SPARQL Query Rewriting with Paths [Master's Thesis]

Abdullah Abbas

Master Thesis

Master of Science in Informatics at Grenoble (MoSIG)
Master Specialty : Artificial Intelligence and the Web (AIW)

University:
Grenoble INP - Ensimag
Laboratory of internship:
Inria Grenoble - TYREX Team

SPARQL Query Rewriting with Paths

Abdullah Abbas

supervised by
Dr. Jérôme Euzenat, INRIA
Dr. Nabil Layaida, INRIA
Dr. Pierre Genevès, CNRS

jury composed of
Prof. Catherine Berrut
Prof. Éric Gaussier
Dr. Noha Ibrahim
Dr. Jean-Marc Vincent
Dr. Pierre Genevès
Dr. Jean-Yves Vion-Dury

Defense date: *June 23, 2014*

Abstract

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It involves publishing in languages specifically designed for data like (RDF) Resource Description Framework. In order to access the published data, it offers a query language named SPARQL.

The goal of this study is to transform SPARQL queries to other SPARQL queries which can be executed more efficiently. Our main goal of transformation is to eliminate non-distinguished variables, which are source of extra complexity, where such elimination is possible. We rewrite SPARQL queries with property paths, which was introduced in SPARQL 1.1.

Acknowledgements

I would like to thank all the people who participated in different ways and helped me to accomplish my internship. First of all, I want to express my gratitude to my supervisors Dr. Jérôme Euzenat, Dr. Pierre Genevès, and Dr. Nabil Layaida for their support, constructive remarks, and continuous sharing of knowledge and expertise along this work. I also thank my colleagues and other interns I have met during this internship. I thank all of them for providing a very causal and comfortable working atmosphere. I thank INRIA for hosting me during my internship period. I thank all my teachers during my master studies at Grenoble INP, and of course, I thank all my family for their great support all the time.

It was a real pleasure working with each of these persons and I wish I was the person they expected me to be.

Contents

1 Introduction **3**

2 Background **5**
- 2.1 RDF . . . 5
  - 2.1.1 RDF Syntax . . . 5
  - 2.1.2 RDF Semantics . . . 6
  - 2.1.3 Inference Mechanism . . . 7
- 2.2 SPARQL . . . 8
  - 2.2.1 SPARQL Syntax . . . 9
  - 2.2.2 SPARQL Semantics . . . 10
  - 2.2.3 Algebraic Manipulation . . . 11
- 2.3 SPARQL with Property Paths . . . 12
  - 2.3.1 Property Paths in SPARQL 1.1 . . . 13
- 2.4 Conclusion . . . 14

3 SPARQL Query Rewriting **15**
- 3.1 Introduction to Eliminating Non-Distinguished Variables . . . 15
  - 3.1.1 The Purpose of Eliminating Non-Distinguished Variables . . . 15
  - 3.1.2 Challenges Overview . . . 16
  - 3.1.3 Methodology . . . 17
- 3.2 Elementary Transformation Rules . . . 17
  - 3.2.1 Single PP Pattern with a Non-Distinguished Variable . . . 18
  - 3.2.2 Double PP Patterns Sharing a Non-Distinguished Variable . . . 19
  - 3.2.3 Multiple PP Patterns Sharing a Non-Distinguished Variable . . . 19
- 3.3 Query Transformation . . . 21
  - 3.3.1 Case Study . . . 21
  - 3.3.2 Variable Eliminating Function . . . 25
  - 3.3.3 Query Transformation Function . . . 25
  - 3.3.4 Transformation Complexity . . . 26
  - 3.3.5 Graphical Representation Summary . . . 27
- 3.4 Constrained Datasets . . . 28
  - 3.4.1 Various Transformation Suggestions . . . 28
  - 3.4.2 Defining the Constraints . . . 31
- 3.5 Dealing with Non-Constrained Datasets . . . 34
  - 3.5.1 Nested FILTERs . . . 36
- 3.6 Conclusion . . . 39

4 Conclusion **40**

Chapter 1

Introduction

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. Its main purpose is driving the evolution of the current Web by enabling users to find, share, and combine information more easily. It involves publishing in data/knowledge representation languages designed for the web: Resource Description Framework (RDF), Web Ontology Language (OWL), and Extensible Markup Language (XML). These languages can describe arbitrary things such as people, meetings, or machine parts.

In order to access the published data, it offers a query language named SPARQL. SPARQL has many points in common with SQL except that, instead of dealing with relational tables, it deals with RDF graphs.

An example of a SPARQL SELECT query is given below:

```
SELECT ?book ?price
WHERE
{
  ?book rdf:type exp:book.
  ?book exp:author ?author.
  ?author exp:name "Donald Knuth".
  ?library exp:sells ?book.
  ?library exp:in ?city.
  ?city exp:name "Paris"
}
OPTIONAL
{
  ?book exp:in ?offer.
  ?offer exp:price ?price
  FILTER (?price<10)
}
```

Many studies have been carried out in order to both extend the expressiveness of the language and reduce the complexity of query evaluation.

