;
26
27 properties : property | '(' property ('|' property)+ ')' ;
28
29 property : rolename | inverse ;
30
31 rolename : words ;
32
33 inverse : words'-' ;
34
35 words : WORD ('_' WORD)* ;
36
37 WORD : LETTER+ ;
38
39 fragment LETTER : ('a'..'z' | 'A'..'Z') ;
40
41 UNKNOWN_CHAR : . ;
```

Figure 5.7: ANTLR 4 grammar for our query language

5.4 Cypher Translation

The translation of (unions of) RewritableQuery objects is carried out by a specific class, the CypherTranslator. This class implements the interface Translator, which specifies only one function, translate. The discussion of the remainder of this section is based on Cypher/Neo4j v4.4 [Neo21b].

In our discussion of Ξ-restricted queries, we assumed nre-semantics only for the path expressions, because we can express CQs with homomorphism-based semantics in Cypher. Only path expressions remain under nre-semantics, which we have shown return the same answers under homomorphism-based semantics provided the ABox and TBox are Ξ-acyclic and Ξ-compliant, respectively.

Let $q(\vec{x}) = \exists y.\phi$ be a Ξ-restricted query, where the path atoms have been broken up into individual binary atoms. Then, we can express this query in Cypher by replacing the atoms in the query with the Cypher expressions given in Table 5.1.

Finally, we place the free variables in a return statement at the end of the Cypher query i.e., `RETURN distinct` $x_1, x_2, \ldots, x_n$ for each $x_i \in \vec{x}$. If there are no answer variables (i.e. the query is boolean), we set the return clause to `RETURN 1`. This will return 1 for each match of the query, and nothing otherwise.

Another special case we must mention is atoms with role names and their inverses. Because such atoms can be matched "in both directions" depending on the role name, we can not set the direction of the edge in the Cypher clause. Moreover, such atoms can only be expressed in Neo4j's implementation of Cypher, as we have to use the scalar function `startNode()`[2] to determine the direction of the relation, which is part of the opencypher standard[3], but *not* necessarily implemented in every graph DBMS. This scalar function can only be used in the `WHERE` clause of the query, and we have to add such constraints for each atom that has both role names and inverses. For example, consider the query $q(x) \leftarrow (\mathsf{teaches} \cup \mathsf{taughtBy}^-)(x, y)$. In Neo4j's Cypher, this can be expressed as shown in Figure 5.8.

However, we note that there is an alternative to using Neo4j's Cypher implementation. For each query that contains a binary atom with inverses, we can formulate two queries where one query contains the atom with only the role names, and the other the "inverse" version of atom with only the inverses. Still, this has to be done for each combination of binary atoms in the query, raising the number of queries exponentially.

[2] `https://neo4j.com/docs/cypher-manual/4.4/functions/scalar/`, accessed 17th May, 2022

[3] `https://opencypher.org/resources/`, accessed 17th May, 2022

```
match (x)-[r1:teaches|taughtBy]-(y)
where (startnode(r1) = x and type(r1) = "teaches") or (startnode(r1) = y and
type(r1) = "taughtBy")
return distinct x
```

Figure 5.8: Cypher translation of queries with role names and their inverses

Atom	Cypher Match Clause
$A(x)$	`MATCH (x:A)`
$r(x,y)$	`MATCH (x)-[:r]->(y)`
$(r \cup \cdots \cup s)(x,y)$	`MATCH (x)-[:r|...|s]->(y)`
$(r \cup \cdots \cup s^-)(x,y)$	`MATCH (x)-[:r|...|s]-(y)`
$(r \cup \cdots \cup r_n)^*(x,y)$	`MATCH (x)-[:r|...|s *0..]->(y)`
$r^*(x,y)$	`MATCH (x)-[:r *0..]->(y)`

Table 5.1: Rewriting of Ξ-restricted query atoms to Cypher

5.5 Project Structure, Dependency Management and Building

Our project uses Java 11 for compilation. The `src` directory of the repository contains both our source code in `main/java/at/ac/tuwien/informatics` and the tests we wrote over the course of our test-driven development strategy in `test`. We decided to use Maven[4] (Version 4.0.0) to manage the dependencies of our implementation. The dependencies are listed in the `pom.xml` file of our repository. Table 5.2 contains a summary of dependencies in `pom.xml`. Moreover, we also used the ANTLR 4 plugin for Maven.

Group ID	Artifact ID	Version
org.antlr	antlr4-runtime	4.9.3
org.junit.jupiter	junit-jupiter-engine	5.4.0
net.sourceforge.owlapi	owlapi-distribution	5.1.20

Table 5.2: Implementation dependencies

The project can be built with Java 11 and by executing the commands `maven clean compile` and `maven package`. This command generates an executable `.jar` file which contains all needed dependencies and starts the CLI. In an IDE, the class Cli contains the main function of the project.

[4] `https://maven.apache.org/`, accessed 17th May, 2022

5.6 Example Rewriting

We now present an example workflow for our implementation with the ontology from Example 4.1.7. Upon starting the CLI in our console, we are asked to provide a path to an ontology file. Assume our ontology is stored at `my/path/ontology.owl` and contains the elements shown in Figure 5.9. This OWL file represents the TBox $\mathcal{T} = \{A \sqsubseteq \exists r^-, \exists r \sqsubseteq \exists s^-, \exists s \sqsubseteq \exists t^-\}$.

Then, we are asked to specify a query that should be rewritten with regard to the given ontology. We specify the following query: `q(x):-t*(y,zp),s*(zp,zpp),r(zpp,x).` The resulting Cypher query is a union of four queries, shown in Figure 5.10.

5.7 Summary

In this chapter, we described our object-oriented implementation of the rewriting algorithm that we defined in Chapter 4. Our implementation is a simple CLI tool that accepts an ontology in OWL format and a query as an input, and returns a (Neo4j) Cypher query as the output. However, there are some limitations to our implementation.

First, we did not implement using individuals (constants) as terms in query atoms. The main reason is that in OMQ systems, individual names are represented by IRIs. In the Neo4j proprietary property graph model, nodes are represented by internal identifiers (an integer identifier). However, Neo4j does not recommend using internal identifiers directly, and instead proposes application-generated identifiers which are stored as properties. Therefore, for an appropriate use of individuals in queries, we would have to define mappings from Cypher nodes and their properties to IRIs, or add querying of properties. Both of these options are outside of the scope of this thesis.

Second, our rewritten queries are written into Neo4j's Cypher implementation. We use functions specific to Neo4j to express combinations of role names and inverse roles in binary atoms. However, each query containing an atom with inverse roles can be re-expressed in two queries. One for the the inverse roles and one for the role names in the atom. Even if the queries can be written into the Cypher syntax described by Francis et al. [FGG+18], the number of queries can become exponentially larger.

Finally, our implementation is not an answering algorithm. For this, a check of the ABox, and therefore a database connection, is necessary. Still, this can be added with Neo4j's Java driver[5]. Nevertheless, the correctness of our rewriting is still given if the underlying data is not inconsistent with the ontology.

[5] `https://neo4j.com/developer/java/`, accessed 17th May, 2022

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.semanticweb.org/crpqs1#"
     xml:base="http://www.semanticweb.org/crpqs1"
     xmlns:owl="http://www.w3.org/2002/07/owl#"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:xml="http://www.w3.org/XML/1998/namespace"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
    <owl:Ontology rdf:about="http://www.semanticweb.org/crpqs1"/>

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/crpqs1#r">
        <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
        <rdfs:domain>
            <owl:Restriction>
                <owl:onProperty>
                    <rdf:Description>
                        <owl:inverseOf rdf:resource="http://www.semanticweb.org/crpqs1#s"/>
                    </rdf:Description>
                </owl:onProperty>
                <owl:someValuesFrom rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
            </owl:Restriction>
        </rdfs:domain>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/crpqs1#s">
        <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
        <rdfs:domain>
            <owl:Restriction>
                <owl:onProperty>
                    <rdf:Description>
                        <owl:inverseOf rdf:resource="http://www.semanticweb.org/crpqs1#t"/>
                    </rdf:Description>
                </owl:onProperty>
                <owl:someValuesFrom rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
            </owl:Restriction>
        </rdfs:domain>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="http://www.semanticweb.org/crpqs1#t">
        <rdfs:subPropertyOf rdf:resource="http://www.w3.org/2002/07/owl#topObjectProperty"/>
    </owl:ObjectProperty>

    <owl:Class rdf:about="http://www.semanticweb.org/crpqs1#A">
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty>
                    <rdf:Description>
                        <owl:inverseOf rdf:resource="http://www.semanticweb.org/crpqs1#r"/>
                    </rdf:Description>
                </owl:onProperty>
                <owl:someValuesFrom rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>
</rdf:RDF>
```

