Extending VKG Systems with RDF-star Support

by

Lukas Sundqvist

Supervisor: Prof. Diego Calvanese

Co-Supervisor: Dr. Benjamin Cogrel

A document submitted in partial fulfillment of the requirements for the degree of

MSc in Computational Data Science

at

Free University of Bozen-Bolzano

Acknowledgements

I would like to thank my supervisor, Professor Diego Calvanese, and Benjamin Cogrel. They have both served as invaluable advisors and tutors throughout the work of this thesis.

I also want to thank my classmates. Our “Fellowship of the data” kept studying fun and on track during the COVID-19 pandemic, and through the strict Italian lockdown.

And last but not least I would like to thank my loving partner and family for supporting me during this time.

Abstract

This thesis concerns an extension to Virtual Knowledge Graph (VKG) systems, a state of the art technique of data integration. These systems perform data integration through the mapping of relational data sources to a virtual knowledge graph. The graph is often expressed using the W3C standard of Resource Description Framework (RDF), in this case the graph contains data in the form of triples, also known as facts.

RDF-star is a proposed extension to RDF that enables the annotation of RDF triples. It allows for a further layer of contextual data that until now has been difficult to implement using standard RDF. As RDF-star is a current field of research with few publications to its name this thesis contains a description of the proposal.

We investigate how to extend VKG systems with RDF-star support in order to enhance their support for contextual data. To this end we identify the mapping as the VKG component most crucial to extend. This is the component that specifies how to generate the virtual knowledge graph from the data contained in the data sources.

In order to extend the mapping we propose an extension to the W3C standard mapping language R2RML that we call R2RML-star. This allows for generating RDF-star data from relational data sources, enhancing the system's virtual knowledge graph with all the benefits of RDF-star in regard to contextual data handling.

We detail all the ways a VKG system must be extended in order to handle our proposed extension. This includes support for querying the generated RDF-star data, which is done through the use of SPARQL-star, an extension of the W3C standard query language for RDF data, SPARQL.

As a proof-of-concept we implement our proposed changes in the state of the art VKG system Ontop. We describe in detail the challenges this posed and how we overcame them, hoping to serve as inspiration for further development in this field.

Contents

1 Introduction

In this chapter we introduce the reader to the thesis. We present the thesis' background, goals, and main contributions to research. At the end of this chapter we present an overview of the thesis' structure.

1.1 Virtual Knowledge Graphs

This section introduces the background, problem statement, and goals of the thesis. The full background can be read in the first non-introductory part of our work, Chapter 2.

Our work builds on the popular *Virtual Knowledge Graph* (VKG) approach to data integration and it is therefore necessary to introduce this approach in order to describe the remainder of the thesis. Most common data integration techniques in use today follow an approach where the data must be materialised into a new database. This is the case for the data lake and data warehouse settings, which see widespread use in organizations today [19].

Materialising the data sources into a new integrated repository comes with several drawbacks. For one, if the data sources change then they must once again be loaded into the integrated repository, which creates a risk of hosting outdated data if updates are not run frequently enough. Second, the materialisation duplicates data, which requires additional hardware resources. Alternatively the original data sources can be removed after integration, but this means the originals are lost. This is often not an option e.g. because of system or legal constraints. It also requires that the integrator has the relevant access to the data sources and permission to modify them, which is not always the case [1, 19]

A virtual approach to data integration leaves the data sources as they are and does not create a new one. Instead, an application runs as a middle layer between the data sources and the end user, allowing user posed queries to be executed over all the data sources and return integrated answers [19]. The Virtual Knowledge Graph approach is, as the name implies, one such virtual solution.

Using the VKG approach, data sources are integrated into a virtual graph database, a Knowledge Graph, that is queried by the end user using a graph query language. A *mapping* describes how data in the data sources are mapped to elements in the knowledge graph. Optionally an *ontology* is used to give ontological information and enrich query answers.

While running, the VKG system takes a user posed query and, if possible, *rewrites* it with regard to the supplied ontology in order to capture more answers. The query is then *unfolded* with regard to the given mappings and is then ready to be executed over the data sources in their respective query languages. After the answers have been retrieved, they are compiled and returned to the user. From the end user's perspective, they are just querying a graph. The actual functioning of the virtual approach is hidden from them.

1.1.1 The problem of contextual data

The VKG approach suffers from an issue regarding contextual data. The knowledge graph itself is often expressed using the W3C standard language of *Resource Description Framework*, or RDF for short[1]. In this language one data point is expressed as a triple of *subject*, *predicate*, and *object*. We call such a triple a *fact*.

The problem arises when we want to comment our data, or when we want to add context by annotating the facts. In standard RDF this is already supported but through a very cumbersome technique known as RDF reification [27]. This technique does not work well for the VKG approach, as we show in Section 3.1. Other techniques have been suggested to deal with the metadata issue including the use of singleton properties and named graphs. As shown by Hartig, these techniques have their own shortcomings and do not on their own solve the contextual data issue, motivating the development of a new solution [27].

RDF-star is such a solution, developed by Hartig and a W3C Community Working Group under his lead [15, 27]. It extends the RDF standard and allows embedding a full RDF-triple in the subject or object position of another triple. We detail the RDF-star extension and work related to it in Chapter 3. Let us proceed with an example to clear things up.

Example 1.1 We wish to express the fact that Lukas Sundqvist is a student of the Free University of Bozen-Bolzano. This can be represented by the triple:
(`:people/LukasSundqvist`, `:studentOf`, `:institutions/UniBZ`)
Here the subject of the triple is an object representing Mr Sundqvist, coincidentally the author of this thesis. The predicate is the property `:studentOf`, and the object is an object representing the university.

Imagine now that we want to annotate this fact with some context, for example its source. As I just wrote this triple down in the thesis I'll use myself as source. Using a concrete RDF-star syntax, the annotated triple could be expressed as:

```
<< :people/LukasSundqvist :studentOf :institutions/UniBZ >>
  :source :people/LukasSundqvist .
```

Here the original triple is *embedded* or *quoted* inside a root triple where it plays the role of subject. ♢

Improving the support for contextual data in RDF can have a great impact on VKG systems using RDF to express their knowledge graph. In a data integration scenario,

[1]https://www.w3.org/RDF/

contextual data plays an important role, for instance as in the example above one can annotate triples with a source property to indicate from which data source they were generated.

1.1.2 The VKG system Ontop

Ontop[2] is a actively developed state of the art VKG system [1, 2]. It is an open source project developed by researchers and Ontopic[3], a company offshoot of the Free University of Bozen-Bolzano. Ontop currently lacks any support for RDF-star, like all current VKG systems, but plays a large role in this thesis as proof of concept of our work.

The system uses W3C standards for all the parts of the VKG approach. RDF for the knowledge graph, R2RML for the mapping, and OWL2 QL and RDFS as ontology languages. This means that extending these standards to allow RDF-star to play a role in VKG systems allows us to also practically implement and extend Ontop.

R2RML is the W3C standard language for mapping data from relational databases to graphs, and one of the mapping languages supported by Ontop. It allows for mapping relational data to an RDF graph. The mapping itself consists of a set of mapping assertions, and each such assertion is split up into two parts: a source part that is a SQL query, and a target part consisting of *triple templates* showing how to generate RDF facts from the source data delivered through the SQL query. To our knowledge there is no research yet into extending this language to support the generation of RDF-star data.

1.1.3 Goals

Following from our description of the current state of the art in this field, there are three main goals of this thesis. The first is to give a complete description of RDF-star, the second is to investigate how to extend VKG systems with RDF-star support from a theoretical point of view, and the third is to implement this support in the VKG system Ontop.

1.2 Main Contributions

In this section we outline our main contributions to research.

1.2.1 RDF-star

The first main chapter details the current state of RDF-star. In it we expand on the motivation behind RDF-star followed by a description of RDF-star and related extensions to other standards.

[2]https://ontop-vkg.org/
[3]https://ontopic.ai/en/

We first describe RDF-star's abstract syntax as an extension of RDF's abstract syntax, followed by extensions to the concrete RDF syntaxes such as Turtle and N-Triples. We then present a section on SPARQL-star, the accompanying extension to the standard graph query language for RDF graphs, SPARQL. We touch on details of SPARQL-star and how it differs from SPARQL such as its changed basic definitions, its algebra, new functions and operators, and new answer formats.

1.2.2 Extending VKG systems with RDF-star support

The second main chapter contains our investigation into adding RDF-star support to VKG systems. We deduce which is the most important VKG component to extend, namely the mapping, and do so by extending the W3C standard language for mappings, R2RML. The first step is to expand the concept of mapping in the VKG context such that it allows for the generation of RDF-star triples, we do so by introducing the concept of *triple-star template*.

In a similar manner to how RDF-star extends RDF, we propose to extend R2RML by means of extending the target part of the mapping. Our extension necessarily breaks the R2RML standard, and mappings written in the language of our extension will not function in an unsupported system. This is actually a desirable property as it brings attention to the user that something is wrong and does not result in silent errors.

By means of a new vocabulary, the R2RML concepts of TripleMap and TermType are extended. We suggest a new RDF-star TermType that allows the user to describe in the mapping a template for generating RDF-star triples. The new mapping language, which we call *R2RML-star*, is designed to ensure that only valid RDF-star triples are created from the new type of template.

The new mapping language has effects on the query processing of the VKG system. The main parts of processing are the query rewriting and unfolding processes, and we detail how they are affected by the introduction of R2RML-star and how VKG systems are extended to handle this new functionality.

We describe how to extend VKG systems to support the accompanying standard of SPARQL-star. As a VKG system is a query answering system, the query language should support the knowledge graph's data model. If the graph is populated with RDF-star data, SPARQL-star support is required to correctly query it. Strictly speaking a VKG system can have various degrees of SPARQL-star support, having limited support limits the types of answers the end user can receive to their query. This is also detailed in the chapter.

In our view, a VKG system that supports SPARQL-star and R2RML-star mappings has been extended with RDF-star support. It allows the system to populate the knowledge graph with RDF-star contextual data and allows for querying the graph.

1.2.3 Extending Ontop with RDF-star support

Finally, much of the work done during this thesis has concerned a practical implementation of our suggested RDF-star extensions to the VKG system Ontop. This process is detailed in the third and last main chapter.

Based on our theoretical discussion on extending VKG systems, we detail how this was put in practice to implement the extensions in the open source Ontop project. Descriptions of Ontop's full query processing system are followed by a discussion on how to best change them to accommodate for the R2RML-star proposal. Internally in Ontop a query is represented with a tree like data structure called the Intermediate Query or IQ for short. The IQ is intended to bridge the gap between SPARQL and SQL queries and is the internal structure to which all query processing is applied. Much of our work has been focused on this data structure and how to extend its functionality so that rewriting and unfolding with regard to the R2RML-star proposal works as intended.

Through the discussion of this chapter we show that our implementation performs well and gives the expected results, this is verified through testing. The chapter is a valuable resource for learning about how to implement RDF-star support for other VKG systems, how to continue the work we have done on Ontop, or as a general developer's guide to Ontop for those interested in the systems internal workings.

1.3 Thesis Structure

Here follows an outline of the thesis to aid the reader to navigate its structure. The chapter you have just read is the introduction and it is followed by Chapter 2, the background chapter. In it we present to the reader the current state of research related to Virtual Knowledge Graphs. We present the relevant components of a VKG system, namely the ontology, mapping, knowledge graph, and graph query language. We also describe the relevant W3C standards including: RDF for the knowledge graph, R2RML for the mapping, RDFS and OWL2 QL for the ontology, and SPARQL for the query language.

The next three chapters make up the main chapters of the thesis. They begin with Chapter 3 on RDF-star. We explain the motivation behind RDF-star, its abstract and concrete syntaxes, and SPARQL-star in detail.

Chapter 4, investigating RDF-star support in VKG systems, is where we propose our RDF-star extension to VKG systems. We define the concept of RDF-star mapping and put it into practice by defining an extension to the mapping language R2RML that we call R2RML-star. We go through integrating SPARQL-star and how a VKG system must be adapted through all the stages of query processing to handle both SPARQL-star and R2RML-star mappings. We end on an example, showing the usage of our proposed extension.

In Chapter 5, dealing with the implementation of RDF-star support, we describe how we implemented this extension in the open source project Ontop. This includes details on the internal workings of Ontop and how functionality was added in order to handle SPARQL-star and R2RML-star mappings.

The last chapter is the conclusion, Chapter 6. Here our results are summarised and we discuss further work to be done in the field. Our references and appendices are then found at the end of the thesis. The appendices contain the full versions of RDF graphs and R2RML mappings used as examples throughout the thesis.

2 Background

In this section prerequisite knowledge is presented to the reader. Readers that are familiar with one or more of these subjects can freely skip sections.

The main prerequisite knowledge required for this thesis is an understanding of the Virtual Knowledge Graph (VKG) approach to data integration. To this end, we first explain all the components of a VKG system: the knowledge graph, query language, ontology and mapping; before finally explaining the system itself. We assume that the reader is familiar with basic notions of Computer Science, e.g. as taught in a Bachelor's degree in Computer science.

2.1 Resource Description Framework

The Resource Description Framework (RDF) is a framework to describe information on the web, standardized through a collection of W3C recommendations [4]. It is the data model underlying the Semantic Web and the VKG approach to data integration [1].

The basic building blocks of this data model are known as *RDF terms*. An RDF term is either an *IRI*, *blank node*, or *literal*. An IRI is a Unicode string with certain restrictions used to uniquely identify a resource on the web. An IRI looks very much like a URL. Blank nodes are local identifiers used in concrete RDF syntaxes if an identifier is unknown. Literals are values such as strings or integers that do not need to be uniquely identified [4].

An *RDF triple* consists of three components (*subject*, *predicate*, *object*). Seeing the subject and object as two nodes, the predicate forms a directed labeled edge from subject to object. Hence, a set of triples represents a directed edge-labeled graph [4].

Definition 2.1 Assume that $\mathcal{I}$ is the set of IRIs, $\mathcal{B}$ the set of blank nodes, and $\mathcal{L}$ the set of literals. An *RDF triple* is a tuple (s, p, o), consisting of a subject $s \in \mathcal{I} \cup \mathcal{B}$, a predicate $p \in \mathcal{I}$, and an object $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ [4]. ⧫

Definition 2.2 A set of RDF triples is an *RDF graph* [4]. ⧫

When writing RDF graphs, a concrete syntax is needed. For this background chapter, the Turtle syntax is always used unless another syntax is specified [14]. For further information on RDF it is recommended to read the RDF specification [4].

Example 2.3 Consider the IRIs:

- i_1 = `http://thesis.org/data/Lukas%20Sundqvist`
- i_2 = `http://thesis.org/knows`
- i_3 = `http://thesis.org/age`

Along with the blank node b_1 and the literal $l = 28$. Then (i_1, i_2, b_1) and (b_1, i_3, l) both form RDF triples. Together they form an RDF graph describing that the author of this thesis knows someone who's age is 28. ♢

2.2 Knowledge Graphs

Knowledge graphs have been vaguely defined by academics, practitioners, and companies for several years, which makes it difficult to pin down an exact definition. Färber et al. define knowledge graphs (KGs) simply as any graph written in the language of the Resource Description Framework (RDF) [18, p.1]. Ehrlinger on the other hand defines a KG as a system that "acquires and integrates information into an ontology and applies a reasoner to derive new knowledge." [17]. This second definition could nearly describe the entire VKG approach to data integration.