Path-based languages have been provided and extend the expressiveness of SPARQL at no extra computational cost [2]. Paths are now introduced in the new version of SPARQL (1.1) [11].

The SELECT operator is equivalent to projection in relational algebra. This operator is a source of extra complexity in query evaluation: NP for queries composed of basic graph patterns [16, 20]. If this operator can be suppressed, the complexity reduces to just PTIME [20]. So considering the possibility of rewriting SPARQL queries with projection into SPARQL queries without projection is interesting because it would guarantee a lower complexity.

Our purpose in this document is to study, and propose a query transformation methodology, to eliminate non-distinguished variables, i.e., suppressing the projection operator. Non-distinguished variables are the variables that are not part of the result of the query, such as `?author`, `?library`, `?city`, and `?offer` in the previous example.

The previous example can be written without non-distinguished variables, using paths, as follows:

```
SELECT ?book ?price
WHERE
{
  ?book rdf:type exp:book.
  ?book exp:author/exp:name "Donald Knuth".
  ?book ^exp:sells/exp:in/exp:name "Paris"
}
OPTIONAL
{
  ?book exp:in/exp:price ?price
  FILTER (?price<10)
}
```

Both previous queries have exactly the same meaning, and give the same results. The second query may not be intuitive to write, but it has a computation complexity advantage on the first query.

In general, it may not be intuitive for users to write queries without non-distinguished variables. Such variables, which are source of extra complexity, allow users to write meaningful queries more easily. For this reason we suggest automatic transformation of queries, on fly, before their execution.

In this document, we define a query transformation function that takes a query, and returns a transformed query according to a set of rules in order to eliminate non-distinguished variables where such elimination is possible. The transformation was not possible for some cases, and thus we introduced some constraints on the queried data in order to find sound and complete transformations for such cases.

To deal with non-constrained datasets (any possible dataset), we proposed an alternative solution that benefits from our transformation function, but re-introduces the non-distinguished variables in a way that promises a lower computation time than the original query, and only witnesses the same computation time in the worst case.

Chapter 2

Background

Most of the content of this chapter closely follows the "Semantic Web" lecture notes by Jérôme Euzenat. [8]

2.1 RDF

The Resource Description Framework (RDF) is a language for expressing information about resources in the World Wide Web. Resources can be anything, including documents, people, physical objects, and abstract concepts.

RDF is a W3C recommendation.

The RDF 1.1 (current version) specification consists of a suite of W3C Recommendations including (but not limited to) *RDF 1.1 Concepts and Abstract Syntax* [13], *RDF 1.1 XML Syntax* [9], and *RDF 1.1 Semantics* [15].

RDF allows us to make statements about resources. The format of these statements is simple. A statement always has the following structure:

```
<subject> <predicate> <object>
```

An RDF statement expresses a relationship between two resources. The subject and the object represent the two resources being related; the predicate represents the nature of their relationship. The relationship is phrased in a directional way (from subject to object) and is called in RDF a property.

A collection of RDF statements (RDF triples) can be intuitively understood as a directed labeled graph: resources are nodes and statements are arcs (from the subject node to the object node) connecting the nodes.

Our study in this document refers to the notion of Simple RDF, i.e., RDF without specific (RDF or RDFS) vocabulary [14]. Our decision is based on the fact that RDF and RDFS consequences (or entailments) can be polynomially reduced to simple entailment via RDF or RDFS rules [4, 22].

2.1.1 RDF Syntax

RDF is based on the idea of identifying things using Web identifiers (called Internationalized Resource Identifiers, or IRIs), and describing resources in terms of simple properties and property values.

RDF can be expressed in a variety of formats including RDF/XML, Turtle, TriG, N-Triples, and N-Quads.

In this section we use an *abstract syntax* [13], i.e. a data model that is independent of a particular concrete syntax (the syntax used to represent triples stored in text files). Different concrete syntaxes may produce exactly the same graph from the perspective of the abstract syntax. The semantics of RDF graphs in the next section are defined in terms of this abstract syntax.

To define the syntax of RDF, we need to introduce the terminology over which RDF graphs are constructed.

Definition 2.1.1 (RDF terminology). *The RDF terminology $\mathcal{T}$ is the union of three pairwise disjoint infinite sets of terms:*

- *the set $\mathcal{I}$ of IRIs*[1],
- *the set $\mathcal{L}$ of literals (itself partitioned into two sets, the set $\mathcal{L}_p$ of plain literals and the set $\mathcal{L}_t$ of typed literals), and*
- *the set $\mathcal{B}$ of blank nodes.*

The set $\mathcal{V} = \mathcal{I} \cup \mathcal{L}$ of names is called the vocabulary.

Notation. If G is an RDF graph, we use $\mathcal{T}(G)$, $\mathcal{I}(G)$, $\mathcal{L}(G)$, $\mathcal{B}(G)$, $\mathcal{V}(G)$ to denote the set of terms, IRIs, literals, variables or names that appear in at least one triple of G.