Figure 5.9: Example OWL ontology in RDF format

```
1  match (zpp)-[:r]->(x)
2  match (zp)-[:s*0..]->(zpp)
3  match (y)-[:t*0..]->(zp)
4  return distinct x as x
5  union
6  match (x:A)
7  return distinct x as x
8  union
9  match (zpp)-[:r]->(x)
10 match (zp)-[:s*0..]->(zpp)
11 return distinct x as x
12 union
13 match (zpp)-[:r]->(x)
14 return distinct x as x
```

Figure 5.10: Cypher result of rewriting

CHAPTER 6

A Real-World Use Case

The Virtual Vehicle Research GmbH[1] (VVR) is Europe's largest research and development center whose main focus is the development of application-oriented vehicle development and future vehicle concepts. A key interest of VVR is "smart mobility" with the goal that future vehicles should be affordable, safe and environmentally friendly. VVR mainly researches the integration of numerical simulation and hardware testing. From the testing and simulation of single components and technologies to complete vehicles, VVR aims to integrate hardware and software aspects of autonomous driving into one cohesive vehicle perspective. More than 100 national and international industrial partners and over 40 national and international scientific institutions have collaborated with VVR.

One component of modern vehicles is autonomous driving, which has been mainly propelled by ever more powerful machine learning models. Training and validation of machine learning models in and of itself is a challenge, more so for a safety-critical application such as autonomous driving. To this end, several open-source datasets for autonomous driving have been published, such as nuScenes [CBL+20] and Lyft [KUH+19]. These two datasets share the same schema. Hence, VVR uses the combination of these two datasets in their autonomous driving research. Even if the two datasets share the same meta model, they differ in the description of attributes and categories. Therefore, querying the two datasets together is difficult for users, especially if they are not familiar with the particularities of the datasets and the relationships between them. We aim to enable the users to ask queries which ensure data quality and extract scenarios which are safety-critical for autonomous vehicles. For this, we present an ontology which provides a shared vocabulary for these two data sets. The tool we developed in Chapter 5 adds the domain knowledge from the ontology into the query. We have identified five representative queries for the data which should be answered with regard to the domain knowledge.

[1] https://www.v2c2.at/, accessed 17th May, 2022

In the following, we will describe the nuScenes and Lyft datasets on a schematic level. We then present the VVR property graph, which is a combination of these two datasets, where some additional labels and relations have been added to facilitate easier querying. Next, we describe the queries which should be answered over the VVR property graph. Then we describe the ontology that binds the vocabularies in the annotations together such that the rewritten queries return results from both datasets. Finally, we analyze the rewritings of the queries with regard to the ontology, and discuss the results with respect to our findings in Chapter 5.

6.1 Dataset Description

The nuScenes [CBL+20] and Lyft dataset [KUH+19] have been designed with the objective to provide an open-source benchmark set for object detection and trajectory prediction of road users in the domain of autonomous driving. For both datasets, the data were collected in a similar manner. A car with radar, LIDAR and camera sensors was driven on public roads, and the data from each of the sensors were collected and annotated by humans. Moreover, the position and orientation of the vehicle (also referred to as the ego vehicle) is recorded on a map. While the number and exact placement of the sensors were different for the two datasets, they are annotated and organized according to the same schema. The schema of the datasets is written in such a way that it can be stored in a relational database. However, because the data are basically time series, they can be naturally translated to a graph schema. The meta model we show in Figure 6.1 and describe in the following is the same as the one described by NuScenes [CBL+20], but translated from a relational model to a graph model. In our description of the schema, we focus on the entities and the relationships between them.

Scenes are the main building blocks of the datasets. Each scene is a sequence of frames (samples). In the nuScenes and Lyft datasets, scenes are 20 and 25 seconds long, respectively. The log of a scene contains information about the vehicle, the location of the scene, and a relation to a map.

Samples are annotated keyframes. The data of a sample are those that are collected approximately to the timestamp of a single LIDAR sweep. For the nuScenes and Lyft dataset, the keyframes are collected at 2Hz.

Depending on the sensor, sample data can be an image (camera), point cloud (LIDAR) or a radar return. All sample data point to the sample that is closest in time. The associated ego pose is the vehicle pose at a particular time stamp based on the position on the map. In addition, the related calibrated sensor contains the calibration of a particular sensor on the ego vehicle, i.e. the position of the sensor relative to the ego vehicle. Each calibrated sensor has a relation to a sensor that describes its type.

Every sample has annotations associated with it. A sample annotation represents a bounding box of an object in a sample. With each sample annotation, the visibility of the object is also recorded by a relation. Moreover, the annotations also have a relation