The difficulty of formally defining a knowledge graph is testament to the complexity of the term, nevertheless they deserve a proper definition. We will use as base the definition by Färber et al. and are inspired by the work of Ehrlinger [17, 18] in defining KGs within the scope of this thesis.

Definition 2.4 A *Knowledge Graph* is an RDF Graph describing objects of a particular domain and how they relate to each other. ♦

Even though KGs are never materialized in the VKG approach, they play a major role as this is the shape of the final data presented to the end user for querying. This leads us to the query language that the end user of a VKG system uses to retrieve data.

2.3 SPARQL

SPARQL is a standard used for querying RDF data, similar to the more familiar SQL used to query relational data. It is formalized in a W3C recommendation that forms the basis of this section, and the interested reader can look there for more information [5]. SPARQL is used to query an *RDF dataset*, which consists of one or more RDF graphs. One of these is the unnamed default graph and the others are named by IRIs [6].

2.3.1 SPARQL query language

The SPARQL query language is used to write queries posed to RDF datasets. Each query starts with two important parts, at the top are prefix declarations allowing terser syntax, and the second part sets the *query form*. SPARQL has four query forms which are:

- SELECT : Returns all or some of the variables bound.
- CONSTRUCT : Returns a new RDF graph based on a specified pattern.
- ASK : Returns a boolean indicating whether the query found a match or not.
- DESCRIBE : Returns an RDF graph describing the resources found [6].

Of these, the original, most used, and most important form is the SELECT query and that is the one we will be focusing on in this thesis. This form is roughly analogous to a SQL SELECT query and returns results in one of a number of answer formats, which will be further discussed in Section 2.3.2. A SELECT query returns all or a subset of variables bound during the pattern matching stage of executing the query.

The third part of the query is the WHERE clause, from where pattern matching is performed. This clause is analogous to a SQL WHERE clause with the main difference being that we are querying RDF data in the form of a graph in place of a relational database. Everything contained within the WHERE clause forms a *group graph pattern* that the dataset being queried has to fit in order to return results. The simplest group graph patterns are called *basic graph patterns* and they are made up only of *triple patterns*. Triple patterns consist of RDF terms and *SPARQL query variables* which get bound if part of the queried graph matches the pattern [6]. This is illustrated in Example 2.7. We will formally define most of these terms now, for further reading, see the SPARQL standard [5].

Definition 2.5 Let $\mathcal{I}$ be the set of IRIs, $\mathcal{V}$ the set of SPARQL query variables, and $\mathcal{L}$ the set of RDF literals. Then a *SPARQL triple pattern* is a tuple (s, p, o) with

$$s \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}, \quad p \in \mathcal{V} \cup \mathcal{I}, \quad o \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}.$$

Note that unlike the RDF standard, literals are allowed in the subject position [5].♦

Definition 2.6 A set of SPARQL triple patterns is a SPARQL Basic Graph Pattern (BGP) [5]. ♦

Group graph patterns can include more complex keywords such as OPTIONAL, JOIN, or FILTER, which complicates the semantics of the query. We refer the reader to the SPARQL standard for further information [5]. Following the WHERE clause is the fourth part of the query, containing familiar keywords analogous to their SQL counterparts, such as GROUP BY, ORDER BY, or LIMIT [6].

Example 2.7 Recall the RDF graph described in Example 2.3. Assume that it is the only and default graph forming our RDF dataset. Consider the SPARQL query:

```
PREFIX thesis: <http://thesis.org/>
SELECT ?person
WHERE {
  ?person thesis:knows _:b1.
  _:b1 thesis:age 27.}
```

Let us work through the query step by step. The PREFIX declaration allows us to save work writing the rest of the query. Wherever `thesis:` later appears it is replaced with the thesis IRI by the program executing the query. The SELECT clause signifies the query form and is followed by the variables that will be returned to the user. The WHERE clause specifies the group graph pattern to be matched, which in our case consists of two triples.

The triple patterns of the query match to the two triples of our RDF dataset binding the variable `?person`. There is only one match and therefore only one result is returned: `?person` is bound to `http://thesis.org/data/Lukas%20Sundqvist`.♢

2.3.2 Answer formats

The result of a SPARQL SELECT query is returned through one of several standardized formats representing mappings between the variables of the SELECT clause and the RDF-terms they have matched to [5]. These formats are:

- XML, the original format which is standardized in [7].
- JSON, standardized in [8]
- CSV and TSV, standardized in [10].

2.3.3 SPARQL algebra

The grammar and syntax of a SPARQL query are designed for ease of use and for humans writing queries. A translation into a more formal SPARQL algebra is possible and defined by the SPARQL standard [5]. The definitions and algebra in this section are mainly sourced from the seminal work by Pérez, Arenas, and Gutierrez [26]. We describe only a simplified SPARQL algebra, which is needed to understand the work done in this thesis. For additional reading please see the SPARQL standard and the paper by Pérez et al [5, 26]. To start, we define the notion of graph pattern expression.

Definition 2.8 We define a *SPARQL graph pattern expression* recursively as follows [26].

- A SPARQL triple pattern is a graph pattern expression.
- If P_1 and P_2 are graph pattern expressions, then $(P_1\ \texttt{AND}\ P_2)$, $(P_1\ \texttt{OPTIONAL}\ P_2)$, and $(P_1\ \texttt{UNION}\ P_2)$, are all graph pattern expressions.

- If P is a graph pattern expression and R is a SPARQL *built-in condition*, then $(P\,\texttt{FILTER}\,R)$ is a graph pattern expression.

A SPARQL built-in condition is a boolean function such as equality, bound, or isIRI. For further information we refer the reader to Pérez et al. and the SPARQL standard [5, 26]. ♦

Definition 2.9 A *SPARQL solution mapping* is a partial function μ mapping query variables to RDF terms, $\mu : \mathcal{V} \to \mathcal{I} \cup \mathcal{L} \cup \mathcal{B}$, where $\mathcal{B}$ is the set of blank nodes. The *domain* of μ is called $dom(\mu)$. We denote by $u(t)$ the triple obtained through substituting the variables in the triple pattern t according to the solution mapping μ. We say that μ_1 and μ_2 are *compatible* if $x \in dom(\mu_1) \cap dom(\mu_2)$ implies that $\mu_1(x) = \mu_2(x)$ [26]. ♦

Definition 2.10 Let Ω_1 and Ω_2 be sets of SPARQL solution mappings. The *join*, *union*, *difference*, and *left join* are defined as follows [26].

- $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \,|\, \mu_1 \in \Omega_1,\, \mu_2 \in \Omega_2 \text{ are compatible.}\}$.
- $\Omega_1 \cup \Omega_2 = \{\mu \,|\, \mu \in \Omega_1 \cup \Omega_2\}$.
- $\Omega_1 \setminus \Omega_2 = \{\mu \,|\, \forall \mu' \in \Omega_2,\, \mu \text{ and } \mu' \text{ are incompatible}\}$.
- $\Omega_1 ⟕ \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2)$. ♦

Definition 2.11 Let D be an RDF dataset, tp a triple pattern, and P_1 and P_2 graph pattern expressions. The *evaluation* of a graph pattern over D, call it $[\![\cdot]\!]_D$, is a function from the set of graph pattern expressions that returns a set of solution mappings. It is defined as follows [26].

- $[\![tp]\!]_D = \{\mu \,|\, dom(\mu) = var(tp),\, \mu(tp) \in D\}$, with $var(tp)$ being the set of variables occuring in tp.
- $[\![T_1\,\texttt{AND}\ T_2]\!]_D = [\![T_1]\!]_D \bowtie [\![T_2]\!]_D$.
- $[\![T_1\,\texttt{OPTIONAL}\ T_2]\!]_D = [\![T_1]\!]_D ⟕ [\![T_2]\!]_D$.
- $[\![T_1\,\texttt{UNION}\ T_2]\!]_D = [\![T_1]\!]_D \cup [\![T_2]\!]_D$. ♦

2.4 Relational Databases

Relational databases play an important role in this thesis but knowledge of them is assumed to be background knowledge to the reader. If this is not the case we recommend a classic book on database theory such as Abiteboul, Hull, and Vianu [21]. Nonetheless relational data normally makes up the data sources being integrated in the VKG approach and we will now set up an example that will be frequently used throughout the thesis.

IMDB database	imdbTable1	
Name	Year	Score
The Shawshank Redemption	1994	9.2
The Godfather	1972	9.2
Pulp Fiction	1994	8.9

Table 2.1: Description of the first database, containing data sourced from the Internet Movie Database.

Rotten Tomatoes database	RtTable 1	
MovieName	ReleaseYear	Rating
A Star is Born	2018	0.78
A Star is Born	1937	0.79
The Godfather	1972	0.98

Table 2.2: Description of the second database, containing data sourced from Rotten Tomatoes.

Example 2.12 Consider a movie theater holding information about movies in two different relational databases, each containing one table. The databases and the data stored therein are shown in Tables 2.1 and 2.2. We note that the primary key is composed of the Name and Year (or MovieName and ReleaseYear) columns, as the name alone is not a unique identifier.

The problem of data integration in this specific case concerns integrating the two databases so that we can query them both at the same time. ♢

2.5 Ontologies

The notion of *ontology* has its roots in philosophy, but we consider here the term as it is used in the field of Computer Science. In this field it represents the relevant notions in a domain or universe of interest, for example the domain of films which is our running example in this thesis.

Kaufmann (2012) defines an ontology as "a description of the entities in [a specific] domain, relationships between them, and any other known constraints." [19, p.325] and we will narrow this down further to define an ontology for the purpose of this thesis.

Specifically we consider ontologies that have their roots in the world of description logics. Such an ontology consists of an *A-Box* a *T-Box*, with the A and T representing assertions and terminology respectively. In the A-Box fit all those statements that describe individuals, while the T-Box contains all statements with general information about the domain. Using this language one splits an ontology in two sets of statements, assertions and terminology. For an example, imagine once again the Hollywood domain. The statement that Tom Hanks is an actor is an assertion and belongs in the A-Box, while the statement that directors direct movies is part of the T-Box [20].

When using the VKG approach for data integration, the ontology tends to only or mainly contain terminology, i.e. statements belonging to the T-Box. For the language describing the ontology, Ontop supports OWL2 QL or RDFS, both written in the form of an RDF graph [1, 22].

Definition 2.13 An *ontology* is an RDF graph describing the terminology of a certain domain, meaning abstract entities, relationships between them, and any other known constraints. The ontology is written using one or both of the OWL2 QL and RDFS ontology languages. ♦

2.5.1 RDFS

RDF Schema or RDFS is a semantic extension of RDF, which can also be seen as a lightweight ontology language. For more detail see the W3C specification [11]. RDFS allows for:

- Typing, to say that something is a resource or a class. Or that something is a member of a specific class.
- Subclasses, where A is subclass of B means that all objects of class A are also objects of class B.
- Specifying range and domain of properties. E.g. one can say that only humans have an age, and the age must be an integer.

Some example statements using rdfs, assuming prefixes already defined:

```
:Person rdf:type rdf:Class
:Man rdfs:subClassOf :Person.
:stars rdf:type rdf:Property.
:stars rdfs:domain :Films.
:stars rdfs:range :Actor
```

2.5.2 OWL2 QL

OWL2 is a much more expressive ontology language that allows many different types of statements to be made. For this thesis we only make use of OWL2 QL which is a subset of OWL2 developed specifically for database querying [13]. OWL2 QL allows for more detailed information than RDFS, such as some integrity constraints on properties. For example: one can specify that each Film must have a Director.

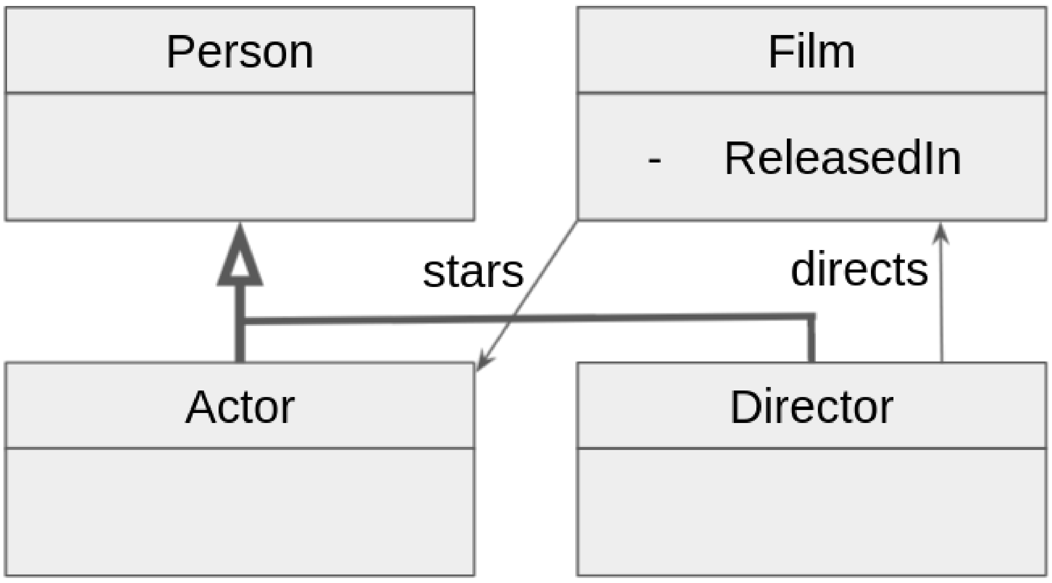


Figure 2.1: The ontology of Example 2.14 visualized as a UML diagram. Boxes represent classes, big arrows show subclass relations, small arrows are other relations between classes, and the ’releasedIn’ signifies a relation to a literal.

Example 2.14 Staying with our film example, imagine that we want to describe the domain surrounding film. To start, we could describe the following notions.

- There are different types of objects: Persons, films, actors, directors.
- There are relations between these objects: A film stars actors and is directed by directors. Clearly actors and directors are people.
- There are relations between the objects and literals, for example Films have a year of release. ♢

This bare-bone ontology is described visually in Figure 2.1. Here the ontology is expressed in the Turtle syntax using the RDFS and OWL2 QL ontology languages.

```
<http://lukas.thesis.org/films> rdf:type owl:Ontology .

# Object Properties
:directedBy rdf:type owl:ObjectProperty ;
        rdfs:domain :Film ;
        rdfs:range :Director .
:directs rdf:type owl:ObjectProperty ;
        rdfs:inverseOf :directedBy ;
        rdfs:domain :Film ;
        rdfs:range :Director .
:stars rdf:type owl:ObjectProperty ;
```

```
        rdfs:domain :Film ;
        rdfs:range :Actor .

# Data properties
:releasedIn rdf:type owl:DatatypeProperty ;
        rdfs:domain :Film .

# Classes
:Actor rdf:type owl:Class ;
        rdfs:subClassOf :Person .
:Director rdf:type owl:Class ;
        rdfs:subClassOf :Person .
:Film rdf:type owl:Class .
:Person rdf:type owl:Class .
```

2.5.3 SPARQL entailment

SPARQL entailment is a way to reason over SPARQL datasets based on a set of rules. Entailment allows a SPARQL query to return answers that are not directly present in the queried graph, but that are inferred based on the information contained in the ontology. We start with an example courtesy of the W3C [12].