Now we define RDF Graphs.

Definition 2.1.2 (RDF graph). *An RDF triple is an element of $(\mathcal{I} \cup \mathcal{B}) \times \mathcal{I} \times \mathcal{T}$. An RDF graph is a finite set of RDF triples.*

It is sometimes convenient to loosen the requirements on RDF triples. For example, the completeness of the RDFS entailment rules is easier to show with a generalization of RDF triples [13].

A GRDF (Generalized RDF) graph is a generalization of an RDF graph defined as follows:

Definition 2.1.3 (GRDF graph). *An GRDF triple is an element of $\mathcal{T} \times (\mathcal{I} \cup \mathcal{B}) \times \mathcal{T}$. An GRDF graph is a finite set of GRDF triples.*

2.1.2 RDF Semantics

This section is devoted to RDF Semantics. We assert again here that we are referring to simple RDF semantics without RDF/RDFS vocabulary.

An interpretation describes possible way(s) the world might be in order to determine the truth value of any ground RDF graph. It does this by specifying for each IRI, what its denotation is. In addition, if it is used to indicate a property, what values that property has for each thing in the universe.

Definition 2.1.4 (Interpretation of a vocabulary). *Let $V \subseteq \mathcal{V} = \mathcal{I} \cup \mathcal{L}$ be a vocabulary, an interpretation of V is a quadruple $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ such that:*

[1]An IRI (Internationalized Resource Identifier) within an RDF graph is a Unicode string that conforms to the syntax defined in RFC 3987[7]. IRIs are a generalization of URIs that permits a wider range of Unicode characters. Every absolute URI and URL is an IRI, but not every IRI is an URI.

- *I_R is a set of resources that contains $V \cap \mathcal{L}$;*
- *$I_P \subseteq I_R$ is a set of properties;*
- *$I_{EXT} : I_P \rightarrow 2^{I_R \times I_R}$ associates to each property a set of pairs of resources called the extension of the property;*
- *the interpretation function $\iota : V \rightarrow I_R$ associates to each name in V a resource of I_R, if $v \in \mathcal{L}$, then $\iota(v) = v$.*

By providing RDF with a formal semantics, [9] expresses the conditions under which an interpretation is a model for an RDF graph. The usual notions of validity, satisfiability and consequence are entirely determined by these conditions.

Intuitively, a ground triple $\langle s, p, o \rangle$ in a GRDF graph will be true under the interpretation I if p is interpreted as a property (for example, r_p), s and o are interpreted as resources (for example, r_s and r_o, respectively), and the pair of resources $\langle r_s, r_o \rangle$ belongs to the extension of the property r_p. A triple $\langle s, p, ?b \rangle$ with the variable $?b \in \mathcal{B}$ would be true under I if there exists a resource r_b such that the pair $\langle r_s, r_b \rangle$ belongs to the extension r_p. When interpreting a variable node, an arbitrary resource can be chosen. To ensure that a variable always is interpreted by the same resource, extensions of the interpretation function is defined as follows.

Definition 2.1.5 (Extension to variables)**.** *Let $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ be an interpretation of a vocabulary $V \subseteq \mathcal{V}$, and $B \subseteq \mathcal{B}$ a set of variables. An extension of ι to B is a mapping $\iota' : \mathcal{V} \cup B \rightarrow I_R$ such that $\forall x \in V, \iota'(x) = \iota(x)$.*

An interpretation I is a model of GRDF graph G if all triples are true under I.

Definition 2.1.6 (Model of a GRDF graph)**.** *Let $V \subseteq \mathcal{V}$ be a vocabulary, and G be a GRDF graph such that every name appearing in G is also in V ($\mathcal{V}(G) \subseteq V$). An interpretation $I = \langle I_R, I_P, I_{EXT}, \iota \rangle$ of V is a model of G iff there exists an extension ι' that extends ι to $\mathcal{B}(G)$ such that for each triple $\langle s, p, o \rangle$ of G, $\iota'(p) \in I_P$ and $\langle \iota'(s), \iota'(o) \rangle \in I_{EXT}(\iota'(p))$. The mapping ι' is called a proof of G in I.*

Now we define the notion of consequence of graph. This definition is the standard model-theoretic definition of consequence.