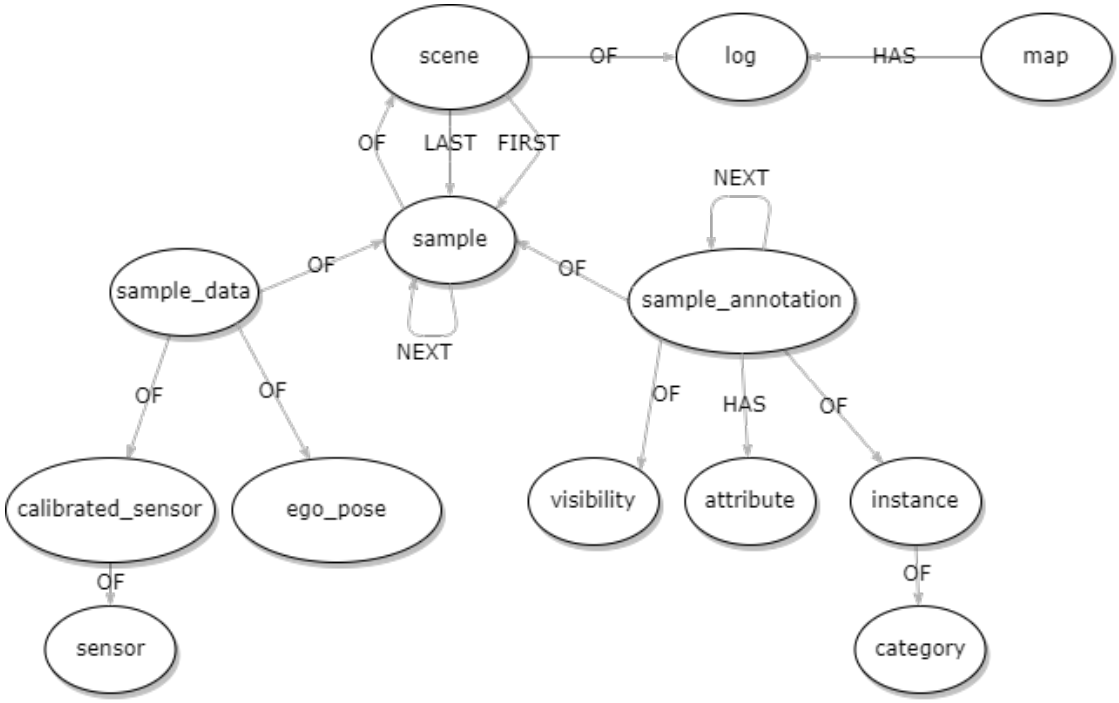

Figure 6.1: The dataset graph schema

to an attribute which describes the object of the annotation. Annotated objects are denoted by instances, which belong to specific categories. Categories are organized in simple category taxonomies, which we will describe in more detail in the next section.

6.2 VVR Property Graph Description

The VVR property graph stores both the Lyft and nuScenes datasets in the same Neo4j property graph database according to the graph schema depicted in Figure 6.1. However, the basic schema has been extended to facilitate better querying.

For one, two types of relations were added between instances and sample annotations. The relation FIRST_ANNOTATION connects each instance to its first annotation. Similarly, LAST_ANNOTATION connects each instance to its last annotation.

Second, each instance has been labelled with the semantic category it belongs to according to the information in the category node. The categories and their taxonomies are different for each instance, depending on which dataset it originally belongs to. To illustrate, the nuScenes dataset contains 23 different categories, while the Lyft dataset only has eight. Table 6.1 shows the names of the categories and which label has been added to each instance connected to it. For example, a node that is labelled with instance and has a relation to a category node that is named `animal`, is also labelled with animal.

Category name	Dataset	New instance label
animal	motional	animal
human.pedestrian.adult	motional	adult
human.pedestrian.child	motional	child
human.pedestrian.construction_worker	motional	construction_worker
human.pedestrian.personal_mobility	motional	personal_mobility
human.pedestrian.police_officer	motional	police_officer
human.pedestrian.stroller	motional	stroller
human.pedestrian.wheelchair	motional	wheelchair
movable_object.barrier	motional	barrier
movable_object.debris	motional	debris
movable_object.pushable_pullable	motional	pushable_pullable
movable_object.trafficcone	motional	trafficcone
static_object.bicycle_rack	motional	bicycle_rack
vehicle.bicycle	motional	bicycle
vehicle.bus.bendy	motional	bendybus
vehicle.bus.rigid	motional	rigidbus
vehicle.car	motional	car
vehicle.construction	motional	constructionvehicle
vehicle.emergency.ambulance	motional	ambulance
vehicle.emergency.police	motional	police
vehicle.motorcycle	motional	motorcycle
vehicle.trailer	motional	trailer
vehicle.truck	motional	truck
pedestrian	Lyft	pedestrian
animal	Lyft	animal
other_vehicle	Lyft	vehicle
bus	Lyft	bus
motorcycle	Lyft	motorcycle
truck	Lyft	truck
emergency_vehicle	Lyft	emergency
bicycle	Lyft	bicycle

Table 6.1: Categories and associated instance labels

The graph schema shows that each sample_annotation can have a HAS relation to an attribute node. This structure allows modelling changes of the status of instances (or objects) in the scene over time. For example, if the attribute were connected to the instance, then the attribute would be static. Moreover, it could lead to problems during reasoning if the object were moving and stopping in the scene over time. Any axioms that encode that an object can not be stationary and moving at the same time would lead to an inconsistency. In the VVR property graph, the semantic label has been added to all attribute nodes according to their description.

In the motional dataset, there are eight distinct attribute nodes:

- vehicle.moving: Only for vehicles with four or more wheels. A moving vehicle.
- vehicle.stopped: Only for vehicles with four or more wheels. A stopped vehicle with no immediate intent to move.
- vehicle.parked: Only for vehicles with four or more wheels. A stopped vehicle that has an intent to move.
- cycle.with_rider: Only for bicycles and motorcycles. Has a rider on it.
- cycle.without_rider: Only for bicycles and motorcycles. Has no rider on it.
- pedestrian.sitting_lying_down: Only for humans. Sitting or lying down.
- pedestrian.standing: Only for humans. Standing.
- pedestrian.moving: Only for humans. Moving.

In the Lyft dataset, the attribute nodes are more fine-grained:

- is_stationary: A stationary object
- object_action_abnormal_or_traffic_violation: An object behaving abnormally or committing a traffic violation
- object_action_driving_straight_forward: A vehicle driving straight forward
- object_action_gliding_on_wheels: A vehicle gliding on wheels
- object_action_lane_change_left: A vehicle changing lanes to the left
- object_action_lane_change_right: A vehicle changing lanes to the right
- object_action_left_turn: A vehicle turning left
- object_action_loss_of_control: A vehicle losing control
- object_action_other_motion: A vehicle performing some other motion

- object_action_parked: A parked vehicle
- object_action_reversing: A vehicle driving in reverse
- object_action_right_turn: A vehicle turning right
- object_action_running: A person running
- object_action_sitting: A person sitting
- object_action_standing: A person standing
- object_action_stopped: An object that is stopped
- object_action_u_turn: A vehicle performing a U-turn
- object_action_walking: A person walking

As we can see, the differences in the description of the instances and attributes constitutes a challenge from a user perspective.

Nevertheless, the combination of these two datasets can provide valuable insights for researchers in the domain of autonomous driving. It should be possible to use both these datasets in the process of researching and designing autonomous driving processes. Moreover, we should be able to provide the users with a holistic view of the data in the domain.

For this purpose, we have identified five representative queries to demonstrate the viability of our approach to achieve this goal. These queries are in part to ensure data quality, and in part to detect scenes which provide a challenge for autonomous vehicles, or the machine learning models they act on. For example, the same pedestrian should not be annotated as two different instances. In addition, these queries identify scenes that include safety-critical scenarios automated driving models should be tested against. More specifically, the following queries should be answered over the VVR property graph:

Q1 Which instances are pedestrians?

Q2 Which pedestrians disappear from a scene, and have another pedestrian appear some time after?

Q3 In which samples do two different humans appear at consecutive times?

Q4 Which pedestrians change from moving to being stationary?

Q5 Which pedestrians change from being stationary to moving?