Example 2.15 Consider the following RDF graph:

```
  ex:book1 rdf:type ex:Publication .
  ex:book2 rdf:type ex:Article .
  ex:Article rdfs:subClassOf ex:Publication .
  ex:publishes rdfs:range ex:Publication .
  ex:MITPress ex:publishes ex:book3 .
```

We wish to execute the following SPARQL query over the dataset consisting of the single RDF graph above:

```
  SELECT ?prop
  WHERE { ?prop rdf:type rdf:Property }
```

Under SPARQL rules as we know them this query will not match any data in the dataset and will return an empty answer, this is called *simple entailment.* However if the SPARQL query is executed under *RDF entailment* the system correctly deduces that `ex:publishes` is of type `rdf:Property`, seeing as it takes the place of predicate in the last line of the RDF graph. Next, consider the query:

```
  SELECT ?pub WHERE { ?pub rdf:type ex:Publication }
```

Under simple or RDF entailment this query has a single answer: `ex:book1`. If we run the query under *RDFS entailment* it will return two additional answers. Due to `ex:Article` being a subclass of `ex:Publication`, this implies that any articles, such as `ex:book2`, are also publications.

Also the third book, `ex:book3`, is returned as an answer under RDFS entailment. This is because it plays the role of object in a triple with `ex:publishes` as predicate, and because there is a RDFS range property specifying publishes to have publications as range. ♢

The example above shows how a query answering system can use basic reasoning to populate a knowledge graph with more data than is explicitly stated. This is the process of SPARQL entailment and it is one of the key contributions of the ontology in the VKG setting.

Entailment regimes

In this section we have mentioned many *entailment regimes.* These are various rules for interpreting ontological information and extending the queried dataset to allow SPARQL to match answers to variables as intended. Examples of such regimes are RDF, RDFS, or OWL entailment regimes. For further reading we recommend to the reader the W3C Recommendation on SPARQL entailment [12].

2.6 Virtual Knowledge Graphs

The core of this thesis concerns an extension to the Virtual Knowledge Graph system Ontop, thus it is necessary to have a good knowledge of what such a system is, in order to understand the coming chapters. We are concerned with the use of *Virtual Knowledge Graphs* (VKG) in the field of data integration, something that is also called *Ontology-Based Data Access* (OBDA) in the literature [3].

In the case of using a VKG for data integration the data sources are usually relational databases. The data is then *virtualised* with the help of a *mapping* and an *ontology*, in order to create a virtual *knowledge graph* that is then *queried* by the end user [3]. Each of these components making up a VKG system have been described in previous sections, except for the mapping.

2.6.1 Mapping

In the context of data integration a *mapping* is a set of expressions describing a relationship between a set of schemata. Typically one considers mapping expressions from the schema of a data source to a mediated schema which is used to present integrated data from different data sources to the end user, using a common vocabulary [19].

In the VKG approach the ontology can be said to play the role of mediated schema. It establishes the terminology of the final knowledge graph to which the end user sends their queries. This means the mapping that we are interested in establishes a relationship between the schemata of the data sources on the one hand and the ontology on the other hand [3]. It must therefore form a link between the relational and graph data models and be a way to construct graph data from relational data. As our focus in this work lies on the W3C standard graph data model RDF, we now define a RDF specific mapping concept.

Definition 2.16 Let Y be a set of columns that is the output of a SQL query, possibly empty. Consider three subsets, Y_1, Y_2, and Y_3, of Y. Then $\mathcal{T}(Y)$ is called a *triple template*, and is composed of three functions $\mathit{subject}(Y_1)$, $\mathit{predicate}(Y_2)$, $\mathit{object}(Y_3)$. The output of these functions must match the respective expected type of a RDF triple. That is, recalling the definitions of $\mathcal{I}$, $\mathcal{B}$, and $\mathcal{L}$ from Definition 2.1, it holds that

$$\mathit{subject}(Y_1) \in \mathcal{I} \cup \mathcal{B}, \quad \mathit{predicate}(Y_2) \in \mathcal{I}, \quad \mathit{object}(Y_3) \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}.$$

Defined in this way the triple template expresses how to use facts retrieved from a data source to construct RDF terms and triples. ♦

Again due to our focus on RDF graphs, we define mapping as it is used in our specific context.

Definition 2.17 A *mapping* in the VKG setting is a set of *mapping assertions*, each having the form $[\mathcal{T}(Y) \leftarrow (Y = \mathit{sql}(x))]$, where x is a data source, $\mathit{sql}(\cdot)$ is a SQL query over a data source schema, Y the output of that query, and $\mathcal{T}(Y)$ is a triple template. ♦

A VKG system uses the mapping to populate a virtual knowledge graph with facts sourced from the data sources. Merging the facts with the supplied ontology gives the complete knowledge graph that the end user queries. In order to describe mappings, Ontop supports both a native mapping language and the W3C recommendation R2RML. For the sake of readability we will use a simplified version of Ontop's native mapping language similar to that of Calvanese et al. in our examples [1]. At the same time we supply R2RML examples in the appendix.

Example 2.18 A mapping must relate one or more databases with the ontology, showing how to use the rows of a relational database to create RDF triples in the knowledge graph. Recall the IMDB database and the ontology of film described in Examples 2.12 and 2.14 respectively. Using a simplified mapping syntax, where `{-}` denotes answer values from the SQL query, the mapping

```
:films/{Name}{Year} rdf:type :Film.
  <- SELECT Name, Year FROM imdbTable1
```

applied to our example database gives rise to the triples

```
<http://f.org/films/The%20Shawshank%20Redemption1994> rdf:type :Film.
<http://f.org/films/The%20Godfather1972> rdf:type :Film.
<http://f.org/films/Pulp%20Fiction1994> rdf:type :Film.
```

And, as part of a VKG system, this combination of components, mapping, ontology, and data source, ensures that these three triples are generated and take place in the knowledge graph. ♢

R2RML

In order to express the mapping we need a mapping language, R2RML is the defacto standard mapping language and a W3C recommendation [9]. R2RML is itself expressed in the form of an RDF graph and written down using the concrete Turtle syntax. It uses a custom vocabulary to define mapping assertions and their target and source parts [9].

Example 2.19 The mapping of Example 2.18 is expressed in R2RML by the following RDF graph:

```
<#TriplesMap1>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
      rr:class :Film;
  ];
  rr:predicateObjectMap [
      rr:predicate ex:name;
      rr:objectMap [ rr:column "Name" ];
  ];
  rr:predicateObjectMap [
      rr:predicate :releasedIn;
      rr:objectMap [ rr:column "Year"];
  ].
```

Where the `rr:` prefix signifies the R2RML vocabulary. For detailed explanations of the vocabulary we refer the reader to the R2RML standard [9]. ♢

2.6.2 Formal description

Virtual Knowledge Graphs are formed by a combination of the components described so far. Formally VKGs have been specified well by Xiao et al. and we will use the same definitions for the purpose of this thesis.

Definition 2.20 A *VKG specification* is a tuple $P = (O, S, M)$, where O is an ontology, S is a data source schema, and M a mapping from S to O [2, 29]. ♦

Definition 2.21 A VKG specification $P = (S, P, O)$ together with a database D compliant with S forms a *VKG instance* (P, D). Applying the mapping M to the data sources produces the RDF graph $M(D)$, and further applying reasoning over the ontology O on $M(D)$ produces the virtual knowledge graph associated with the instance: $G_{P,D}$ [2, 29]. ♦

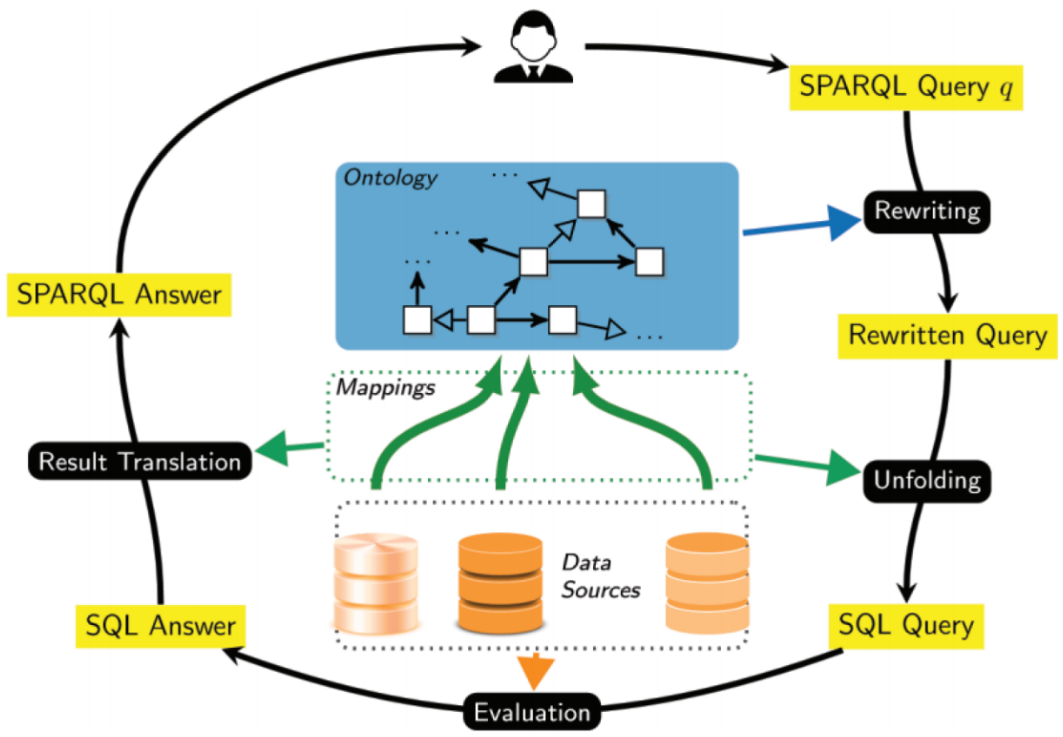


Figure 2.2: An overview of the VKG approach. Credit to Xiao et al. [2].

This graph $G_{P,D}$ is what is exposed to the end user who queries it using the standard language for querying RDF datasets, SPARQL. Taken together the two definitions above contain every necessary component and are sufficient for describing a VKG system.

2.6.3 VKG answering process

There are two large issues with how we have described the VKG approach until now. The first issue is that we have described individual components, but not much how they work together to integrate data. The second issue is that we have described a “bottom up” approach to data integration. We have described how mappings applied to data sources create RDF triples, which when combined with an ontology gives rise to a complete knowledge graph. This might be how the system appears to the end user, but the reality is that the VKG approach internally works completely different [1, 3]. This misunderstanding must be rectified and we take this section to reorient our understanding of the VKG approach.

One can divide the VKG approach to data integration into three parts, a manual configuration, an offline part, and an online part. The manual part is performed by a technical expert, it is the creation of the ontology and mappings. A very time consuming and difficult process requiring deep knowledge of every part of the system and domain, but a process that is out of the scope of this thesis and we will not discuss it much further. The offline part follows when the system loads the created ontology and starts up. The online part is what the system does when it is running [1]. What happens from the point of receiving a query from the end user, to the point of returning results? Figure 2.2 illustrates this process.

The system starts its online processing as the user poses a SPARQL query. As the first step the query is rewritten with respect to the specified ontology. For example, if the query asks to retrieve all objects of type `:Person` and our ontology contains a triple specifying that `:Actor` is a subclass of `:Person`, the system adds to the query asking also for all actors.

The query is then unfolded using the mappings in order to move from SPARQL to SQL. Any reference to RDF terms in the query is checked against the mapping expressions to find where those terms are mentioned in the target part. The RDF terms in the query are then replaced by the corresponding source SQL query, generating one or more SQL queries over the data sources. The generated SQL queries are executed over the relational data sources, and the result is translated back to a SPARQL answer using the mapping in the conventional way [2].

In the process of rewriting and unfolding the query the VKG system Ontop uses an intermediate data structure called the Intermediate Query (IQ) to bridge the gap between the SPARQL algebra and relational algebra [3]. Much of the work of this thesis focuses on this area of Ontop's internal workings, and we will describe it in a later chapter.

Example 2.22 Let us combine all the previous parts of this chapter in order to give an example of a working VKG system. Let the system be set up with the data sources, ontology, and mapping as described in the previous sections.

Let the end user pose a simple SPARQL query asking for all objects that are films. The first step in the process is the rewriting of the query. In our case with a very bare bones ontology nothing will happen. If one wants to imagine how this step looks, think of an ontology stating that all films are movies and vice-versa. Our query would then be rewritten to ask both for films and movies. We are now in the Intermediate Query (IQ) stage.

The next step is the unfolding with regards to the mapping. In our case, we will use Mapping 1 of Example 2.18 and see that in order to answer queries asking for films, we must turn our attention to the database described in Table 2.1. The IQ will add a node describing a SQL query over this database.

Finally, all the SQL queries (in our case one) part of the IQ will be executed by the database systems of the data sources. This happens completely outside the VKG system. Once the relational databases return their answers Ontop takes over again, using the mappings to transform the answers into RDF triples. The final answers are then returned to the user that posed the original query.

Assuming the query was a simple SPARQL SELECT query asking for all films, since we only had the one mapping, the answers would be:

- `:TheShawshankRedemption1994`

- `:TheGodfather1972`
- `:PulpFiction1994`. ♢

3 RDF-star

RDF-star is an extension of the RDF language proposed by a W3C community group[1]. Its main contribution is to improve support for statement level metadata in RDF graphs through the use of quoted triples, also known as embedded triples [27].

In this chapter a motivation is presented for the extension, followed by a description of the abstract and concrete syntaxes of RDF-star and SPARQL-star.

3.1 Motivation

Adding metadata to RDF triples is already possible through the process of *RDF reification*, and in order to understand why RDF-star is needed we must first investigate how RDF reification works and where it falls short.

RDF reification is done using the specific RDF vocabulary of `rdf:Statement`, `rdf:subject`, `rdf:predicate`, and `rdf:object`. In combination with a blank node the vocabulary gives rise to the semantic meaning of a blank node representing a given RDF triple. In this way the blank node can then be used as subject or object in any other RDF triple [4].