Definition 2.1.7 (Consequence)**.** *A graph G' is a consequence of a graph G, or G entails G', denoted $G \models_{GRDF} G'$, iff every model of G is also a model of G'.*

The main result for simple RDF inference is:

Interpolation lemma ([12])**.** S entails a graph E if and only if a subgraph of S is an instance of E

Proof of the previous lemma can be found in [12]. □

2.1.3 Inference Mechanism

Simple RDF entailment can be characterized as a kind of graph homomorphism. A graph homomorphism from an RDF graph H into an RDF graph G, as defined in [4, 10], is a mapping π from the nodes of H into the nodes of G preserving the arc structure, i.e., for each node $x \in H$, if $\lambda(x) \in \mathcal{I} \cup \mathcal{L}$ then $\lambda(\pi(x)) = \lambda(x)$; and each arc $x \xrightarrow{p} y$ is mapped to $\pi(x) \xrightarrow{\pi(p)} \pi(y)$. This definition is similar to the projection used to characterize entailment of conceptual graphs [5] (see [6] for precise relationship between RDF and conceptual graphs). We modify this definition to the one that maps $\mathcal{T}(H)$ (terms of H) into $\mathcal{T}(G)$ (terms of G). Maps are used to ensure that a variable always mapped to the same term, as done for extensions to interpretations.

Definition 2.1.8 (Map)**.** *Let $V_1 \subseteq \mathcal{T}$, and $V_2 \subseteq \mathcal{T}$ be two sets of terms. A map from V_1 to V_2 is a mapping $\mu : V_1 \to V_2$ such that $\forall x \in (V_1 \cap \mathcal{V}), \mu(x) = x$.*

An RDF homomorphism is a map preserving the arc structure.

Definition 2.1.9 (GRDF homomorphism)**.** *Let G and H be two GRDF graphs. A GRDF homomorphism from H into G is a map π from $\mathcal{T}(H)$ to $\mathcal{T}(G)$ such that $\forall \langle s, p, o\rangle \in H, \langle \pi(s), \pi(p), \pi(o)\rangle \in G$.*

Theorem 2.1.1. Let G and H be two GRDF graphs, then $G \models_{GRDF} H$ iff there is a GRDF homomorphism from H into G.

Proof of this theorem is given in [1]. □

This equivalence between the semantic notion of entailment and the syntactic notion of homomorphism is the ground by which a correct and complete query answering procedure can be designed. More precisely, the set of answers to a GRDF graph query Q over an RDF knowledge base G are the set of RDF homomorphisms from Q into G which, by *Theorem 2.1.1*, correspond to RDF consequence.

2.2 SPARQL

RDF itself can be used as a query language for an RDF knowledge base using RDF consequence. Nonetheless, the use of consequence is still limited for answering queries. In particular, answering those that contain complex relations requires complex constructs. Therefore the need for added expressiveness in queries has led to define several query languages on top of graph patterns that are basically RDF and more precisely GRDF graphs.

There has been early proposals for specific RDF query languages, such as RDQL, RQL and SeRQL.

In 2004, the W3C launched the Data Access Working Group for designing an RDF query language, called SPARQL, from these early attempts [65]. SPARQL 1.0 [19] became an official W3C Recommendation in 2008, and SPARQL 1.1 [11] in 2013.

In this section we define the syntax and semantics of the SPARQL query language. SPARQL 1.1 extends SPARQL 1.0 by adding features to the query language such as aggregates, subqueries, negation, property paths, and an expanded set of functions and operators. Any of these extensions, if described in this section, will be mentioned explicitly as belonging to SPARQL 1.1. Otherwise, the syntax and semantics are common between the two versions.

For a complete description of SPARQL, the reader is referred to the SPARQL 1.1 specification document [11] or to [16, 18] for its formal semantics.

2.2.1 SPARQL Syntax

In this section we define a syntax for SPARQL. For some definitions we adopt an abstract syntax that helps us to define the semantics formally in a clear way.

First we define query variable. A SPARQL query may contain variables that will be bind to values to give a solution for the query.

Definition 2.2.1 (Query Variable[2])**.** *A query variable is a member of an infinite set that is disjoint from the set of RDF terms.*

For the purpose of this document, we are going to deal with variables and blank nodes interchangeably. Semantic results of interpretation and entailment is the same in both cases, so we use the set $\mathcal{B}$ defined previously (in section 2.1) as the set of blank nodes to also mean the set of variables in the case of SPARQL. The specificity of blanks with regard to variables is their quantification. A blank in RDF is a variable existentially quantified over a particular graph.

Definition 2.2.2 (Triple Pattern)**.** *A triple pattern is member of the set* $(\mathcal{T}) \times (\mathcal{I} \cup \mathcal{B}) \times (\mathcal{T})$.

Notice that the definition of *Triple pattern* is equivalent to the definition of *GRDF triple* (the generalized version of an RDF triple) defined previously in section 2.1.

Definition 2.2.3 (Basic Graph Pattern)**.** *A Basic Graph Pattern is a set of Triple Patterns.*

Basic graph patterns are consequently equivalent to GRDF graphs (defined in section 2.1).

SPARQL is based around graph pattern matching. Complex graph patterns can be formed by combining smaller patterns in various ways. We define the following graph patterns used in SPARQL:

- **Basic Graph Patterns**, where a set of triple patterns must match. (It combines triple patterns by conjunction)
- **Group Graph Pattern**, where a set of graph patterns must all match. (It combines graph patterns by conjunction)
- **Optional Graph patterns**, where additional patterns may extend the solution.
- **Alternative Graph Pattern**, where two or more possible patterns are tried. It provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found.