Each of these queries can be formulated as a Ξ-restricted query. The relation NEXT in the property graph signifies objects or events which are consecutive in time, and therefore the relation is acyclic by nature. Below, we present the above queries in our logic notation:

Q1 $q(x) \leftarrow \mathsf{pedestrian}(x)$

Q2 $q(x, xp) \leftarrow \mathsf{pedestrian}(x), \mathsf{LAST\_ANNOTATION}(x, y), \mathsf{OF}(y, z), \mathsf{sample}(z),$
$\mathsf{NEXT} \cdot \mathsf{NEXT}^*(z, zp), \mathsf{OF}(yp, zp), \mathsf{FIRST\_ANNOTATION}(xp, yp), \mathsf{pedestrian}(xp)$

Q3 $q(z, zp) \leftarrow \mathsf{pedestrian}(x), \mathsf{LAST\_ANNOTATION}(x, y), \mathsf{OF}(y, z), \mathsf{sample}(z),$
$\mathsf{NEXT} \cdot \mathsf{NEXT}^*(z, zp), \mathsf{OF}(yp, zp), \mathsf{FIRST\_ANNOTATION}(xp, yp), \mathsf{pedestrian}(xp)$

Q4 $q(x) \leftarrow \mathsf{pedestrian}(x), \mathsf{OF}(y, x), \mathsf{HAS}(y, z), \mathsf{pedestrian\_moving}(z),$
$\mathsf{NEXT} \cdot \mathsf{NEXT}^*(y, yp), \mathsf{HAS}(yp, zp), \mathsf{pedestrian\_stationary}(zp)$

Q5 $q(x) \leftarrow \mathsf{pedestrian}(x), \mathsf{OF}(y, x), \mathsf{HAS}(y, z), \mathsf{pedestrian\_stationary}(z),$
$\mathsf{NEXT} \cdot \mathsf{NEXT}^*(y, yp), \mathsf{HAS}(yp, zp), \mathsf{pedestrian\_moving}(zp)$

6.3 Adding Domain-Specific Knowledge

To relate the labels of the categories and attributes of the two datasets together, we designed a simple ontology. It contains all the labels in the VVR property graph described in the previous chapter as basic concepts. In addition, we defined the following role names in the ontology: OF, HAS, FIRST, LAST, NEXT, FIRST_ANNOTATION, and LAST_ANNOTATION.

As we have already mentioned, categories are organized in a hierarchy. Similarly, the attributes can also be organized hierarchically. Therefore, our ontology is a concept taxonomy. For better readability, we show the taxonomy for categories in Figure 6.2, and the one for attributes in Figure 6.3. Still, these two (disjoint) taxonomies are part of the same ontology.

Using only taxonomies does not utilize the full expressive power of DL-Lite. However, given with how the data is structured in the use case, the benefits of a more complex DL-Lite ontology are not apparent for the queries considered so far. The data contains only few, generic role names. For example, the relation OF is used between many different concepts. If there were more specific role names present in the schema, then more specific axioms and role inclusions could be added in a meaningful way to the ontology, enabling us to formulate more interesting queries.

6.4 Rewriting

We executed the query rewriting algorithm on the queries Q1-Q5 with regard to the ontology described in the previous section. In addition, we recorded the time needed to perform the rewritings on our machine. We used a Windows 10 machine with a 6-core AMD Ryzen 5 3600 CPU running at 3.6 GHz and 32 GB of RAM.

Note that the rewriting is independent from the underlying data i.e., the Cypher queries can be executed on any database that accepts the Cypher query language. Therefore, the number of queries in the union of C2RPQs returned by our rewriting and the time

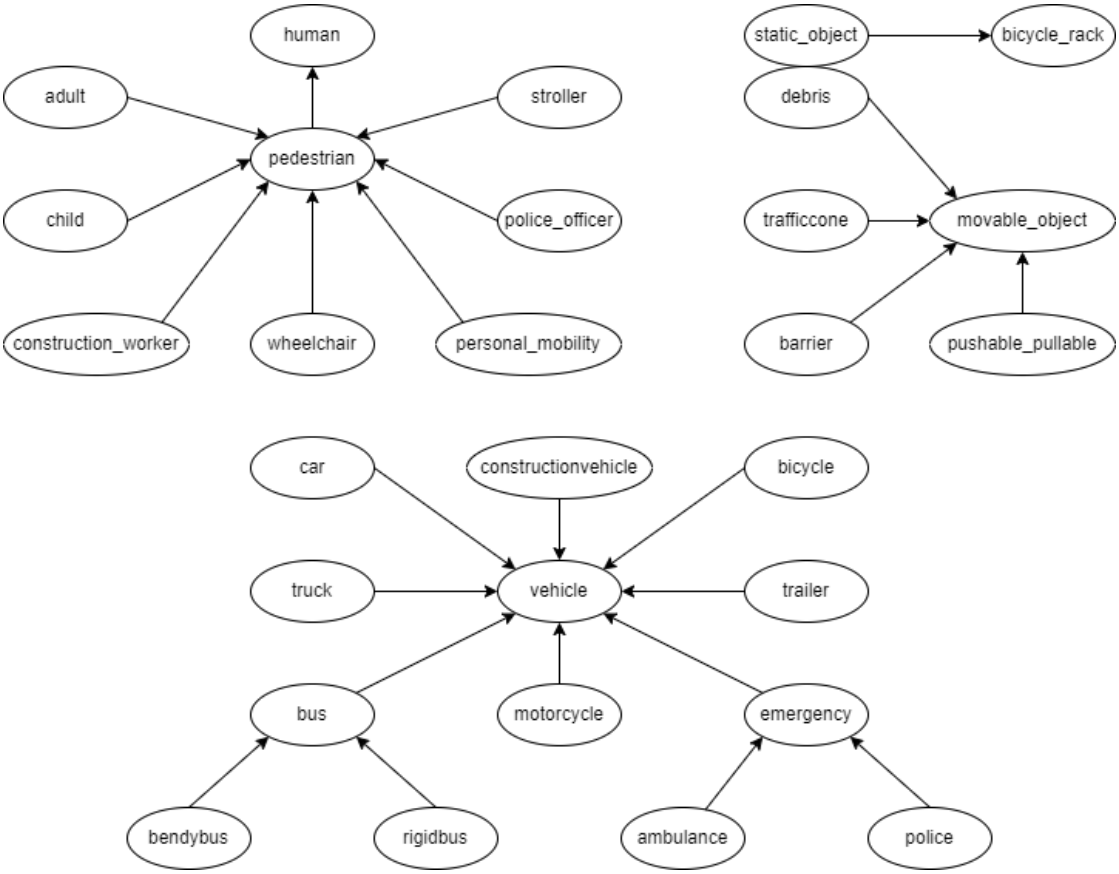

Figure 6.2: Category taxonomy

needed to generate the rewritings is also independent from the data. Hence, when we discuss the size of the rewritings and the time needed to generate them, we only take the ontology into account.