Example 3.1 Consider an RDF triple representing that a film has a certain score. We would like to annotate this triple stating that the source of the statement is the Internet Movie Database. A way to do that through RDF reification is shown below.

```
:/data/TheGodFather1972 :hasScore "9.2".
_:a a rdf:Statement;
  rdf:subject :/data/TheGodFather1972;
  rdf:predicate :hasScore;
  rdf:object "9,2";
  :source "IMDB".
```

And in order to retrieve such data, consider this SPARQL query fragment:

```
SELECT ?film ?score ?source
WHERE {
  ?film :hasScore ?score.
  ?b a rdf:Statement;
     rdf:subject ?film;
     rdf:predicate :hasScore;
     rdf:object ?score;
     :source ?source.}
```

[1]The latest draft can be seen at `https://w3c.github.io/rdf-star/cg-spec`

In addition to being very complex, the user must have knowledge specifically of RDF reification in order to formulate such a query correctly. $\diamondsuit$

In short, for every triple one wishes to add metadata to through RDF reification, a blank node and three additional triples need to be introduced. As a result, both the RDF data and SPARQL query in the example above are written in a very convoluted way. If one needs to form more complex expressions, such as nesting meta comments, it further complicates things.

Other ways of adding statement level metadata to RDF graphs have been proposed in order to improve this situation. Named graphs and RDF singleton properties are two of these approaches, but they both suffer from their own flaws making them unsuitable for their purpose [27]. RDF-star aims to improve upon the state of the art by filling the function of adding metadata, also known as contextual data, to RDF triples in a concise manner [27].

3.2 Extending RDF's Abstract Syntax

The basis of RDF-star's contribution lies in the following definition.

Definition 3.2 Let $\mathcal{I}$ be the set of IRIs, $\mathcal{B}$ the set of blank nodes, and $\mathcal{L}$ the set of literals, and let $s \in \mathcal{I} \cup \mathcal{B}$ be a subject, $p \in \mathcal{I}$ a predicate, and $o \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$ an object. Recall the RDF triple of Definition 2.1.

We define that any RDF triple also is an *RDF-star triple*. Further, if t and t' are RDF-star triples then (t, p, o), (s, p, t), and (t, p, t'), are RDF-star triples and t, t' are called *quoted triples* [15]. $\blacklozenge$

This definition is then naturally used to define several other terms. RDF terms, see Section 2.1, together with RDF-star triples form the *RDF-star terms*, and a set of RDF-star terms is called an *RDF-star graph* [15].

In this way the RDF triple that one desires to annotate simply takes the place of subject or object of an RDF-star triple, and the problems of other reification techniques are avoided. For example to add meta-data to a triple, only one additional triple needs to be introduced, in contrast with RDF reification where four additional triples must be added.

An RDF-star triple is uniquely identified by its three components of subject, predicate, object. It makes no difference where it is placed, as root triple or embedded. For example, assume $t = (s, p, o)$ is a triple and i an IRI; the triple (t, i, t) contains only one RDF-star triple but in two places [15].

A difference is made between quoted triples on the one hand and triples taking a place in an RDF-star graph. RDF-star triples that are part of an RDF-star graph are *asserted triples*, intuitively such a triple is a “root” triple. As an RDF-star triple is uniquely identified by its components, the same triple may be both asserted and quoted.

RDF-star is also backwards compatible with RDF. Through a *star-mapping*, unrelated to the mappings discussed previously in this thesis, RDF-star data can be converted to RDF data using the RDF reification vocabulary, and vice-versa [27].

Example 3.3 We wish to state the following fact expressed in English: The Internet Movie Database claims that the film The Godfather has a score of 9.2. Using the abstract RDF-star syntax, if we do not need to be very exact, this can be encapsulated by the triple: (t, p_1, `"IMDB"`), where

$$\begin{aligned} t &= (s\,, p_2, \texttt{"9.2"}),\\ p_1 &= \texttt{http://lukas.thesis.org/films\#source},\\ s &= \texttt{http://lukas.thesis.org/films\#data/TheGodfather1972},\\ p_2 &= \texttt{http://lukas.thesis.org/films\#hasScore}. \end{aligned}$$

In this example the root RDF-star triple contains another quoted RDF-star triple in the subject position. ♢

Following Hartig [27] we further define some useful functions that, unlike what we have previously discussed, have no equivalent in the RDF standard.

Definition 3.4 The *elements* of an RDF-star triple t, denoted Elmts(t), are all RDF-star terms appearing in the triple, or any quoted triples [27]. Mathematically speaking, if $t = (s, p, o)$ and $\mathcal{T}$ is the set of all RDF-star triples,

$$\mathrm{Elmts}(t) = \{s, p, o\} \cup \{x \in \mathrm{Elmts}(t') \,|\, t' \in \{s, o\} \cap \mathcal{T}\}.$$

The function is extended to an RDF-star graph G to be the union of Elmts(t) [27], for all triples t in G:

$$\mathrm{Elmts}(G) = \bigcup_{t \in G} \mathrm{Elmts}(t).$$

Further we define the *triples of* an RDF-star graph G to be any RDF-star triples, embedded or not, contained in G [27]:

$$T(G) = G \cup (\mathrm{Elmts}(G) \cap \mathcal{T}).$$

Note that following from the basic RDF-star definitions the sets defined here are all finite. ♦

Example 3.5 The elements of the triple t from Example 3.3 are

$$\begin{aligned}\mathrm{Elmts}(t) &= \{p_1, p_2, s, \texttt{"IMDB"}, \texttt{"9.2"}\}, \text{ where}\\ p_1 &= \texttt{http://lukas.thesis.org/films\#source},\\ p_2 &= \texttt{http://lukas.thesis.org/films\#hasScore},\\ s &= \texttt{http://lukas.thesis.org/films\#data/TheGodfather1972}.\end{aligned}$$

Note that t itself is not part of Elmts(t). ◇

3.3 Concrete RDF-star Syntaxes

The RDF-star community group has also described extensions to several of the concrete syntaxes used in RDF documents.

3.3.1 Turtle-star

Turtle has been extended to allow quoted triples as subjects or objects forming the syntax of Turtle-star. The main way to represent a quoted triple is through the use of double less/greater than signs (`<< s p o >>`), we refer to this as the *less/greater than syntax*, or the *main syntax*.

Alternatively, an *annotation syntax* is supported to add metadata to triples [15]. The annotation syntax is enclosed in curly braces and vertical bars (`{| |}`) and contains what is called a *PredicateObjectList* in the Turtle standard [14]. The PredicateObjectList is a sequence of (predicate, object) pairs that follows the annotated triple, using it as subject to create RDF-star triples. We illustrate with an example.

Example 3.6 The RDF document below, written in the Turtle-star syntax, describes the score of a movie and its source.

```
:data/TheGodfather1972 :hasScore "9,2".
<< :data/TheGodfather1972 :hasScore "9,2" >> :source "IMDB".
```

The same graph can be expressed using the annotation syntax:

```
:data/TheGodfather1972 :hasScore "9,2" {| :source "IMDB" |}.
```

In this case the quoted triple plays the role of subject in the `:source` relationship. As we want the triple to also be asserted we need to type out two lines using the main syntax, but only one line using the annotation syntax. In this example we have written in Turtle-star the RDF-star triple described in Example 3.3. ◇

Note that the annotation syntax can easily be converted to the main syntax, but as the triple being annotated always is asserted it is not possible to convert every graph written in the main syntax to the annotation syntax. Therefore we will focus our discussion and examples on the main syntax. Another way to express this is to say that the less/greater than syntax is more expressive than the annotation syntax as it allows the use of a triple in the subject position without asserting it in the RDF-star graph.

Throughout this thesis, just as we've used Turtle before to express RDF-graphs, we will be using Turtle-star for RDF-star graphs, unless otherwise mentioned.

3.3.2 TriG-star, N-Triples-star, N-Quads-star

Additionally the community draft describes extensions of other popular concrete RDF syntaxes. As they are not relevant for this thesis we will not describe them further but refer the reader to the RDF-star draft report [15].

3.4 SPARQL-star

In addition to their extension of the RDF language, Hartig et al. have developed an extension to the SPARQL query language named SPARQL-star [15, 27]. Just like RDF-star builds on the concept of a triple, the SPARQL-star extension builds on the SPARQL concepts of triple pattern and basic graph pattern, see Definitions 2.5 and 2.6. The definitions below are sourced from Hartig [27] and the W3C community group working on RDF-star [15].

Definition 3.7 *SPARQL-star triple patterns* are defined recursively. First, every SPARQL triple pattern is also a SPARQL-star triple pattern. Second, let tp and tp' be SPARQL triple patterns, $\mathcal{I}$ the set of IRIs, $\mathcal{V}$ the set of SPARQL query variables, and $\mathcal{L}$ the set of RDF literals. Then (tp, p, o), (s, p, tp'), and (tp, p, tp') are all *SPARQL-star triple patterns* [5, 27], under the assumption that

$$s \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}, \quad p \in \mathcal{V} \cup \mathcal{I}, \quad o \in \mathcal{V} \cup \mathcal{I} \cup \mathcal{L}.$$

As you can see this definition closely follows the form of the definition of RDF-star triple. ⧫

Note *Just like for standard RDF and SPARQL there is a disconnect here. SPARQL-star triple patterns admit literals in the subject position, unlike RDF-star triples where the subject is an IRI, blank node, or another RDF-star triple.*

Definition 3.8 A finite set of SPARQL-star triple patterns is a *SPARQL-star basic graph pattern* [15, 27]. ⧫

The concrete syntax of the query language is extended in a similar way to how Turtle-star extends turtle. Either through the use of the double less/greater than signs, or through an annotation syntax. SPARQL-star is used to query an *RDF-star dataset*, which is one or more RDF-star graphs just like an RDF dataset is one or more RDF graphs.

Example 3.9 Recall the reified RDF data of Example 3.1, and the query retrieving films, their scores, and the source of that score. Assuming an equivalent RDF-star dataset, a SPARQL-star query to retrieve the same answers would look like this.

```
SELECT ?film ?score ?source
WHERE {
  ?film :hasScore ?score {| :source ?source |}.
  }
```

Alternatively using the main syntax, the WHERE block is equivalently expressed as

```
WHERE {
  ?film :hasScore ?score.
  << ?film :hasScore ?score >> :source ?source.
  }
```

Same as in Turtle-star, the annotation syntax is more concise. $\diamondsuit$

In the example query both (`?film :hasScore ?score.`) and (`<< ?film :hasScore ?score >> :source, ?source.`) are SPARQL-star triple patterns. Taken together they form a SPARQL-star basic graph pattern.

Similarly to how they are defined for RDF-star triples, we define the *elements* of a triple pattern $tp = (s, p, o)$ as: $\text{Elmts}(tp) = \{s, p, o\} \cup \{\text{Elmts}(tp') \mid tp' \in \{s, o\} \cap \mathcal{TP}\}$, where $\mathcal{TP}$ is the set of all RDF-star triple patterns. For every RDF-star basic graph pattern the elements of that pattern is the union of the elements of each triple pattern tp in the graph pattern [27].

3.4.1 Algebra

Recall the definition of a SPARQL solution mapping from Definition 2.9. It has been extended by Hartig to allow for a formal semantics of SPARQL-star [27].

Definition 3.10 A *SPARQL-star solution mapping* is a partial function η from the set of query variables V to the set of RDF-star terms $\mathcal{T} \cup \mathcal{I} \cup \mathcal{B} \cup \mathcal{L}$. Given a SPARQL-star basic graph pattern BGP, $\eta(BGP)$ is the basic graph pattern obtained by replacing the variables of BGP according to the mapping η. Call all variables for which η is defined the *domain* of η, denoted $\text{dom}(\eta)$ [27]. $\blacklozenge$

Definition 3.11 Let D be an RDF-star dataset, tp a triple pattern, and BGP an RDF-star basic graph pattern. The *evaluation* of a graph pattern over D, call it $[\![\cdot]\!]_D$, is a function from the set of RDF-star basic graph patterns that returns a set of SPARQL-star solution mappings. It is defined as [27]

$$[\![BGP]\!]_D = \{\eta \mid \text{dom}(\eta) = \text{Elmts}(BGP) \cup \mathcal{V} \text{ and } \eta(BGP) \subset T(G)\}.$$

These definitions allow for formal evaluation of SPARQL-star. $\blacklozenge$

3.4.2 Functions and operators

The SPARQL standard defines a number of functions and operators that apply to standard RDF terms and query variables allowing for example filtering of query results. SPARQL-star extends these to apply to RDF-star terms instead of only standard RDF terms. In addition, five new functions are introduced, three of which are modified functions from the SPARQL standard. For further information we refer the reader to the RDF-star community draft [15].

3.4.3 Answer formats

Depending on the dataset being queried, the SPARQL answer format might not be sufficient to capture variable bindings produced when querying RDF-star data.

Example 3.12 Consider the SPARQL-star query:

```
SELECT ?s
WHERE {<< ?s ?p1 ?o1 >> ?p2 ?o2}.
```

If it is evaluated over the RDF-star dataset consisting of the single triple: `<< :s :p1 :o1 >> :p2 :o2`, there is no need for an extension of the SPARQL answer format. This is because there is only one answer, namely the solution mapping binding `?s` to `:s`, and `:s` is an IRI, a RDF-term. If we instead execute the SPARQL query:

```
SELECT ?s
WHERE {?s ?p ?o}
```

over the same dataset, the query variable `?s` is bound to the quoted triple `<< :s :p1 :01 >>`. This triple is not an RDF term and is not supported as answer in the standard SPARQL answer formats. ♢

As the example above illustrates, there is a need to extend the SPARQL answer formats. This has been done by the RDF-star community group and the details can be read in the RDF-star draft [15]. The SPARQL JSON and XML answer formats have been extended to allow for quoted triples being bound to variables. This is a necessary addition to allow correctly querying RDF-star datasets [15].

4 Investigating RDF-star Support in VKG Systems

In this chapter we investigate which parts of a VKG system could be extended with RDF-star support, what the purpose of it is, and how it is done. We deduce that the mapping is the component most suitably extended and propose an extension to R2RML that we call R2RML-star. We detail how to extend VKG systems to support the relevant query language, SPARQL-star. In addition we discuss how the stages of query processing, meaning rewriting and unfolding, are affected by these changes, and finally there is a large example to illustrate the lessons of the chapter.

4.1 RDF-star Support in the Mapping

Recall that a VKG mapping, as defined in Definition 2.17, is a set of mapping assertions, each consisting of an RDF triple template and a SQL query. The triple template and SQL query are also known as the target and source part, respectively, of the assertion.

The mapping is a key component of a VKG system, it is used to populate the virtual knowledge graph with facts mapped from the data sources. For this reason we wish to extend the target part of the mapping assertion to support RDF-star triples.