Constraints in SPARQL are boolean-valued expressions that limit the number of answers to be returned. They can be defined on a graph pattern using the keyword `FILTER`. Graph pattern matching produces a solution sequence, where each solution has a set of bindings of variables to RDF terms. SPARQL FILTERs restrict solutions to those for which the filter expression evaluates to TRUE.

[2]In SPARQL, a query variable is marked by the use of either "?" or "$"; the "?" or "$" is not part of the variable name. In a query, $abc and ?abc identify the same variable.

Definition 2.2.4 (SPARQL Graph Pattern). *A SPARQL graph pattern is defined inductively in the following way:*
– *every basic graph pattern (or GRDF graph) is a SPARQL graph pattern.*
– *if P and P' are SPARQL graph patterns and K is a SPARQL constraint, then $(P\ AND\ P')$, $(P\ UNION\ P')$, $(P\ OPT\ P')$, and $(P\ FILTER\ K)$ are SPARQL graph patterns.*

SPARQL Query

SPARQL has four query forms. These query forms use the solutions from pattern matching to form result sets or RDF graphs. The query forms are:

- **SELECT:** Returns all, or a subset of, the variables bound in a query pattern match.
- **CONSTRUCT:** Returns an RDF graph constructed by substituting variables in a set of triple templates.
- **ASK:** Returns a boolean indicating whether a query pattern matches or not.
- **DESCRIBE:** Returns an RDF graph that describes the resources found.

Definition 2.2.5 (SPARQL query). *Given a SPARQL graph pattern P, a sequence $\vec{B}$ of variables in P , an IRI μ, and a basic graph pattern Q,*
– `ASK FROM` μ `WHERE` P
– `SELECT` $\vec{B}$ `FROM` μ `WHERE` P
– `CONSTRUCT` Q `FROM` μ `WHERE` P
– `DESCRIBE` $\vec{B}$ `FROM` μ `WHERE` P
are SPARQL queries.

2.2.2 SPARQL Semantics

In the following, we characterize query answering with SPARQL as done in [16]. The approach relies upon the correspondence between GRDF entailment and maps from RDF graph of the query graph patterns to the RDF knowledge base. SPARQL query constructs are defined through algebraic operations on maps: assignments from a set of variables to terms that preserve names.

Definition 2.2.6 (Map). *Let $V_1 \subseteq \mathcal{T}$, and $V_2 \subseteq \mathcal{T}$ be two sets of terms. A map from V_1 to V_2 is a function $\sigma : V_1 \rightarrow V_2$ such that $\forall x \in (V_1 \cap \mathcal{V}), \sigma(x) = x$.*

Definition 2.2.7 (Application of a map to a basic graph pattern). *The application $\sigma(P)$ of a map σ to a basic graph pattern P , is defined by:*
– $\sigma(P) = \{\sigma(t); t \in P\}$ *if P is a GRDF graph;*
– $\sigma(P) = \langle\sigma'(s), \sigma'(p), \sigma'(o)\rangle$ *if P is a triple $\langle s, p, o\rangle$;*
– $\sigma'(x) = \sigma(x)$ *if $x \in dom(\sigma)$;*
– $\sigma'(x) = x$ *otherwise.*

Operations on maps: If σ is a map, then the domain of σ, denoted by $dom(\sigma)$, is the subset of $\mathcal{T}$ on which σ is defined. The restriction of σ to a set of terms X is defined by $\sigma|_Y^X = \sigma \cup \{\langle x, \texttt{null}\rangle \mid x \in X and x \notin dom(\sigma)\}$[3].

If P is a graph pattern, then $\mathcal{B}(P)$ is the set of variables occurring in P and $\sigma(P)$ is the graph pattern obtained by the substitution of $\sigma(b)$ to each variable $b \in \mathcal{B}(P)$. Two maps σ_1 and σ_2 are compatible when $\forall x \in dom(\sigma_1) \cap dom(\sigma_2), \sigma_1(x) = \sigma_2(x)$. Otherwise, they are said to be incompatible and this is denoted by $\sigma_1 \bot \sigma_2$. If σ_1 and σ_2 are two compatible maps, then we denote by $\sigma = \sigma_1 \oplus \sigma_2 : T_1 \cup T_2 \to \mathcal{T}$ the map defined by $\forall x \in T_1, \sigma(x) = \sigma_1(x)$ and $\forall x \in T_2, \sigma(x) = \sigma_2(x)$.

In the following, we use an alternate characterization of SPARQL query answering that relies upon the correspondence between GRDF entailment and maps from the query graph patterns to the RDF graph [16]. The answers to a basic graph pattern query are those maps which warrant the entailment of the graph pattern by the queried graph. In the case of SPARQL, this entailment relation is GRDF entailment.