Query	Size of UC2RPQ	Time
Q1	8	3
Q2	880	1112
Q3	880	922
Q4	1056	1152
Q5	1056	1168

Table 6.2: Summary of query sizes and rewriting execution times (ms)

Table 6.2 depicts the sizes of the queries rewritten by our implementation. While the rewriting of the first query consists only of a set of eight queries, the other four queries have 880 and 1056 different queries in their rewriting. For the first query, we can easily

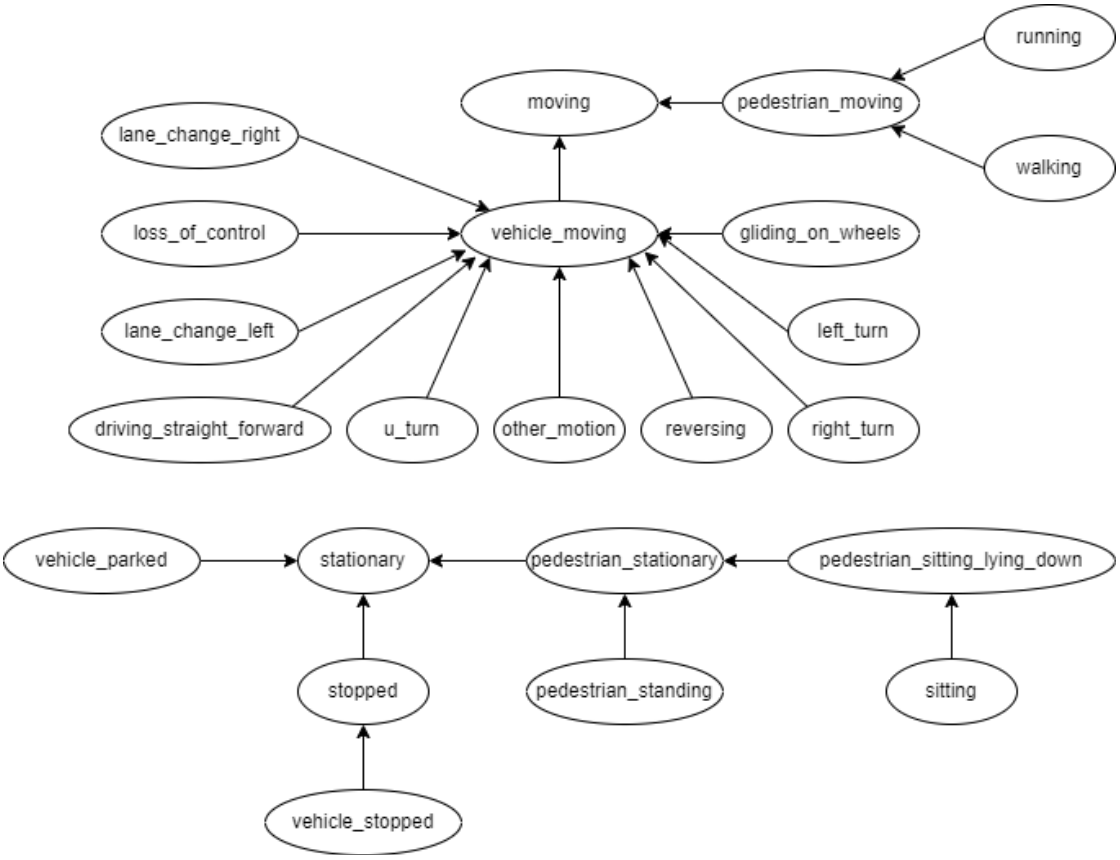

Figure 6.3: Attribute taxonomy

deduce the number of queries from the taxonomy shown in Figure 6.2. The concept pedestrian has seven distinct sub-concepts. Hence, for each sub-concept, our rewriting algorithm introduces a new query. For example, adult is a sub-concept of pedestrian. Therefore, our rewriting will add the query $q(x) \leftarrow \mathsf{adult}(x)$ to the set of queries.

However, as we have also discussed in Section 4.4, the number of queries returned by our procedure may be exponential in the number of atoms. For example, consider a "simpler" version of query Q2: $q(x, y) \leftarrow \mathsf{pedestrian}(x), \mathsf{pedestrian}(y)$. It is easy to see that a complete and correct rewriting must contain at least $8 \times 8 = 64$ queries for each possible combination of replacing the atoms of pedestrian with a sub-concept. Thus, it is not surprising that the number of queries contained in the rewritings of queries Q2-Q5 are much larger than the number of queries in the rewriting of Q1.

6.5 Query Evaluation

We evaluated the queries on a database constructed from a sample of the Lyft[2] and nuScenes [3] datasets, which are available for public download. The VVR property graph can be constructed in Neo4j from these data by translating the associated metadata to the graph schema shown in Figure 6.1 and following the descriptions laid out in Section 6.2. For example, the creation of the relations FIRST_ANNOTATION and LAST_ANNOTATION is shown in Figure 6.4. Regarding the category labels, we show an example query for adding the bicycle label in Figure 6.5.

```
1 // create edge from instance to the first annotation
2 match (x:instance)<-[:OF]-(y:sample_annotation)
3 where x.first_annotation_token = y.token
4 create (x)-[:FIRST_ANNOTATION]->(y)
5
6 // create edge from instance to the last annotation
7 match (x:instance)<-[:OF]-(y:sample_annotation)
8 where x.last_annotation_token = y.token
9 create (x)-[:LAST_ANNOTATION]->(y)
```

Figure 6.4: Creation of the relations FIRST_ANNOTATION and LAST_ANNOTATION

```
1 match (a:category {name: 'vehicle.bicycle'})<-[:OF]-(x:instance)
2 set x:bicycle
```

Figure 6.5: Creation of the appropriate labels for bicycle

The property graph we created this way contains 91.891 nodes and 254.555 distinct relations. Table 6.3 shows the result sizes (number of tuples) and time needed to execute the queries. We can see that even if the result sizes are not large, the queries need a lot of time to be executed because of the number of queries contained in the rewritings.

6.6 Summary

In this chapter, we have shown how ontologies can be utilized in a real-life use case to query data from heterogeneous sources in a property graph database setting. We could answer queries with regard to ontologies with our rewriting procedure. Still, the

[2] https://level-5.global/download/, accessed 17th May, 2022
[3] https://www.nuscenes.org/nuscenes#download, accessed 17th May, 2022

Query	Result size	Time
Q1	234	1
Q2	751	111071
Q3	264	109209
Q4	10	136767
Q5	8	143516

Table 6.3: Summary of Cypher query execution times (ms)

evaluation of our representative sample of queries has shown that the possibly exponential number of queries in the rewriting can occur in practice.

The data in presented in our use case have been complete in the sense that there were no entities missing from the data. This is reflected by the fact that the ontology we designed is a taxonomy. However, the DL-Lite ontology our framework supports can also reason about missing data by using axioms with existentials on the right-hand side. Moreover, if ontologies are intended to be used at the start of the design of an OMQ system, some data preparation may become redundant. Basic concepts or roles between individuals could be inferred from the ontology, removing the need to explicitly add them to the data.

Finally, ontologies can make integration of data into standardized processes easier. For example, if a new standard for describing data is released in the form of an ontology, then the integration of existing data can be facilitated through this ontology. In the autonomous driving domain, such standardization efforts are heralded by the Association for Standardization of Automation and Measuring Systems (ASAM).

CHAPTER 7

Conclusion and Future Work

In our work, we have presented a framework for querying property graphs with ontologies. This included defining a query and ontology language, as well as developing a new rewriting procedure to answer those queries with Cypher over a Neo4j database.