Definition 4.1 We define a *triple-star template* $T_*(Y)$ as follows. First, every triple template, see Definition 2.16, is also a triple-star template. Second, assume that

$$\begin{aligned} T_*(Y) &= (\mathit{subject}(Y_1), \mathit{predicate}(Y_2), \mathit{object}(Y_3), \\ T'_*(Y') &\text{ and } T''_*(Y'') \end{aligned}$$

are all triple-star templates, then

$$\begin{aligned} &(T'_*(Y'), \mathit{predicate}(Y_2), \mathit{object}(Y_3)) \\ &(\mathit{subject}(Y_1), \mathit{predicate}(Y_2), T''_*(Y'')) \\ &(T'_*(Y'), \mathit{predicate}(Y_2), T''_*(Y''). \end{aligned}$$

are also triple-star templates. ♦

Definition 4.2 An *RDF-star mapping* is a set of *mapping assertions*, each having the form: $[\mathcal{T}_*(Y) \leftarrow (Y = \mathit{sql}(\cdot))]$, where $\mathit{sql}(\cdot)$ is a SQL query over a data source schema S, Y is the set of output variables of the SQL query, and $\mathcal{T}_*(Y)$ is a triple-star template. ♦

4.1.1 R2RML extension

In order to extend the concept of a mapping to that of an RDF-star mapping we propose a standard-breaking extension to the mapping language R2RML, for the purpose of this thesis we call it R2RML-star. This is done through the use of the prefix `star:`[1], a new R2RML term type representing RDF-star triples (thus breaking the standard), and a set of predicates specifying the subject, predicate, and object of the triple.

Some work has already been done to create a language mapping relational data to RDF-star by Delva et al. in the form of RML-star [28]. The reason for us choosing a different approach is that the RML mapping language was designed for Extract-Transform-Load (ETL) use and is not suitable for creating virtual knowledge graphs. RML is an extension of R2RML supporting unstructured forms of data such as XML and CSV files [16]. These files are often slow to access and parse as they don't support for example indexing, making accessing them through a VKG approach unsuitable.

The R2RML language specifies a language to describe mappings between relational databases and RDF graphs, it is described in Section 2.6.1. Part of this mechanism lies in the term maps of R2RML, which specify how to generate RDF terms [9]. We extend this mechanism to have the ability to generate RDF-star terms.

Every column- and template-valued term map has an inferred or specified term type which specifies which type of RDF term is generated. In standard R2RML these are: IRI, blank node, and literal. To add support for the additional RDF-star term of quoted triple, we define a new term type: `star:RDFStarTermType`, which in R2RML-star can be the specified term type in subject or object maps. This breaks the standard and leads to any software supporting standard R2RML and not our extension to produce an error, which however is a desirable property.

In order to specify the subject, predicate, and object of the quoted triple to be generated, three new properties are defined: `star:subject`, `star:predicate`, and `star:object`. The object of the `star:subject` and `star:object` properties must be an object map, which in turn can have any term type, including the new RDF-star term type representing a deeper nested triple. The `star:predicate` property takes a predicate map as object and it can have only one term type: IRI.

Example 4.3 To see the full mapping featured in this example see Mappings 2 and 3 in the appendix. The triple-star template [(`:/{Name}{Year}`, `rdf:type`, `:Film`), `:source`, "IMDB database"] would be expressed as follows in R2RML-star.

```
rr:subjectMap [ a rr:SubjectMap;
    rr:termType star:RDFStarTermType;
```

[1]`star` being the IRI: `https://w3id.org/obda/r2rmlstar#`

```
      star:subject [a rr:ObjectMap;
        rr:template "http://lukas.thesis.org/films/{Name}{Year}";
        ];
      star:predicate [a rr:PredicateMap;
        rr:constant rdf:type
        ];
      star:object [a rr:ObjectMap;
        rr:constant :Film
        ];
    ];
  rr:predicateObjectMap [
      rr:predicate :source;
      rr:object "IMDB database";
    ].
```

The triple-star template [`:/{Name}{Year}`, `:isMentionedBy`, (`:/{Name}{Year}`, `:releasedIn`, "{year}")] would be expressed as follows in R2RML-star.

```
  rr:subjectMap [ a rr:SubjectMap;
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
    ];
  rr:predicateObjectMap [
      rr:predicate :isMentionedBy;
      rr:objectMap [  a rr:ObjectMap;
        rr:termType star:RDFStarTermType;
        star:subject [a rr:ObjectMap;
            rr:template "http://lukas.thesis.org/films/{Name}{Year}";
            rr:termType rr:IRI;
          ];
        star:predicate [a rr:PredicateMap;
            rr:constant :releasedIn;
          ];
        star:object [a rr:ObjectMap;
            rr:column "year";
            rr:termType rr:Literal;
          ];
        ];
    ].
```

These are simple examples to illustrate the basics of R2RML-star but more complex templates are of course possible. As follows from the definition of RDFStarTermType, nesting is also allowed, just as in RDF-star. ♢

4.2 SPARQL-star Support

A part of implementing RDF-star support is to support the extended query language SPARQL-star. In order to do this the VKG system's query processing must be extended from supporting the normal SPARQL algebra described in Section 2.3.3, to supporting the extended SPARQL-star algebra of Section 3.4.1

Recall that a VKG system goes through several stages of query processing. After parsing the query, it is manipulated through the stages of rewriting and unfolding with regards to the ontology and mapping respectively. Both of these processes must be extended to take into account the new grammar of the SPARQL-star algebra.

4.2.1 Parsing SPARQL-star queries

The first step to SPARQL-star support is the correct parsing of SPARQL-star queries. This means that the VKG system must correctly read a SPARQL-star query; the system must be able to recognize SPARQL-star triple patterns, SPARQL-star functions and operators, and other aspects of the SPARQL-star standard as described in Section 3.4.

4.2.2 Extending query rewriting

VKG systems rewrite posed queries with the help of the supplied ontology in order to enhance the answers to the query, we call this *query rewriting.* This is done through the process of SPARQL entailment, see Section 2.5.3. In order to support entailment under SPARQL-star, or standard SPARQL queries combined with an RDF-star mapping, the VKG system needs to be extended to apply entailment rules to RDF-star triples.

Example 4.4 Consider the RDF-star dataset consisting of the single triple `<< :John a :Actor >> :source "IMDB".` and an ontology with the single statement `:source rdfs:range :Database.` In this case the query

```
SELECT ?person
WHERE {
  << ?person a :Actor >> :source ?source.
  ?source a :Database.
}
```

should return `:John` as only answer, if the system runs under an RDFS entailment regime. ♢

We call this form of rewriting simple rewriting. That is, when the ontological rule applies to the root RDF-star triple and not to a quoted triple.

Definition 4.5 *Simple rewriting* is the process of query rewriting when SPARQL entailment rules are applied to root RDF-star triples. That is, for each asserted ontological rule ϕ and RDF-star triple t_1 such that $\phi(t_1) \implies t_2$, the VKG system handles simple rewriting if a query that returns a variable binding to t_1, also returns a binding to the same variable, t_2. ⧫

Example 4.6 Consider the RDF-star dataset consisting of the single triple `<< :John a :Actor >> :source "IMDB".` and an ontology with the single statement `:Actor rdfs:subclassOf :Person.` In this case the query

```
SELECT ?person
WHERE {<< ?person a :Person >> :source "IMDB".}
```

should return `:John` as only answer, if the system runs under an RDFS entailment regime. ♢

Here the entailment has to be applied to a quoted triple which makes the rewriting more complex and we thus refer to this case as complex rewriting.

Definition 4.7 *Complex rewriting* is the process of query rewriting when SPARQL entailment rules are applied to quoted triples, and possibly root RDF-star triples. For each asserted ontological rule ϕ and RDF-star triple t_1 such that $\phi(t_1) \implies t_2$, the VKG system handles complex rewriting if a query that returns a variable binding to (t_1, p, o), also returns a binding to the same variable, (t_2, p, o). Similarly the rewriting should apply also if the triple takes the place of object in an RDF-star triple. ⧫

In order for a VKG system to have full RDF-star support at least simple rewriting should be supported. As for complex rewriting, the RDF-star community is still undecided on how it should be implemented. Rewriting quoted triples can lead to undesirable results, as the following example illustrates.

Example 4.8 Consider the ontology consisting of the single RDFS statement (`:Actor`, `rdfs:subClassOf`, `:Person.`). This is combined with the RDF-star graph consisting of a single RDF-star triple:
`<< :John`, `rdf:type`, `:Actor >>`, `:dateAdded`, `"2022-03-03".`
The quoted triple denotes John being an actor, and the annotation specifies when the triple was added. Now when a VKG system is running and rewriting this triple with regard to the ontology a new triple will be implicitly generated:
`<< :John`, `rdf:type`, `:Person >>`, `:dateAdded`, `"2022-03-03".`
Unless the system is running on the third of March 2022, this triple is factually incorrect. Similar issues arise in many use cases where the annotation of a triple does not necessarily apply to the output of its rewriting. ♢

4.2.3 Extending query unfolding

Once a query has been parsed and rewritten, it is necessary to unfold it with regards to the mapping. The VKG system must be extended to handle the new triple patterns, functions, and operators as described in Section 3.4.

Even if we don't include these new parts of the grammar from the query, so that it looks like a standard SPARQL query, the unfolding must be extended. This is because querying RDF-star data might return answer variables bound to quoted triples, regardless of the fact that a standard SPARQL query does not contain SPARQL-star triple patterns.

Example 4.9 Consider the RDF-star dataset consisting of the single triple from Example 4.4 (`<< :John a :Actor >> :source "IMDB".`), the query

```
SELECT ?s
WHERE {?s ?p ?o.}
```

returns a single value bound to `?s`, the quoted triple: `:John a :Actor`. ♢

R2RML-star unfolding

As we know, VKG systems do not query RDF datasets directly. In Example 4.9 above, the triple retrieved must have been generated from a mapping. To our knowledge the only mapping that generates RDF-star data is our proposed R2RML-star extension. In this section we describe how a VKG system handles R2RML-star.

The query unfolding in a VKG system fundamentally works through the process of graph matching. The SPARQL query's WHERE clause is matched with the target part of the mapping. Already here R2RML-star poses problems as each quoted RDF-star triple can match with either one or three query variables.

Example 4.10 A mapping with the target part ([`:`{Person}, `rdf:type`, `:Actor`], `:source`, `"IMDB"`), where {Person} refers to a database column, should match to both of the queries below.

```
SELECT ?s
WHERE {?s ?p ?o.}

SELECT ?s
WHERE {<<?s ?p1 ?o1>> ?p2 ?o2.}
```

Assuming that this database column contains `"John"` as its only value, each query should return one answer each, `<< :John a :Actor >>` and `:John`, respectively. ♢

The VKG system must be extended to correctly handle these two scenarios, which is non trivial. The system must also be extended to correctly handle the accompanying source part of the mapping, the SQL queries, and execute them over the data sources.

4.2.4 Extending query answers

As shown by Example 4.9 there is a need to extend the answers returned by the VKG system. Quoted triples can be bound to variables and the system must be changed to take this into account and produce as output the SPARQL-star standard's new answer formats as described in Section 2.3.2.

4.2.5 Adding support for new functions and operators

In Section 3.4.2 we saw that SPARQL-star redefines the standard SPARQL functions and operators, and introduces new ones. This must be taken into account when extending a VKG system with SPARQL-star support and the internal query processing must be extended to correctly apply the updated and new functions and operators.

4.3 End to End Example

We showcase here a complete end to end example demonstrating the power of a VKG system extended with RDF-star support. As a first step we introduce a data integration scenario. The next subsection focuses on how it would be solved with a VKG system supporting only standard RDF. Finally we show an example of how a RDF-star system as described in this thesis could operate.

4.3.1 The data integration scenario

Recall the two databases of Example 2.12, visually described in Tables 2.1 and 2.2. The data stored is sourced from two different popular websites, the Internet Movie Database (IMDB) and Rotten Tomatoes. For each database, the data contains three pieces of information, the film's name, release year, and score as given on the source website. An example entry of the IMDB database looks as such:

`Name: Pulp Fiction; Year: 1972; Score: 9.2.`

Now, our hypothetical end user is interested in integrating these two data sources. That is, they want to combine them into one for the purpose of querying them simultaneously. There are many ways to this but let us assume that he or she wants to integrate them using the VKG approach. A particular problem arises due to the heterogeneity of the data sources. The scores of the movies differ between the two sources. Not only the scores themselves but also the format, how should our user solve this conundrum?

We assume that the user wants to preserve the scores as they are, without converting them to the same format for example. But some information should be added, such as meta-data specifying from which data source the score comes.

4.3.2 A standard RDF solution

We will focus on the mapping of the VKG solution as that is the crucial part.

Example 4.11 To map the score in R2RML we use the following two mappings, written in a simplified syntax:

```
:films/{Name}{Year} :score {Score}.
  <- SELECT Name, Year, Score FROM imdbTable1
:films/{MovieName}{ReleaseYear} :score {Rating}.
  <- SELECT MovieName, ReleaseYear, Rating FROM RtTable1
```

The first mapping expressed in R2RML is:

```
<#Mapping-film-score>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:template "http://lukas.thesis.org/films/{Name}{Year}";
  ];
  rr:predicateObjectMap [
    rr:predicate :score;
    rr:objectMap [ rr:column "Score" ];
  ].
```

In order to include the desired metadata, we make use of RDF reification. This leads us to this far more complex mapping:

```
<#Mapping-reify-annotate-score>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:template "Statement{Name}{Year}{Score}";
    rr:termType rr:BlankNode;
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:subject;
    rr:objectMap [
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
      rr:termType rr:IRI; ];
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:predicate;
    rr:objectMap [
      rr:constant :score ; ];
  ];
  rr:predicateObjectMap [
    rr:predicate rdf:object;
    rr:objectMap [
      rr:column {Score}; ];
  ];
  rr:predicateObjectMap [
    rr:predicate :source;
    rr:objectMap [
      rr:constant "IMDB" ];
  ].
```

The usage of RDF reification in this way is necessary in order to annotate triples, however it adds some difficulty for the end user of the system. For example, this intuitively correct SPARQL query asking for all film scores will not return any answers using the reification based mappings.

```
SELECT ?film ?score
WHERE { ?film :score ?score }
```

Instead the more complex query below will give the expected results.

```
SELECT ?film ?score
WHERE {
  ?statement rdf:subject ?film .
  ?statement rdf:predicate :score .
  ?statement rdf:object ?score . }
```

This complicates the use of the system for the end user, not to mention the performance impact on the system when it has to handle more complicated queries. ♢

4.3.3 A R2RML-star mapping

Next, we wish to simplify by using our proposed R2RML-star extension.

Example 4.12 The metadata infused mapping of the previous section can be rewritten in R2RML-star as shown below.

```
<#Mapping-film-score>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:template "http://lukas.thesis.org/films/{Name}{Year}";
    ];
  rr:predicateObjectMap [
    rr:predicate :score;
    rr:objectMap [ rr:column "Score" ];
    ].

<#Mapping-star-annotate-score>
  rr:logicalTable [ rr:tableName "imdbTable1" ];
  rr:subjectMap [
    rr:termType star:RDFStarTermType;
    star:subject [a rr:ObjectMap;
      rr:template "http://lukas.thesis.org/films/{Name}{Year}";
      ];
    star:predicate [a rr:PredicateMap;
      rr:constant :score
      ];
    star:object [a rr:ObjectMap;
      rr:column {Score};
      ];
    ];
  rr:predicateObjectMap [
    rr:predicate :source ;
    rr:objectMap [ rr:constant "IMDB" ];
    ].
```

The first mapping here, `<#Mapping-film-score>` was used also in the previous section. It uses standard RDF and serves to assert the triple in the graph. The second mapping has the same triple pattern but generates quoted triples, along with the annotated `:source`.