Definition 2.2.8 (Compound Graph Pattern Entailment). *Let $\models$ be an entailment relation on basic graph patterns, P and P' be SPARQL graph patterns, K be a SPARQL constraint, and G be an RDF graph. Graph pattern entailment by an RDF graph modulo a map σ is defined inductively by:*

- $G \models \sigma(P \texttt{ AND } P')$ *iff* $G \models \sigma(P)$ *and* $G \models \sigma(P')$
- $G \models \sigma(P \texttt{ UNION } P')$ *iff* $G \models \sigma(P)$ *or* $G \models \sigma(P')$
- $G \models \sigma(P \texttt{ OPT } P')$ *iff* $G \models \sigma(P)$ *and* $[G \models \sigma(P')$ *or* $\forall \sigma' : G \models \sigma'(P), \sigma \bot \sigma']$
- $G \models \sigma(P \texttt{ FILTER } K)$ *iff* $G \models \sigma(P)$ *and* $\sigma(K) = \top$

A SPARQL constraint K is a boolean expression involving terms from $(\mathcal{V} \cup \mathcal{B})$, e.g., a numeric test. Hence, $\sigma(K) = \top$ means that this function is evaluated to true once the variables in K are substituted by σ If not all variables of K are bound, then $\sigma(K) \neq \top$.

As usual for this kind of query language, an answer to a query is an assignment of variables appearing in $\vec{B}$ (those variables in the `SELECT` part of the query). Such an assignment is a map from variables in the query to nodes of the graph. The defined answers may assign only one part of the variables, those sufficient to prove entailment. The answers are these assignments extended to all variables of $\vec{B}$.

Definition 2.2.9 (Answer to a SELECT SPARQL query). *Let `SELECT` $\vec{B}$ `FROM` μ `WHERE` P be a SPARQL query with P a SPARQL graph pattern and G be the (G)RDF graph identified by the IRI μ, then the set of answers to this query is:*
$\mathcal{A}(\vec{B}, G, P) = \{\sigma|_{\vec{B}}^{\vec{B}} \mid G \models_{GRDF} \sigma(P)\}$

2.2.3 Algebraic Manipulation

Answering SPARQL queries may be obtained by directly manipulating graphs and maps. The original semantics of SPARQL was given in this way. In this section we are going to define these manipulations.

The *join* and *difference* of two sets of maps Ω_1 and Ω_2 are defined as follows [16]:

- *(join)* $\Omega_1 \bowtie \Omega_2 = \{\sigma_1 \oplus \sigma_2 \mid \sigma_1 \in \Omega_1, \sigma_2 \in \Omega_2$ *are compatible*$\}$

[3]The `null` symbol is used for denoting the NULL values introduced by the OPTIONAL clause.

– *(difference)* $\Omega_1 \setminus \Omega_2 = \{\sigma_1 \in \Omega_1 \mid \forall \sigma_2 \in \Omega_2, \sigma_1$ *and* σ_2 *are not compatible*$\}$

The answers to a basic graph pattern query are those maps which warrant the entailment of the graph pattern by the queried graph. In the case of SPARQL, this entailment relation is RDF-entailment. Answers to compound graph patterns are obtained through the operations on maps.

Definition 2.2.10 (Answers to compound graph patterns). *Let $\models_{RDF}$ be the RDF entailment relation on basic graph patterns, P and P' be SPARQL graph patterns, K be a SPARQL constraint, and G be an RDF graph. The set $\mathcal{S}(P, G)$ of answers to P in G is the set of maps from $\mathcal{B}(P)$ to $\mathcal{T}(G)$ defined inductively in the following way:*

– $\mathcal{S}(P, G) = \{\sigma|_{\mathcal{B}(P)} \mid G \models_{RDF} \sigma(P)\}$ *if P is a basic graph pattern*
– $\mathcal{S}(P\ \texttt{AND}\ P', G) = \mathcal{S}(P, G) \bowtie \mathcal{S}(P', G)$
– $\mathcal{S}((P\ \texttt{UNION}\ P'), G) = \mathcal{S}(P, G) \cup \mathcal{S}(P', G)$
– $\mathcal{S}((P\ \texttt{OPT}\ P'), G) = (\mathcal{S}(P, G) \bowtie \mathcal{S}(P', G)) \cup (\mathcal{S}(P, G) \setminus \mathcal{S}(P', G))$
– $\mathcal{S}(P\ \texttt{FILTER}\ K, G) = \{\sigma \in \mathcal{S}(P, G) \mid \sigma(K) = \top\}$

Corollary. *Let* `SELECT` $\vec{B}$ `FROM` μ `WHERE` P *be a SPARQL query with P a basic graph pattern, G be the RDF graph identified by the IRI μ, and $\mathcal{S}(P, G)$ be the set of answers to P in G, then*
$\mathcal{A}(\vec{B}, G, P) = \{\sigma|_{\vec{B}}^{\vec{B}} \mid \sigma \in \mathcal{S}(P, G)\}$

Proof. By the interpolation lemma, if there exists an homomorphism from P to G, then $G \models_{GRDF} P$, moreover, this homomorphism determines an instance of G, hence $G \models_{GRDF} \pi(P)$. Hence, $\mathcal{S}(P, G) = \{\sigma|_{\mathcal{B}(P)} \mid G \models_{GRDF} \sigma(P)\}$, so $\{\sigma|_{\vec{B}}^{\vec{B}} \mid \sigma \in \mathcal{S}(P, G)\} = \{\sigma|_{\vec{B}}^{\vec{B}} \mid G \models_{GRDF} \sigma(P)\}$ which is exactly $\mathcal{A}(\vec{B}, G, P)$. □

This corollary can be extended to any graph patterns by induction on the structure of graph patterns.