We conclude our work by answering the research questions from Section 1.3. In our discussion of the answers to the research questions, we will point to the relevant sections of our thesis where they were addressed. Finally, we present possible improvements to our approach, as well as extensions to our framework and possible avenues for further research on OMQ for property graphs.

7.1 Research Questions

Our research questions aim at utilizing the underlying graph database management system as much as possible. We aimed to refer much of the query answering to the underlying DBMS, where we can make use of its query optimization techniques which have been developed for more than a decade. Moreover, one of our goals was to make the different query semantics of ontologies, SPARQL 1.1 and Cypher compatible.

RQ1: What is an appropriate way to enable OMQ of property graphs?

In our work, we have presented a framework capable to add domain knowledge from a DL-Lite ontology to the answers of a Cypher query. Users can pose a query using the vocabulary of the ontology and receive a Cypher query which can be evaluated over a Neo4j graph database. Moreover, we allow navigational features in addition to conjunctive queries in our query language.

We defined a query language in Chapter 3 which can be expressed in Cypher and SPARQL 1.1. Our query language is capable of expressing conjunctive queries, and also

has navigational features. In addition, we defined an appropriate semantics for the query language.

Queries in our framework can be rewritten into a set of queries which can be evaluated over the plain data, thus eliminating the need to complete the data with domain knowledge. We described this rewriting technique in Chapter 4, and showed that the complexity of query answering in our framework does not exceed the complexity of query answering in graph databases.

One caveat of our approach is that the roles used in paths must be acyclic in the data, and the TBox must adhere to certain syntactic restrictions such that the returned answers are the same in SPARQL 1.1 and Cypher semantics. Nevertheless, confirming that these restrictions hold do not make query answering harder i.e., they do not increase the computational complexity of our approach.

We provide a description of our implementation our rewriting algorithm in Chapter 5, which was used to answer queries using data from a real-life use case presented in Chapter 6. The evaluation of the queries showed that computing the rewriting consumed less time than evaluating the queries in the database. This was expected, as our rewriting can produce an exponential number of queries, which makes query evaluation in databases time-consuming.

RQ2: How do Cypher semantics affect the answers of navigational queries?

In Section 2.3.2, we have extensively discussed the semantics of query languages in our framework. Cypher follows no-repeated-edges semantics, compared to the homomorphism-based semantics used in ontologies and SPARQL 1.1. We have shown that Cypher queries can be formulated in such a way that the difference to homomorphism-based semantics is only in the definition of matches for paths. From our definition of no-repeated-edges semantics it also follows that any no-repeated-edges match is a homomorphism-based match. However, the converse does not hold i.e., not every homomorphism-based match is a no-repeated-edges match.

In Lemma 2.4.1, we proved that no-repeated-edges matches are not preserved under homomorphisms, showing that the certain answer semantics from ontology-mediated querying is not appropriate for no-repeated-edges semantics. Thus, we demonstrated the need not only for restrictions on the knowledge base, but also for an alternative semantics for ontology-mediated querying in graph databases with Cypher. We defined such syntactic restrictions on the knowledge base in 3.1 and shown that under those restrictions, the answers coincide for the canonical model of the knowledge base. Hence, we defined the no-repeated-edges semantics certain answers to queries in our knowledge base as the answers in the canonical model.

RQ3: What is a suitable rewriting strategy using Cypher as a target language?

We presented a rewriting algorithm for queries in our framework and DL-Lite ontologies in Chapter 4 (Algorithm 4.1). Furthermore, we proved the correctness and completeness of our rewriting approach, in addition to termination. The number of queries generated by our rewriting procedure is exponential in the number of atoms in the input query. However, this is already the case for conjunctive queries. Still, rewriting is independent of the data, because it only requires a query and an ontology as an input. Therefore, we demonstrated that it is a viable approach to rewriting in our framework.

RQ4: How feasible are Cypher rewritings for OMQ of property graphs?

Based on our rewriting, we designed a query answering procedure for our framework (Algorithm 4.3). Additionally, we discussed the computational complexity of query answering in our framework. We showed that query answering with regard to ontologies does not increase the computational complexity of query answering in graph databases. In particular, query answering in our framework is in NL in data complexity for no-repeated-edges set semantics for paths. The combined complexity of evaluating queries is not harder than evaluating C2RPQs with Cypher, but still NP-complete.

In Chapter 6, we presented the application of our rewriting on a use case from the autonomous driving domain. We used an ontology to query data from two sources with different vocabularies. Our rewritings could be executed over the plain data in Neo4j, and returned the answers in reasonable time compared to the number of queries in the rewriting.

7.2 Future Work

The framework we have presented in this thesis opens a wide variety of interesting questions which can be subject to additional research. In the following, we will describe three such questions of which we deem that they would most improve the usability of our approach in practice.

Query Containment

Query containment describes a classical problem in database theory [AHV95]. The goal of the query containment problem is to decide whether the answers to a query q_1 are contained in the answers to a query q_2 in every database. Consider our rewriting from Example 4.1.6, which contains the two queries $q_1() \leftarrow r(x, \_), s(x, x)$ and $q_2() \leftarrow r(x, \_), s^*(x, x)$. We can see that q_1 is contained in q_2, as q_2 returns the empty tuple if q_1 returns the empty tuple. Because the rewriting is a union of C2RPQs, we do not lose any answers if we remove q_1 from our set of queries.

As illustrated by our example, the rewriting technique we presented in Chapter 4 does not necessarily produce a minimal set of queries, as is the case for PerfectRef [CGL+07]. In addition, we did not consider any optimizations or reductions of the rewriting in our implementation described in Chapter 5. Thus, one possible improvement would be to study query containment for DL-Lite ontologies and Ξ-restricted C2RPQs, and apply it to the output of our rewritings.

Querying Properties

Property graphs distinguish themselves from other graph-structured data models by adding property keys and data values to their model. Each semantic object, such as a node in a graph, can be adorned with a set of property (attribute) keys and corresponding values. For example, the Neo4j proprietary model (Definition 2.1.1) allows to not only add properties to nodes, but also to relations. This way, property graphs can model relations with arities greater than one. Consider a relation friendOf between Person objects, where we also store where the two parties met (met) and since when they have been friends (friends_since). In a Neo4j property graph, this can easily be added to the relation object itself via a property key and corresponding value. On the other hand, an RDF representation would not be as straightforward. One possibility to model such a relation in RDF would be to add a new object that represents the relationship, then add triples to relate the two Person objects to the relationship object, and finally add triples with the relationship object to store the additional information.

Thus, extending our framework to account for property values in property graph databases would be a useful feature. Indeed, if we recall our definition of interpretations (Definition 2.2.5), we consider both nodes and relations in the graph as semantic objects. Hence, adding property keys and values should be possible for our interpretations. Still, we would also need to extend our query language and semantics to account for this extension. A simple way to include property-value pairs for individuals would be to interpret them as special binary atoms. However, we did not include any atoms that use relation objects directly in our query language. Hence, querying the properties of relations, and by extension paths, is a more intricate problem that requires more care, especially if domain knowledge about properties should be included.