Using this R2RML-star mapping, the system remains intuitive for the end-user and the following SPARQL query will return the expected results.

```
SELECT ?film ?score
WHERE { ?film :score ?score }
```

In addition one can easily retrieve the associated metadata with this SPARQL-star query:

```
SELECT ?film ?score ?source
WHERE { << ?film :score ?score >> :source ?source .}
```

This simplicity stands in stark contrast to the complexity of RDF reification. ◊

5 Implementing RDF-star Support for Ontop

In this chapter we apply the studies done in the previous chapter to extend the open source system Ontop with RDF-star support. We describe what needs to be implemented, how it is done, and also the parts of Ontop's internal functioning that are necessary to understand the other descriptions. The functions that are extended include query parsing, rewriting, and unfolding, and the parsing and handling of R2RML-star mappings. We also describe work that is in progress, including the support of SPARQL-star answer formats and SPARQL-star functions and operators. We explain internal Ontop structures such as the Intermediate Query and query nodes.

5.1 Parsing the Query

As described in Section 4.2.1 a VKG system must be able to parse SPARQL-star queries in order to support them, and we show now how this support has been added to Ontop. Ontop uses the RDF4J library to parse SPARQL queries input by the end user [22]. RDF4J is a popular Java library used to interact with RDF data, it includes support for reading and parsing not only the standard concrete RDF syntaxes, but also RDF-star data, SPARQL queries, and SPARQL-star queries [25]. Hence, by virtue of RDF4J's SPARQL-star support, Ontop supports the parsing of SPARQL-star queries.

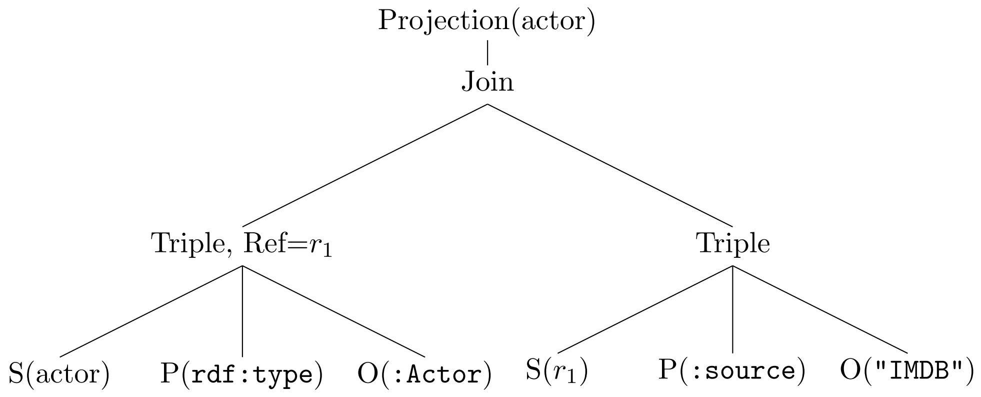
5.1.1 RDF4J's parsing of SPARQL-star queries

SPARQL and SPARQL-star queries are parsed by RDF4J into a tree structure, call it an *RDF4J Tree*, representing the SPARQL algebra. The trees contain nodes representing operations of the SPARQL algebra. For SPARQL-star queries a design decision has been taken wherein an RDF-star triple containing quoted triples is not represented by a single node, but rather by a sub-tree of joins, triples, and quoted triples. Each quoted triple is represented as a triple with an added fourth element used as reference. A pointer to this reference then takes the place of subject or object in the encompassing RDF-star triple.

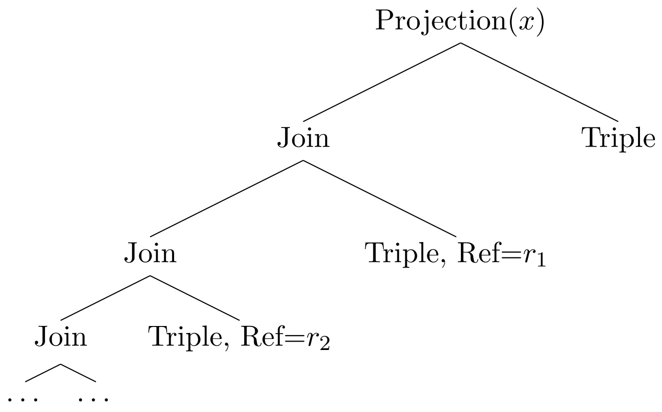
Example 5.1 The SPARQL query

```
SELECT ?actor
WHERE { <<?actor rdf:type :Actor>> :source "IMDB". }
```

is parsed into the following RDF4J tree



Where S, P, O signify the subject, predicate, and object respectively of the RDF-star triple. A deeply nested triple generates multiple joins as RDF4J's join node is a binary operator. For example the generated RDF4J tree of a deeply nested triple might look like this:



This limitation on RDF4J's join node leads to convoluted trees coming from SPARQL-star queries. We will later show how this issue is dealt with when it comes to Ontop's internal data structures. ♢

5.2 Intermediate Query

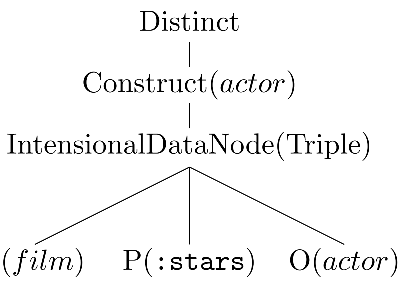
In order to bridge the gap between SPARQL queries on the one hand and SQL queries on the other hand, the VKG system Ontop uses a data structure called the *Intermediate Query*, or *IQ* for short. This is the format that is used to store a query throughout all the stages of query manipulation that Ontop performs. This includes rewriting the query in regard to the ontology, unfolding the query in regards to the mapping, and optimizing the query. The SPARQL query posed by the system's end user is first turned into an RDF4J tree independent of Ontop, then directly transformed to an IQ before these stages. At the end of the manipulation most of the IQ is converted to SQL queries to be evaluated over the data sources by the external Database Management Systems [3].

Through the work of this thesis we have extended many parts of Ontop that interact with this data model and to understand these changes it is necessary to understand the IQ itself. The IQ is at its core a tree structure built up of different *Query Nodes* that we describe further below.

Example 5.2 The SPARQL query

```
SELECT DISTINCT ?actor
WHERE {
 ?film :stars ?actor .
}
```

Is parsed by RDF4J to construct an RDF4J tree. That tree is then converted by Ontop into the IQ:



The distinct node is self explanatory, the construct node signifies projection, which variables are projected upwards through the tree, and the data node will be explained below. ♢

5.2.1 Query nodes

The Query nodes are an abstraction representing query operations common to both the SPARQL and SQL query languages, for example the distinct node which eliminates duplicate answers. The design of these nodes is complex as the two query languages have their differences and the IQ must be sufficiently abstract to represent them both. We will present those parts of the IQ that are relevant for our work. For a full characterization we refer the reader to Ontop's documentation for developers [23, 24].

Data nodes and predicates

Data nodes are the typical leaf nodes of the IQ. They represent the data that will ultimately be returned to the system's end user. They come in two kinds, *intensional* and *extensional* data nodes. Intensional data nodes represent data in the RDF format such as triples and quads. During query unfolding, these nodes are transformed by Ontop into extensional data nodes containing the information necessary to execute SQL queries over the source databases. Our work does not touch on extensional data nodes and we will focus our efforts on describing intensional nodes.

Each intensional data node contains a *predicate* acting as a marker for its type. The version of Ontop supporting only standard RDF contains two such predicates that could be contained in an intensional data node, triple and quad. We have extended this to a hierarchy of predicates capable of signaling three properties necessary to represent RDF-star triples: does the triple have a quoted subject, does it have a quoted object, and is it itself a quoted triple? An example of a predicate combining answers to these questions is the TripleRefNestedSubject predicate. It has a quoted subject, but no quoted object, and the Ref denotes that it is itself a quoted triple. The full list of such predicates is eight long and of course includes also the standard RDF triple, which is not itself quoted and contains no quoted triples. The full list:

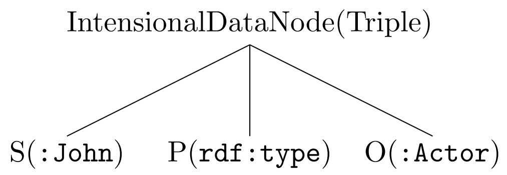
- Triple
- TripleNestedSubject
- TripleNestedObject
- TripleNestedSO (Nested Subject & Object)
- TripleRef
- TripleRefNestedSubject
- TripleRefNestedObject
- TripleRefNestedSO

These nodes and their predicates first appear during runtime as the system translates the end user's SPARQL-star query into an IQ. The triple patterns of the SPARQL query are internally represented by intensional data nodes. For a SPARQL-star query, any triple-star patterns are represented by data nodes containing the newly created predicates. The next time they appear is when an R2RML-star mapping is read by Ontop. The target part of the mapping is translated into its own IQ tree, to be matched with a part of or the whole IQ tree coming from the SPARQL query.

This typing system identifying RDF-star triples as separate from those of standard RDF, along with their properties, gives the system a tool by which to avoid incorrectly matching mappings with the wrong level of nesting with a SPARQL-star query. It could also be used to implement many of the new SPARQL-star operators as described in Section 3.4.2.

In addition to the predicate, each data node contains data. In the case of intensional data nodes, regardless of predicate, they represent an RDF(-star) triple, hence they contain three data points: subject, predicate, and object.

Example 5.3 A simple RDF triple such as (`:John`, `rdf:type`, `:Actor`) is represented by the intensional data node:

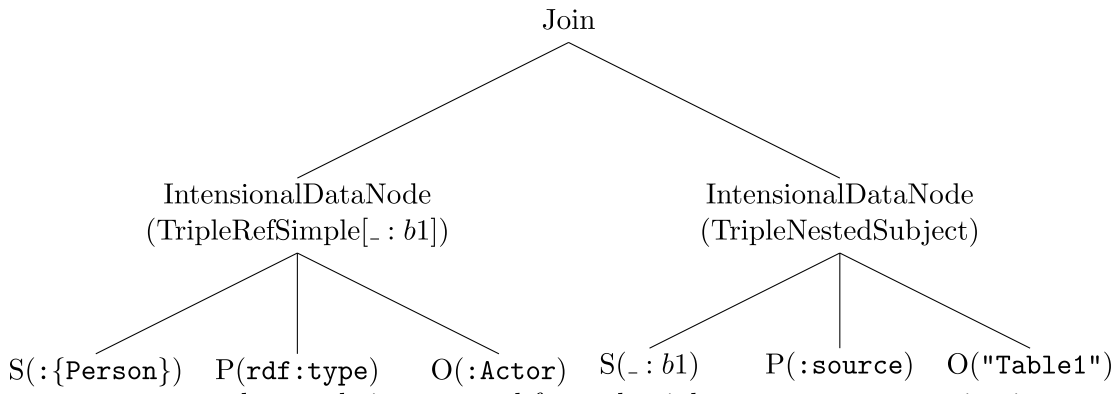


Here the node's predicate is denoted within parentheses (Triple), and the node's data shown by the subject, predicate, object triple below. ♢

Example 5.4 More common in VKG systems is that an intensional data node represents the target part of a mapping. The R2RML-star mapping

```
(<<:{Person}, rdf:type, :Actor>>, :source "Table1") <-
  SELECT Person FROM Table1
```

is read in and represented by the IQ:



As you can see one data node is generated for each triple-star pattern occurring in the query. ♢

5.3 Rewriting SPARQL-star Queries in Ontop

We have extended Ontop's rewriting capabilities to apply not only to standard RDF triples, but to RDF-star triples as well. This was accomplished through allowing rewriting to be applied to the new data node types, those with predicates signifying RDF-star triples. We have elected to apply only simple rewriting, not complex rewriting, see Definitions 4.5 and 4.7. This is due to the current indecision of the community on if complex rewriting should be supported. See Example 4.8 for an illustration of the issues surrounding complex rewriting.

5.4 Unfolding SPARQL-star Queries in Ontop

In section 4.2.3 we describe to what extent a VKG system must be expanded to support the unfolding of SPARQL-star queries. In Ontop the query unfolding process consists of two steps. First the SPARQL query must match with the relevant mapping, in a correct way such that the right mappings match with the right query.

This is done through a process known as pattern matching. In practice in Ontop it works by checking that the query's IQ representation, or a sub-tree of the IQ, matches with the IQ representing the target part of the mapping, and then replacing that sub tree by the mapping's IQ. We delve into further detail on this in Section 5.4.2. This will populate the IQ being processed with RDF terms.

The second step is the conversion of intensional data nodes into extensional ones. All RDF terms and functions are replaced with data to be retrieved from the relational data sources and database functions before these queries are sent off to their respective DBMS. This is done by replacing the parts of the IQ being processed that are identical to the mappings' target parts with the relevant mappings' source parts.

5.4.1 Motivation for matching

Let us focus our discussion on the matching process and take the easiest example first: a standard SPARQL query, free of SPARQL-star grammar. Recall the typical s p o query, a SPARQL query whose `WHERE` clause is just `{s p o}`. Such a query will match with every triple of the queried dataset. As the query is free from RDF-star specific grammar it will be treated as any other standard SPARQL query during the first step of processing: parsing. The IQ arising will have one intensional data node: triple(s, p, o).

However, just because such a query is free from SPARQL-star grammar does not mean that it is not affected by our work. If the data queried contains quoted triples, in Ontop meaning the mapping is an R2RML-star mapping populating the virtual Knowledge Graph with quoted triples, then quoted triples should be bound to the answer variables. This means we must be able to match such standard SPARQL triples with R2RML-star mappings without changing the behaviour of the query when combined with standard R2RML mappings.

In the more complex case, the query posed is a SPARQL-star query having for example a quoted triple in the subject of a triple pattern. In this case the query should be matched with a suitable R2RML-star mapping and must not match with any standard R2RML mappings.