Proposition 2.2.1 (Answers to a SPARQL query [3]). *Let* `SELECT` $\vec{B}$ `FROM` μ `WHERE` *P be a SPARQL query, G be the RDF graph identified by the IRI μ, and $\mathcal{S}(P, G)$ be the set of answers to P in G, then the answers $\mathcal{A}(\vec{B}, G, P)$ to the query are the restriction and completion to $\vec{B}$ of answers to P in G, i.e.:*
$\mathcal{A}(\vec{B}, G, P) = \{\sigma|_{\vec{B}}^{\vec{B}} \mid \sigma \in \mathcal{S}(P, G)\}$

This shows the equivalence between the semantic definition (Definition 2.2.8) and the algebraic definition (Definition 2.2.10).

2.3 SPARQL with Property Paths

Triple Patterns are written as subject, predicate and object constructs. It can be abstractly represented as $\langle s, p, o\rangle$, where s is a subject, p is a predicate, and o is an object. In a SPARQL query, s and o may be any RDF term or a variable, while the predicate is an IRI (or variable) that defines a relation between the two entities (the subject and the object).

Many studies have been carried out in order to extend the expressiveness of the language. Probably one of the important studies carried out concerns extending the way a relation between two entities can be expressed. Introduction of regular expression at the predicate position of the triple patterns is responsible for providing such an increase in expressiveness. Works like [2] which proposes a complete syntax and semantics of PSPARQL (an extension of SPARQL), and [17] which proposes the navigational language called nSPARQL are examples of works whose basic idea was the introduction of regular expressions at the predicate position. In SPARQL 1.1 this concept was also introduced.

2.3.1 Property Paths in SPARQL 1.1

In SPARQL 1.1, a property path is a possible route through a graph between two graph nodes. A trivial case is a property path of length exactly 1, which is a triple pattern. The ends of the path may be RDF terms or variables. Variables cannot be used as part of the path itself, only the ends.

Property paths allow for more concise expressions for some SPARQL basic graph patterns and they also add the ability to match connectivity of two resources by an arbitrary length path.

A property path expression is an expression using the property path forms described in the following table:

In the description below, ***iri*** *is an IRI and* ***elt*** *is a path element, which may itself be composed of path constructs.*

Syntax Form	**Matches**
iri	An IRI. A path of length one.
$\hat{}elt$	Inverse path (object to subject).
elt_1/elt_2	A sequence path of elt1 followed by elt2.
$elt_1\|elt_2$	A alternative path of elt1 or elt2 (all possibilities are tried).
$elt*$	A path that connects the subject and object of the path by zero or more matches of elt.
$elt+$	A path that connects the subject and object of the path by one or more matches of elt.
$elt?$	A path that connects the subject and object of the path by zero or one matches of elt.
$!iri$ or $!(iri_1\|...\|iri_n)$	Negated property set. An IRI which is not one of irii. !iri is short for !(iri).
$!\hat{}iri$ or $!(\hat{}iri_1\|...\|\hat{}iri_n)$	Negated property set where the excluded matches are based on reversed path. That is, not one of iri1...irin as reverse paths. !^iri is short for !(^iri).
$!(iri_1\|...\|iri_j\|\hat{}iri_{j+1}\|...\|\hat{}iri_n)$	A combination of forward and reverse properties in a negated property set.
(elt)	A group path elt, brackets control precedence.

Definition 2.3.1 (Property Path Pattern)**.** Let PP be the set of all property path

expressions. A property path pattern is a member of the set:
$\mathcal{T} \times PP \times \mathcal{T}$

A Property Path Pattern is a generalization of a Triple Pattern to include a property path expression in the property position. A property path expression returns matches based on nodes connected by the path.

2.4 Conclusion

In this chapter we studied and reported on the Semantic Web technologies: RDF and SPARQL.

The Resource Description Framework (RDF) is a framework for expressing information about resources in the World Wide Web. RDF can be expressed in a variety of formats. We have presented an abstract syntax for RDF, then we provided a detailed study about its semantics, and an inference mechanism based on the abstract syntax.

We also reported on SPARQL - a query language for accessing RDF published data. We have presented the syntax and the different fragments of this query language. We then provided its semantics, and an algebraic manipulation for it. We also introduced one of its recent extensions, concerning property paths, which has been proposed by different studies and was recently introduced in the latest version, SPARQL 1.1.