Extending the Ontology Language

Related to the querying of properties in the underlying graph database is an extension of the ontology language which uses property values in its concepts. Kharlamov et al. [KKM+16] defined the ontology language DL-Lite$^{\mathcal{A}}_{agg}$, which uses *aggregation* of property values in its concepts. The use of concepts based on the aggregation of property values has proven useful in the use case from Siemens that they discussed in their work. Thus, using such an ontology language could also be suitable for property graph databases.

List of Figures

2.1 Basic structure of a Cypher query 18

4.1 Cypher query to check acyclicity of roles 48
4.2 Cypher query to return all tuples of arity two 48

5.1 Implementation Workflow 51
5.2 UML class diagram of Terms 53
5.3 UML class diagram of atoms and related elements 54
5.4 UML class diagram for paths and path elements 55
5.5 UML class diagram of rewritable atoms 55
5.6 Java example for parsing a query string with ANTLR 57
5.7 ANTLR 4 grammar for our query language 58
5.8 Cypher translation of queries with role names and their inverses 60
5.9 Example OWL ontology in RDF format 62
5.10 Cypher result of rewriting 63

6.1 The dataset graph schema 67
6.2 Category taxonomy 72
6.3 Attribute taxonomy 73
6.4 Creation of the relations FIRST_ANNOTATION and LAST_ANNOTATION 74
6.5 Creation of the appropriate labels for bicycle 74

List of Tables

4.1 PerfectRef rewritings with disjunctions, "_" denote unbound variables . . . 37

5.1 Rewriting of Ξ-restricted query atoms to Cypher 60
5.2 Implementation dependencies 60

6.1 Categories and associated instance labels 68
6.2 Summary of query sizes and rewriting execution times (ms) 72
6.3 Summary of Cypher query execution times (ms) 75

List of Algorithms

4.1 Rewrite 35
4.2 SaturatePaths 36
4.3 Answer 49

Bibliography

[AAB+17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of Modern Query Languages for Graph Databases. *ACM Computing Surveys*, 50(5), September 2017.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[ASM22] Lakshya A. Agrawal, Nikunj Singhal, and Raghava Mutharaju. A SPARQL to Cypher Transpiler: Proposal and Initial Results. In Gargi Dasgupta, Yogesh Simmhan, Balaji Vasan Srinivasan, Sourav Bhowmick, Amith Singhee, Maya Ramanath, Nipun Batra, and Abhinandan S. Prasad, editors, *CODS-COMAD 2022: 5th Joint International Conference on Data Science & Management of Data (9th ACM IKDD CODS and 27th COMAD), Bangalore, India, January 8 - 10, 2022*, pages 312–313. ACM, 2022.

[BBXK16] Stefan Brüggemann, Konstantina Bereta, Guohui Xiao, and Manolis Koubarakis. Ontology-based data access for maritime security. In Harald Sack, Eva Blomqvist, Mathieu d'Aquin, Chiara Ghidini, Simone Paolo Ponzetto, and Christoph Lange, editors, *The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings*, volume 9678 of *Lecture Notes in Computer Science*, pages 741–757. Springer, 2016.

[BCC+19] Elena Botoeva, Diego Calvanese, Benjamin Cogrel, Julien Corman, and Guohui Xiao. Ontology-based Data Access – Beyond Relational Sources. *Intelligenza Artificiale*, 13(1):21–36, 2019.

[BHLS17] Franz Baader, Ian Horrocks, Carsten Lutz, and Uli Sattler. *An Introduction to Description Logic*. Cambridge University Press, 2017.

[Bie16] Meghyn Bienvenu. Ontology-Mediated Query Answering: Harnessing Knowledge to Get More from Data. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, page 4058–4061. AAAI Press, 2016.

[BMT17] Angela Bonifati, Wim Martens, and Thomas Timm. An Analytical Study of Large SPARQL Query Logs. *Proc. VLDB Endow.*, 11(2):149–161, oct 2017.

[BMT19] Angela Bonifati, Wim Martens, and Thomas Timm. Navigating the Maze of Wikidata Query Logs. In *WWW 2019 - The World Wide Web Conference*, pages 127–138, San Francisco, United States, May 2019. ACM.

[BOS15] Meghyn Bienvenu, Magdalena Ortiz, and Mantas Simkus. Regular Path Queries in Lightweight Description Logics: Complexity and Algorithms. *Journal of Artificial Intelligence Research*, 53:315–374, 2015.

[CBL+20] Holger Caesar, Varun Kumar Reddy Bankiti, Alex Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuScenes: A Multimodal Dataset for Autonomous Driving. In *CVPR*, pages 11618–11628, 06 2020.

[CDGL+11] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Antonella Poggi, Mariano Rodriguez-Muro, Riccardo Rosati, Marco Ruzzi, and Domenico Fabio Savo. The MASTRO System for Ontology-Based Data Access. *Semantic Web*, 2:43–53, 01 2011.

[CGL+07] Diego Calvanese, Giuseppe Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family. *J. Autom. Reason.*, 39(3):385–429, October 2007.

[CKNT08] Diego Calvanese, Evgeny Kharlamov, Werner Nutt, and Camilo Thorne. Aggregate Queries over Ontologies. In Ramez Elmasri, Martin Doerr, Mathias Brochhausen, and Hyoil Han, editors, *Proceedings of the 2nd International Workshop on Ontologies and Information Systems for the Semantic Web, ONISW 2008, Napa Valley, California, USA, October 30, 2008*, pages 97–104. ACM, 2008.

[CLW14] Richard Cyganiak, Markus Lanthaler, and David Wood. RDF 1.1 Concepts and Abstract Syntax. W3C recommendation, W3C, February 2014. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[CS16] Nicolle Chaves Cysneiros and Ana Carolina Salgado. Including hierarchical navigation in a Graph Database query language with an OBDA approach. In *32nd IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2016, Helsinki, Finland, May 16-20, 2016*, pages 109–114. IEEE Computer Society, 2016.

[CS17] Jaime Castro and Adrián Soto. A Comparison between Cypher and Conjunctive Queries. In Juan L. Reutter and Divesh Srivastava, editors, *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of*

Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017, volume 1912 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[DCS12] Souripriya Das, Richard Cyganiak, and Seema Sundara. R2RML: RDB to RDF Mapping Language. W3C recommendation, W3C, September 2012. https://www.w3.org/TR/2012/REC-r2rml-20120927/.

[DKT16] Georgios Drakopoulos, Andreas Kanavos, and Athanasios K. Tsakalidis. Evaluating Twitter Influence Ranking with System Theory. In Tim A. Majchrzak, Paolo Traverso, Valérie Monfort, and Karl-Heinz Krempels, editors, *Proceedings of the 12th International Conference on Web Information Systems and Technologies, WEBIST 2016, Volume 1, Rome, Italy, April 23-25, 2016*, pages 113–120. SciTePress, 2016.

[FGBH20] Naglaa Fathy, Walaa Gad, Nagwa Badr, and Mohammed Hashem. Querying Heterogeneous Property Graph Data Sources Based on a Unified Conceptual View. In *Proceedings of the 2020 9th International Conference on Software and Information Engineering (ICSIE)*, ICSIE 2020, page 113–118, New York, NY, USA, 2020. Association for Computing Machinery.

[FGG+18] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1433–1445. ACM, 2018.