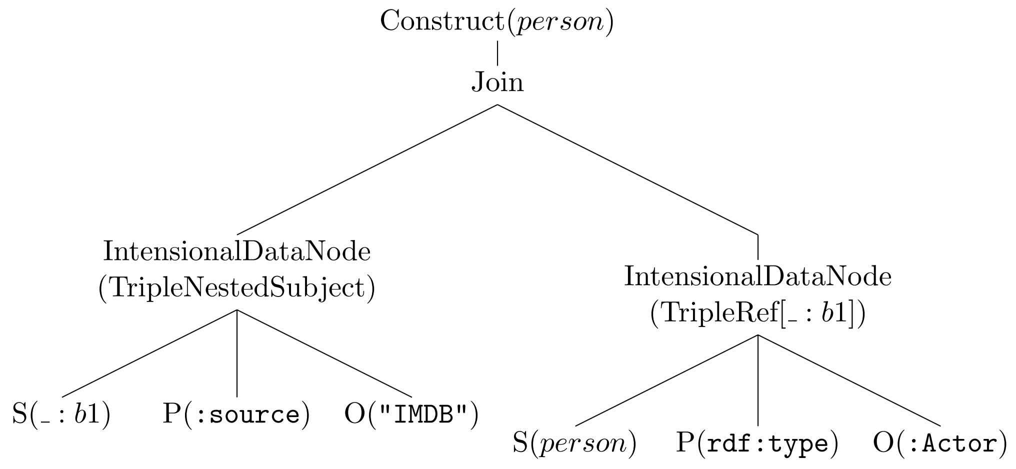
Example 5.5 In this example we use similar queries as in Examples 4.4, 4.6, and 4.9 from the previous chapter. Having a dataset consisting of the single triple `<< :John a :Actor >> :source "IMDB"`, we wish to run two SPARQL queries on this dataset. The SPARQL-star query:

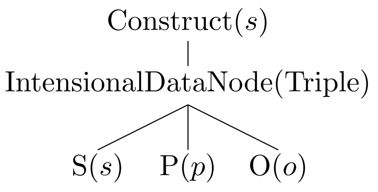
```
SELECT ?person
WHERE {<< ?person a :Actor >> :source "IMDB".}
```

and the standard SPARQL query:

```
SELECT ?s
WHERE {?s ?p ?o.}
```

They are read by Ontop and converted to the two IQs respectively:





◇

5.4.2 Translation from R2RML-star to IQ

To discuss pattern matching of SPARQL-star queries with R2RML-star mappings we must first describe how such mappings are parsed and read. In Section 4.1.1 we detail how we extend R2RML and in this section we will elaborate on how Ontop deals with R2RML-mappings.

For each R2RML mapping Ontop generates a set of internal Ontop specific mapping objects, call them *i-mappings*, for internal mapping. All the i-mappings generated from one R2RML-mapping have the same source part corresponding to the source part of the R2RML-mapping. The source part of the i-mapping is an IQ consisting of an extensional data node, representing the R2RML mapping's SQL query. As this thesis does not touch on the source part we will not discuss it further.

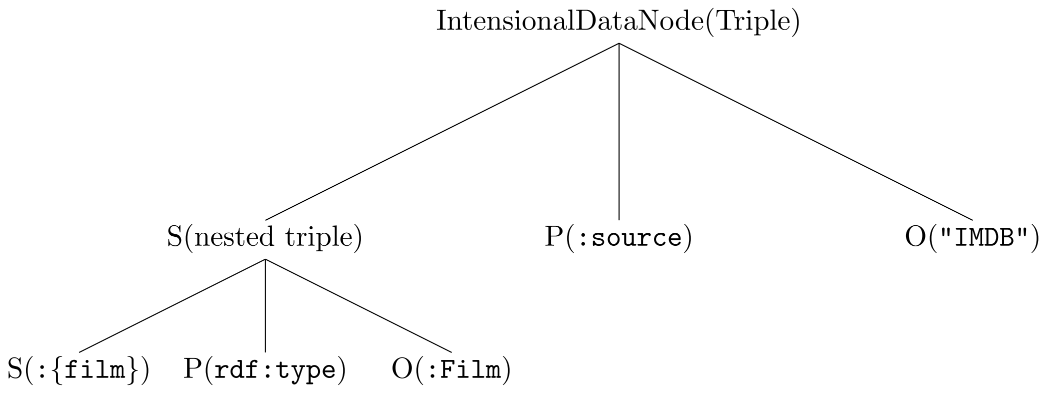
A single R2RML mapping can contain multiple triple or quad templates and the purpose of Ontop's i-mapping is to divide the R2RML mapping into smaller pieces where each piece, each i-mapping, contains one triple template in its target part. This also make them easier to process.

In the initial stage of query parsing an R2RML-star mapping is parsed the same way as a standard R2RML mapping. An IQ consisting of one intensional data node is created to be the target part of each i-mapping, having the predicate of triple. Should the term type of a subject or object be the RDF-star term type, the system knows that the mapping writer is using a quoted triple in that position.

The system will then read the specific RDF-star properties of subject, predicate, and object in order to populate the intensional data nodes with the quoted triple. This is done by essentially wrapping the three RDF-star terms just read in, to a specific object, a nested triple function, that fits in Ontop's typing system and can take the place of subject or object in the root triple. This process is recursive and multiple levels of nesting are possible as defined by the R2RML-star extension, see Section 4.1.1. This is compatible with recursively nested triples as specified by the RDF-star draft [15].

At this point the i-mapping can already match with a standard SPARQL query as in the example below.

Example 5.6 Consider Mapping 2 from the Appendix, we express its target part simplified here as: ([`:{film}`, `rdf:type`, `:Film`], `:source`, `"IMDB"`). When Ontop parses the mapping this is initially parsed just as any standard R2RML mapping, generating the IQ:



Due to the triple predicate in the intensional data node this matches to any SPARQL query that has a triple with query variables in the WHERE clause, e.g. asking for {`?s ?p ?o.`}. This would then introduce a quoted triple bound to the query variable `?s`. This IQ would not match to a SPARQL-star query explicitly asking for quoted triples, e.g. a WHERE clause containing {`<< ?s ?p1 ?o1 >> ?p2 ?o2.`}, as this query would be parsed into an entirely different IQ and the process of pattern matching fails. ♢

Extending the translation

In order to match R2RML-star mappings with SPARQL-star queries this translation process needs to be extended further. For each i-mapping containing an intensional data node that has a quoted triple in the subject or object position, a new i-mapping with the same source part and a new target part is created. This creation of a new target IQ follows the following algorithm:

1. Check if an intensional data node in the IQ representing the i-mapping's target part has a nested triple function in the subject position, if yes proceed to Step 2, otherwise to Step 5.

2. Use the subject, predicate, and object of the nested triple function to create a new intensional data node with the simple tripleRef predicate, give it a unique IRI as reference.

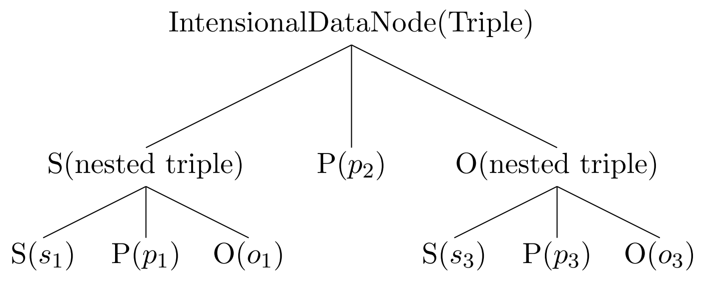
3. Replace the nested triple function with the unique IRI created in the previous step. Change the original node's predicate to add the property of having a nested subject.

4. Replace the original intensional node's position in the IQ with a new tree consisting of a JOIN as parent, and the two newly created nodes below.

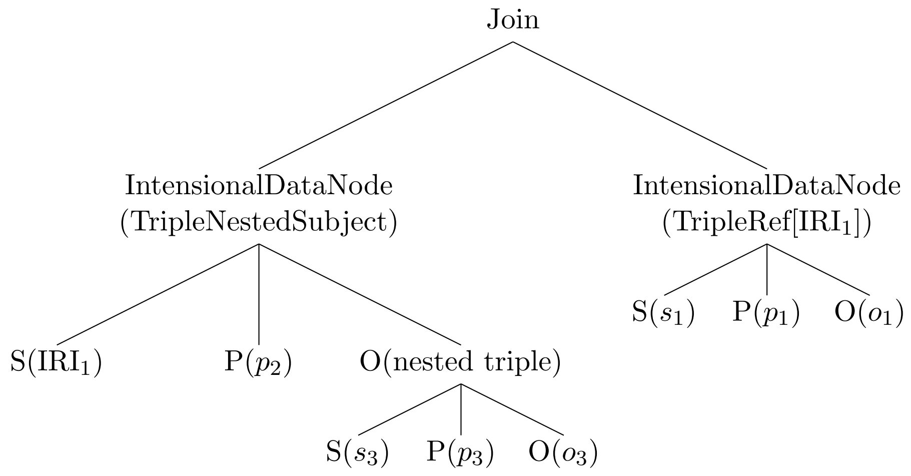
5. If no nested triple function was found in Step 1, look for a nested triple function in the object position. Proceed in a similar manner as Steps 2-4 did but now considering the object as nested.

Note that this algorithm is applied to each i-mapping and can thus be considered recursive as it is applied even to i-mappings already generated by the algorithm. This means that it also handles RDF-star triples that contain quoted triples in both the subject and object positions.

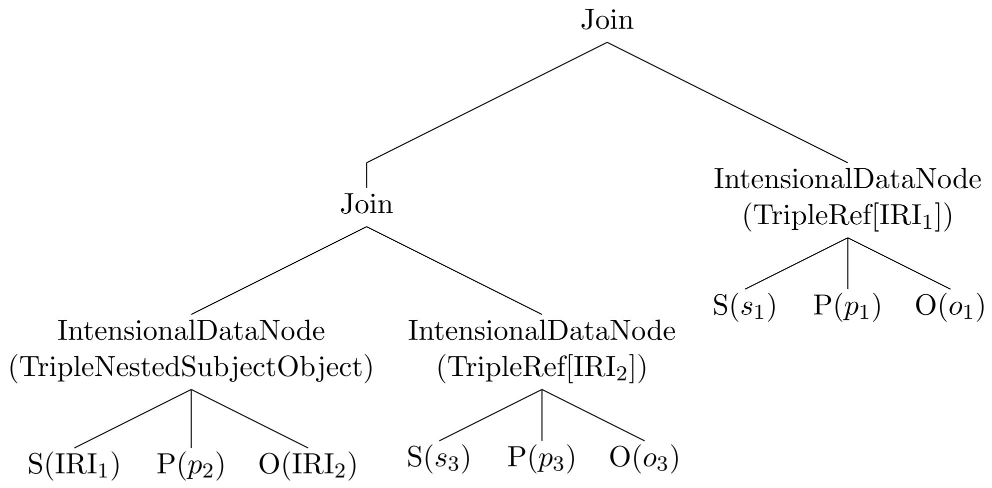
Example 5.7 Consider the i-mapping target represented by the following IQ:



Such an IQ will already match with IQs generated by standard SPARQL queries asking for triples. It will however not match with any SPARQL-star queries specifically asking for nested triples. When the algorithm described above is applied to this IQ the result is a new IQ:



At this point the IQ will match sub trees of IQs representing SPARQL-star queries containing a nested triple in the subject position in the where clause. A further pass of the algorithm produces the following IQ:


Join
Join
IntensionalDataNode (TripleRef[IRI_1])
S(s_1)
P(p_1)
O(o_1)
IntensionalDataNode (TripleNestedSubjectObject)
IntensionalDataNode (TripleRef[IRI_2])
S(IRI_1)
P(p_2)
O(IRI_2)
S(s_3)
P(p_3)
O(o_3)

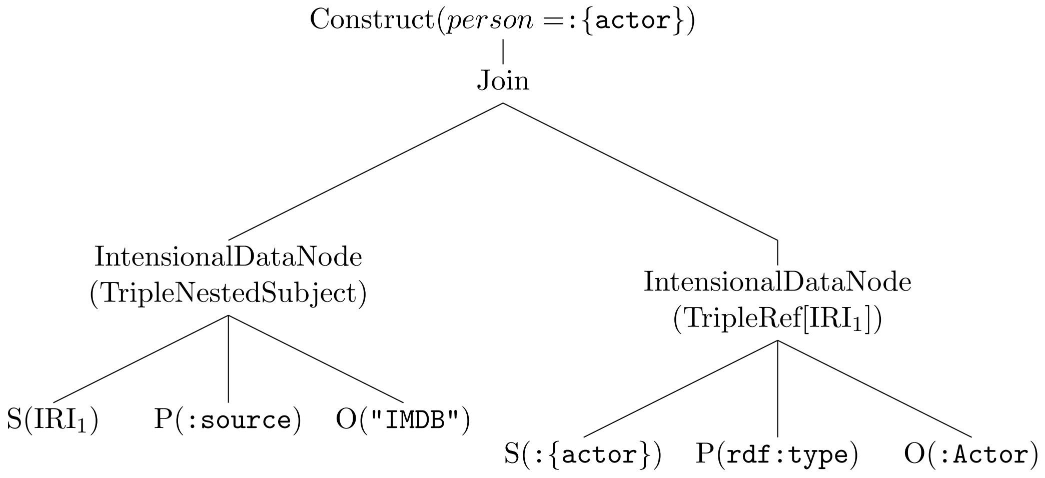
One can imagine that for a deeply nested structure a lot of join nodes are produced. Luckily the IQ's join query node is n-ary and can have more than two children. At a later stage of query optimization such structures are simplified and every data node is lifted up to sit under one parent join node. ♢

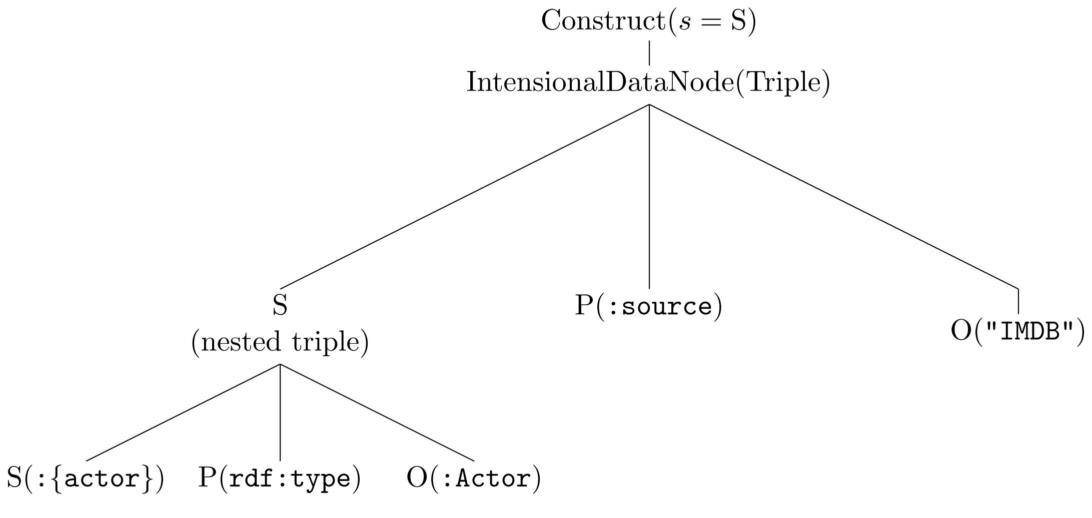
5.4.3 Conversion from intensional to extensional data node

After the query's IQ has been matched with the IQ's of each relevant mapping, the unfolding process must continue in order to execute SQL queries over the data sources. All RDF terms, functions, and operators must be converted to database terms and functions or otherwise lifted up to the top of the query so that the DBMS does not get passed any RDF terms.

Ontop already supports this process but through our work we have introduced a new term, namely the quoted triple, this is internally represented through a nested triple function. At this time we have been unable to implement support for translating this function in Ontop. This means that only a limited amount of query and mapping combinations are supported at this time, namely those queries and mappings that match without introducing the nested triple function to the query's IQ.