[Har17] Olaf Hartig. Foundations of RDF⋆ and SPARQL⋆ (An Alternative Approach to Statement-Level Metadata in RDF). In Juan L. Reutter and Divesh Srivastava, editors, *Proceedings of the 11th Alberto Mendelzon International Workshop on Foundations of Data Management and the Web, Montevideo, Uruguay, June 7-9, 2017*, volume 1912 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[HMG+12] Ian Horrocks, Boris Motik, Bernardo Cuenca Grau, Zhe Wu, and Achille Fokoue. OWL 2 Web Ontology Language Profiles (Second Edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/.

[Hor11] Horridge, Matthew and Bechhofer, Sean. The owl api: A java api for owl ontologies. *Semant. Web*, 2(1):11–21, jan 2011.

[KBL+21] Tahir Emre Kalaycı, Bor Bricelj, Marko Lah, Franz Pichler, Matthias K. Scharrer, and Jelena Rubeša-Zrim. A Knowledge Graph-Based Data Integration Framework Applied to Battery Data Management. *Sustainability*, 13(3), 2021.

[KHS+17] Evgeny Kharlamov, Dag Hovland, Martin G. Skjæveland, Dimitris Bilidas, Ernesto Jiménez-Ruiz, Guohui Xiao, Ahmet Soylu, Davide Lanti, Martin Rezk, Dmitriy Zheleznyakov, Martin Giese, Hallstein Lie, Yannis E. Ioannidis, Yannis Kotidis, Manolis Koubarakis, and Arild Waaler. Ontology based data access in statoil. *J. Web Semant.*, 44:3–36, 2017.

[KK17] Holger Knublauch and Dimitris Kontokostas. Shapes Constraint Language (SHACL). W3C recommendation, W3C, July 2017. https://www.w3.org/TR/2017/REC-shacl-20170720/.

[KKM+16] Evgeny Kharlamov, Yannis Kotidis, Theofilos Mailis, Christian Neuenstadt, Charalampos Nikolaou, Özgür Özcep, Christoforos Svingos, Dmitriy Zheleznyakov, Sebastian Brandt, Ian Horrocks, Yannis Ioannidis, Steffen Lamparter, and Ralf Möller. Towards Analytics Aware Ontology Based Access to Static and Streaming Data (Extended Version), 2016.

[KKZ11] Stanislav Kikot, Roman Kontchakov, and Michael Zakharyaschev. On (In)Tractability of OBDA with OWL 2 QL. In Riccardo Rosati, Sebastian Rudolph, and Michael Zakharyaschev, editors, *Proceedings of the 24th International Workshop on Description Logics (DL 2011), Barcelona, Spain, July 13-16, 2011*, volume 745 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[KKZ12] Stanislav Kikot, Roman Kontchakov, and Michael Zakharyaschev. Conjunctive Query Answering with OWL 2 QL. In Gerhard Brewka, Thomas Eiter, and Sheila A. McIlraith, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, Rome, Italy, June 10-14, 2012*. AAAI Press, 2012.

[KLT+10] Roman Kontchakov, Carsten Lutz, David Toman, Frank Wolter, and Michael Zakharyaschev. The Combined Approach to Query Answering in DL-Lite. In Fangzhen Lin, Ulrike Sattler, and Miroslaw Truszczynski, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010*. AAAI Press, 2010.

[KUH+19] R. Kesten, M. Usman, J. Houston, T. Pandya, K. Nadhamuni, A. Ferreira, M. Yuan, B. Low, A. Jain, P. Ondruska, S. Omari, S. Shah, A. Kulkarni, A. Kazakova, C. Tao, L. Platinsky, W. Jiang, and V. Shet. Level 5 Perception Dataset 2020. `https://level-5.global/level5/data/`, 2019.

[LRS+16] Artem Lysenko, Irina A. RoznovăŢ, Mansoor Saqi, Alexander Mazein, Christopher J. Rawlings, and Charles Auffray. Representing and querying disease networks using graph databases. *BioData Mining*, 9(1):23, Jul 2016.

[MDFZM17] Franck Michel, Loïc Djimenou, Catherine Faron Zucker, and Johan Montagnat. xR2RML: Relational and Non-Relational Databases to RDF Mapping Language. Research Report ISRN I3S/RR 2014-04-FR, CNRS, October 2017.

[MM82] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, apr 1982.

[MNT20] Wim Martens, Matthias Niewerth, and Tina Trautner. A Trichotomy for Regular Trail Queries. In Christophe Paul and Markus Bläser, editors, *37th International Symposium on Theoretical Aspects of Computer Science, STACS 2020, March 10-13, 2020, Montpellier, France*, volume 154 of *LIPIcs*, pages 7:1–7:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[MR20] Ezequiel José Veloso Ferreira Moreira and José Carlos Ramalho. SPARQLing Neo4J (Short Paper). In Alberto Simões, Pedro Rangel Henriques, and Ricardo Queirós, editors, *9th Symposium on Languages, Applications and Technologies (SLATE 2020)*, volume 83 of *OpenAccess Series in Informatics (OASIcs)*, pages 17:1–17:10, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[Neo21a] Neo4j Team. Neosemantics(n10s) User Guide. `https://neo4j.com/labs/neosemantics/4.3/`, October 2021.

[Neo21b] Neo4j Team. The Neo4j Cypher Manual v4.4. `https://neo4j.com/docs/cypher-manual/4.4/`, December 2021.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[PLC+08] Antonella Poggi, Domenico Lembo, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Riccardo Rosati. Linking Data to Ontologies. In Stefano Spaccapietra, editor, *Journal on Data Semantics X*, pages 133–173, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[PSPM12] Peter Patel-Schneider, Bijan Parsia, and Boris Motik. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[RLT+14] Alireza Rahimi, Siaw-Teng Liaw, Jane Taggart, Pradeep Ray, and Hairong Yu. Validating an ontology-based algorithm to identify patients with type 2 diabetes mellitus in electronic health records. *Int. J. Medical Informatics*, 83(10):768–778, 2014.

[SH13] Andy Seaborne and Steven Harris. SPARQL 1.1 Query Language. W3C recommendation, W3C, March 2013. https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[Var82] Moshe Y. Vardi. The Complexity of Relational Query Languages (Extended Abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 137–146, New York, NY, USA, 1982. Association for Computing Machinery.

[XCK+18] Guohui Xiao, Diego Calvanese, Roman Kontchakov, Domenico Lembo, Antonella Poggi, Riccardo Rosati, and Michael Zakharyaschev. Ontology-Based Data Access: A Survey. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, pages 5511–5519. International Joint Conferences on Artificial Intelligence Organization, 7 2018.

[XLK+20] Guohui Xiao, Davide Lanti, Roman Kontchakov, Sarah Komla-Ebri, Elem Güzel-Kalaycı, Linfang Ding, Julien Corman, Benjamin Cogrel, Diego Calvanese, and Elena Botoeva. The Virtual Knowledge Graph System Ontop. In Jeff Z. Pan, Valentina Tamma, Claudia d'Amato, Krzysztof Janowicz, Bo Fu, Axel Polleres, Oshani Seneviratne, and Lalana Kagal, editors, *The Semantic Web – ISWC 2020*, pages 259–277, Cham, 2020. Springer International Publishing.