Example 5.8 Consider a VKG instance with only one mapping having the target part: ([`:{actor}`, `rdf:type`, `:Actor`], `:source`, `"IMDB"`). Wishing to run the two queries from Example 5.5, they will both match with the mapping according to the process we have outlined above. halfway through the unfolding process we end up with the following two IQs:


Construct($person$ = :{actor})
Join
IntensionalDataNode (TripleNestedSubject)
S(IRI$_1$)
P(:source)
O("IMDB")
IntensionalDataNode (TripleRef[IRI$_1$])
S(:{actor})
P(rdf:type)
O(:Actor)



Of these queries the first one will pass through the second stage of unfolding, execute correctly over the data sources, and return the expected answers. The second query will fail due to the current inability of Ontop to translate the nested triple function. $\diamondsuit$

5.5 Supporting SPARQL-star Answers in Ontop

In Section 4.2.4 we described how VKG systems should support additional query answers under SPARQL-star. Our answer to this question ties to Section 5.4 on how Ontop unfolds SPARQL-star queries and translates R2RML-star mappings. In order to support binding quoted triples to SPARQL answer variables there must be support for the nested triple function at the latter stage of unfolding. As we have not yet reached that milestone, Ontop at the moment only supports returning standard answers to SPARQL ASK and SELECT queries, with standard RDF terms bound to the answer variables.

Similarly there has been no work done to implement SPARQL-star specific answers to the other types of query, ASK, DESCRIBE, or CONSTRUCT, and they have limited functionality. DESCRIBE and CONSTRUCT can not create RDF-star graphs under Ontop and ASK only functions under a VKG environment that does not give rise to the nested triple function as described in the previous section.

5.6 Supporting SPARQL-star Functions and Operators

Section 4.2.5 describes how standard SPARQL functions and operators must be extended under SPARQL-star, and newly defined operators and functions. Due to time constraints we have not been able to implement this functionality in Ontop during the work of this thesis, although we have laid some groundwork.

The extensions necessary fall in two categories. Redefining the standard SPARQL functions is one of them, and for those the domain must be extended to RDF-star terms and their behaviour should be defined for such terms. For example, the BOUND function should return true if a query variable is bound to an RDF-star term such as a quoted triple [15].

The second category contains new functions such as isTRIPLE which takes in one RDF-star term and returns a boolean indicating if the term is a triple or not [15]. For these functions the groundwork has been laid in the typing system outlined in Sections 5.2.1 and 5.4.2. In the specific example one can check if the term as it is represented in the IQ is either the nested triple function or holds a reference to an intensional data node with a type of tripleRef predicate.

6 Conclusion

This chapter forms the final part of the thesis. In it we summarise the results of our main chapters, and end by discussing further work.

6.1 Results

The work of this thesis consists of three main parts. First a description of RDF-star which adds value to a recent field of research with few publications taking place in Chapter 3. Second an investigation into how to extend VKG systems with RDF-star support, this takes place in Chapter 4. The third part concerns extending the specific VKG system Ontop with RDF-star support and is detailed in Chapter 5.

6.1.1 Theoretical investigation

In order to extend the virtual knowledge graph to an RDF-star graph we have chosen to extend the mapping, specifically the mapping language R2RML. As the mapping form the basis of creating the virtual graph it seems the natural candidate component. To this end we propose an extension to the standard mapping language R2RML called R2RML-star which allows for the generation of RDF-star triples from database sourced facts.

We detail the impact this has on the query processing of the VKG system including the rewriting and unfolding processes. We further describe other ways in which the system should be extended to fully support RDF- and SPARQL-star, such as new SPARQL-star functions and operators, and extending the query answers.

6.1.2 Extending Ontop

The results presented here are explained in more detail in the various sections of Chapter 5. What has been implemented and is functioning is support for parsing SPARQL-star queries, parsing R2RML-star mappings, and matching the SPARQL-star queries and R2RML-star mappings in a correct way. Unfolding of the query is then supported provided that the queries and mappings map in an exact way and no quoted triples are bound to variables.

Query rewriting of the simple kind from Definition 4.5 is supported. Complex rewriting as in Definition 4.7 remains unsupported as it is unclear if this should be implemented.

The work done for the implemented functionality includes extending Ontop's internal typing system with RDF-star terms and writing test cases, which lays the foundation for further implementing the missing functionality.

6.2 Further Work

We split this section into two parts corresponding to the two main parts of this thesis. The theoretical investigation into applying RDF-star to VKG systems, and the implementation of these ideas to the VKG system Ontop.

6.2.1 Further theoretical work

In this thesis we have not exhausted all the possibilities of applying RDF-star to VKG systems. While we have extended the standard mapping language R2RML in order to add RDF-star support to the mapping of the VKG system, there is another major component left untouched. The ontology could plausibly be extended with support for RDF-star, a work that is left out of this thesis. A theoretical investigation should be made into the applications of RDF-star to the VKG ontology, including if one should extend the standard ontology languages such as RDFS or OWL2QL using RDF-star.

We have not described how to apply query rewriting in the case of complex RDF-star rewriting, as this case is still an open discussion, see Section 4.2.2.

Further, our thesis deals with the extension of VKG system's contextual data capabilities with the help of RDF-star. This necessarily excludes any VKG system not using the RDF language for their knowledge graph. As RDF is a widely used W3C standard we still maintain that this limitation of scope was a correct choice, but we welcome further research into enhancing the contextual data capabilities of VKG systems using other data models.

6.2.2 Further implementation work

Not all of our conclusions of Chapter 4 have been implemented in Ontop. The implementation issue of unfolding mappings that introduce a variable binding to a quoted triple remains unsolved. Also remaining to implement is support for the new SPARQL-star functions, operators, and answer formats. For these additional SPARQL-star features the groundwork has been laid in the new typing system we have developed. All further theoretical investigations suggested in Section 6.2.1 above could also eventually be implemented in Ontop.

Reference List

[1] Diego Calvanese et al. “Ontop: Answering SPARQL Queries over Relational Databases”. In: *Semantic Web J.* 8.3 (2017), pp. 471–487. DOI: 10.3233/SW-160217.

[2] Guohui Xiao et al. “Virtual Knowledge Graphs: An Overview of Systems and Use Cases”. In: *Data Intelligence* 1.3 (2019), pp. 201–223. DOI: 10.1162/dint_a_00011.

[3] Guohui Xiao et al. “The Virtual Knowledge Graph System Ontop”. In: *Proc. of the 19th Int. Semantic Web Conf. (ISWC 2020).* Vol. 12507. Lecture Notes in Computer Science. Springer, 2020, pp. 259–277. DOI: 10.1007/978-3-030-62466-8_17.

[4] David Wood, Markus Lanthaler, and Richard Cyganiak. *RDF 1.1 Concepts and Abstract Syntax.* W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[5] *SPARQL 1.1 Overview.* W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-overview-20130321/.

[6] Andy Seaborne and Steven Harris. *SPARQL 1.1 Query Language.* W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-query-20130321/.

[7] Dave Beckett and Jeen Broekstra. *SPARQL Query Results XML Format (Second Edition).* W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-rdf-sparql-XMLres-20130321/.

[8] Andy Seaborne. *SPARQL 1.1 Query Results JSON Format.* W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-results-json-20130321/.

[9] Richard Cyganiak, Seema Sundara, and Souripriya Das. *R2RML: RDB to RDF Mapping Language.* W3C Recommendation. W3C, Sept. 2012. URL: https://www.w3.org/TR/2012/REC-r2rml-20120927/.

[10] Andy Seaborne. *SPARQL 1.1 Query Results CSV and TSV Formats.* W3C Recommendation. W3C, Mar. 2013. URL: https://www.w3.org/TR/2013/REC-sparql11-results-csv-tsv-20130321/.

[11] Dan Brickley and Ramanathan Guha. *RDF Schema 1.1.* W3C Recommendation. W3C, Feb. 2014. URL: https://www.w3.org/TR/2014/REC-rdf-schema-20140225/.

[12] Chimezie Ogbuji and Birte Glimm. *SPARQL 1.1 Entailment Regimes.* W3C Recommendation. https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/. W3C, Mar. 2013.

[13] *OWL 2 Web Ontology Language Document Overview (Second Edition).* W3C Recommendation. W3C, Dec. 2012. URL: `https://www.w3.org/TR/2012/REC-owl2-overview-20121211/`.

[14] Eric Prud'hommeaux and Gavin Carothers. *RDF 1.1 Turtle.* W3C Recommendation. W3C, Feb. 2014. URL: `https://www.w3.org/TR/2014/REC-turtle-20140225/`.

[15] Olaf Hartig et al. *RDF-star and SPARQL-star.* W3C Draft Community Group Report. W3C Community, July 2021. URL: `https://w3c.github.io/rdf-star/cg-spec/2021-07-01.html`.

[16] Anastasia Dimou and Miel Van Der Sande. *RDF Mapping Language (RML).* Unofficial W3C Draft. Oct. 2020. URL: `https://rml.io/specs/rml/`.

[17] Lisa Ehrlinger and Wolfram Wöß. "Towards a Definition of Knowledge Graphs". In: Sept. 2016.

[18] Michael Färber et al. "Linked data quality of DBpedia, Freebase, OpenCyc, Wikidata, and YAGO". In: *Semantic Web* 9 (Mar. 2017), pp. 1–53. DOI: `10.3233/SW-170275`.

[19] AnHai Doan, Alon Halevy, and Zachary Ives. *Principles of Data Integration.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 0124160441.

[20] *The Description Logic Handbook: Theory, Implementation and Applications.* 2nd ed. Cambridge University Press, 2007. DOI: `10.1017/CBO9780511711787`.

[21] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level.* 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201537710.

[22] Ontop. *Ontop - Main Features.* 2021. URL: `https://ontop-vkg.org/guide/#main-features` (visited on 05/21/2021).

[23] Ontop. *Formal characterization of IQs.* 2020. URL: `https://ontop-vkg.org/research/iq-formal.html` (visited on 10/03/2021).

[24] Ontop. *Intermediate Query.* 2020. URL: `https://ontop-vkg.org/dev/internals/iq.html` (visited on 10/03/2021).

[25] RDF4J Team. *RDF4J Documentation.* 2021. URL: `https://rdf4j.org/documentation/` (visited on 08/09/2021).

[26] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. "Semantics and Complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009). ISSN: 0362-5915. DOI: `10.1145/1567274.1567278`. URL: `https://doi.org/10.1145/1567274.1567278`.

[27] Olaf Hartig. “Foundations of RDF* and SPARQL* (An Alternative Approach to Statement-Level Metadata in RDF)”. In: *AMW*. 2017. URL: http://ceur-ws.org/Vol-1912/paper12.pdf (visited on 07/07/2021).

[28] Thomas Delva et al. “RML-star: A Declarative Mapping Language for RDF-star Generation”. In: Aug. 2021.

[29] Guohui Xiao et al. “Ontology-Based Data Access: A Survey”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 5511–5519. DOI: 10.24963/ijcai.2018/777. URL: https://doi.org/10.24963/ijcai.2018/777.

Reading List

[30] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. "Reifying RDF: What Works Well With Wikidata?" In: *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems*. Ed. by Thorsten Liebig and Achille Fokoue. Vol. 1457. CEUR Workshop Proceedings. CEUR-WS.org, 2015, pp. 32–47.

[31] Claudio Gutierrez et al. "Foundations of Semantic Web databases". In: *Journal of Computer and System Sciences* 77.3 (2011). Database Theory, pp. 520–541. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/j.jcss.2010.04.009`. URL: `https://www.sciencedirect.com/science/article/pii/S0022000010000516`.

[32] Graham Klyne and Jeremy Carroll. "Resource Description Framework (RDF): Concepts and Abstract Syntax". In: *World Wide Web Consortium* 10 (Jan. 2006).

A RDF Graphs

Note that all of these graphs are written in the Turtle syntax. We imagine the following prefixes:

```
@prefix : <http://lukas.thesis.org/films#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@base <http://lukas.thesis.org/films#> .
```

RDF Graph 1 (Barebones film ontology)

```
<http://lukas.thesis.org/films> rdf:type owl:Ontology .

# Object Properties
:directedBy rdf:type owl:ObjectProperty ;
            rdfs:domain :Film ;
            rdfs:range :Director .
:stars rdf:type owl:ObjectProperty ;
       rdfs:domain :Film ;
       rdfs:range :Actor .

# Data properties
:releasedIn rdf:type owl:DatatypeProperty ;
            rdfs:domain :Film .

# Classes
:Actor rdf:type owl:Class ;
       rdfs:subClassOf :Person .
:Director rdf:type owl:Class ;
          rdfs:subClassOf :Person .
:Film rdf:type owl:Class .
:Person rdf:type owl:Class .
```

B R2RML Mappings

In addition to the prefixes of appendix A, we add two more:

```
@prefix rr: <http://www.w3.org/ns/r2rml#>.
@prefix ex: <http://example.com/ns#>.
@prefix star: < https://w3id.org/obda/r2rmlstar#>.
```

Mapping 1 (Film class with name and release year)

```
<#TriplesMap1>
    rr:logicalTable [ rr:tableName "imdbTable1" ];
    rr:subjectMap [
        rr:template "http://lukas.thesis.org/films/{Name}{Year}";
        rr:class :Film;
    ];
    rr:predicateObjectMap [
        rr:predicate ex:name;
        rr:objectMap [ rr:column "Name" ];
    ];
    rr:predicateObjectMap [
        rr:predicate :releasedIn;
        rr:objectMap [ rr:column "Year"];
    ].
```

Mapping 2 (R2RML-star example 1)

```
<urn:MAPID-film-source> a rr:TriplesMap;
  rr:logicalTable [ a rr:R2RMLView;
      rr:sqlQuery "SELECT name, year FROM imdbTable1;"
    ];
  rr:subjectMap [ a rr:SubjectMap;
      rr:termType star:RDFStarTermType;
      star:subject [a rr:ObjectMap;
        rr:template "http://lukas.thesis.org/films/{Name}{Year}";
        ];
      star:predicate [a rr:PredicateMap;
        rr:constant rdf:type
        ];
      star:object [a rr:ObjectMap;
        rr:constant :Film
        ];
    ];
  rr:predicateObjectMap [
      rr:predicate :source;
      rr:object "IMDB database";
    ].
```

Mapping 3 (R2RML-star example 2)

```
<urn:MAPID-film-mentioned> a rr:TriplesMap;
  rr:logicalTable [ a rr:R2RMLView;
      rr:sqlQuery "SELECT name, year FROM professors;"
    ];
```

```
rr:subjectMap [ a rr:SubjectMap;
    rr:template "http://lukas.thesis.org/films/{Name}{Year}";
    rr:class :Film;
  ];
rr:predicateObjectMap [
    rr:predicate :isMentionedBy;
    rr:objectMap [  a rr:ObjectMap;
      rr:termType star:RDFStarTermType;
      star:subject [a rr:ObjectMap;
          rr:template "http://lukas.thesis.org/films/{Name}{Year}";
          rr:termType rr:IRI;
        ];
      star:predicate [a rr:PredicateMap;
          rr:constant :releasedIn;
        ];
      star:object [a rr:ObjectMap;
          rr:column "year";
          rr:termType rr:Literal;
        ];
      ];
  ].
```