The presentation and definitions of these technologies are the necessary building blocks for our work in the rest of this document.

Chapter 3

SPARQL Query Rewriting

3.1 Introduction to Eliminating Non-Distinguished Variables

Our purpose in this work is to study and propose a methodology for eliminating non-distinguished variables from a SPARQL query. In our study, we only deal with SPARQL queries with projection, as other queries do not have non-distinguished variables.

For instance, consider a SELECT SPARQL query of the form:

`SELECT` $\vec{B}$ `FROM` μ `WHERE` P

The `SELECT` clause is equivalent to projection in relational algebra. It suppresses the result set to the variables of $\vec{B}$. The graph pattern P may also explicitly contain variables that are not included in $\vec{B}$, such variables are called non-distinguished variables, and there values' assignment is not a part of the result set. Such variables may be useful for expressing certain intended situations that are otherwise cannot be expressed using only the distinguished variables (variables in $\vec{B}$) and triple patterns.

As said earlier, introduction of property path expressions into the language increases its expressive power. The main effort in our study is to exploit this additional power in order to eliminate the need for non-distinguished variables in the graph pattern of the query. Basically, our work is to find for a given query with non-distinguished variables, an equivalent query that does not introduce such variables.

So why it is interesting to eliminate non-distinguished variables variables?

3.1.1 The Purpose of Eliminating Non-Distinguished Variables

The projection operator is a source of extra complexity in query evaluation: NP for queries composed of basic graph patterns [16, 20]. If this operator can be suppressed, the complexity reduces to just PTime [20].

In general, the more there are variables in a query graph pattern, means more computation time will be needed for the query to be executed. The query computation process is based on query graph pattern matching against the data graph. The number of matchings done dramatically increases (exponentially) with the number of variables in the query graph pattern.

So considering the possibility of rewriting SPARQL queries with projection into SPARQL queries without projection is interesting because it would guarantee a lower complexity. Our purpose is to limit the presence of variables in the query graph pattern by keeping the distinguished variables only, and eliminating any other variable.

3.1.2 Challenges Overview

Although property paths increase the expressive power of the language, but there is no clue that it can be a complete alternative for using non-distinguished variables to express some situation in the graph pattern. Yes what is really known is that the mixture of both techniques (non-distinguished variables and property paths), gives users the handy tools to frame their query. This is the great challenge that we would face: Is it possible to translate every query with non-distinguished variables to a query without non-distinguished variables while exactly preserving the meaning?

In order to study this problem, we will be in the flow of this document defining the situations in which such a transformation is completely possible, and also studying other situations and the possible solutions that can be applied in order to keep the computation time as low as possible.

Even though some fragments of SPARQL are completely transformable into queries without non-distinguished variables, for many of these queries it is more intuitive, readable/writable, and understandable for humans to use non-distinguished variables rather than using its equivalent that does not include non-distinguished variables (in the example below, *Graph Pattern 1* is more intuitive to write/read than *Graph Pattern 2*). But as eliminating these non-distinguished variables guarantees a lower computation time, here comes the importance of automatic transformation of queries, the main topic of this document.

For example, consider the following two graph patterns:

Graph Pattern 1:

```
?x name "Alice"
?x age 24
```

Graph Pattern 2:

```
"Alice" ^name/age 24
```

Both graph patterns hold exactly the same meaning: *"There exist a resource whose name is "Alice" and whose age is 24"*. In *Graph Pattern 2*, a property path was used (`^name/age`). It was possible to eliminate the need of including the variable `?x` while preserving the meaning. Given this simple example, our purpose in the rest of this document is to generalize patterns and define the rules for the transformation process to exhibit the desired form.

3.1.3 Methodology

The methodology in our study is to define a variable eliminating function for an arbitrary variable (let it be `?x`). This function takes any graph pattern and eliminates `?x` from it where such elimination is possible. Our purpose in the definition of this function is to preserve the meaning of the graph pattern after transformation. Having our variable eliminating function defined, we apply it on a query graph pattern for each of the non-distinguished variables in the query.

For reaching our purpose, we present in the rest of this document a case study for the possible SPARQL fragments, and we deal with each fragment separately. We finally combine our results into one global transformation function.

We found out that not all fragments of SPARQL can be rewritten without non-distinguished variables while preserving the meaning of the query. For this reason, we define constraints on the queried dataset that allows our proposed transformation rules to become sound and complete if such constraints are met.

To deal with non-constrained datasets (any possible dataset), we propose an alternative solution that benefits from our transformation function, but re-introduces the non-distinguished variables in a way that promises a lower computation time than the original query, and only witnesses the same computation time in the worst case.

3.2 Elementary Transformation Rules

We will use "BGP" and "PP" as abbreviations for "Basic Graph Pattern" and "Property Path" respectively in order to make short titles.

In this section, our goal is to define elementary transformation rules for some basic cases, upon which our final transformation function will be collaboratively defined. For now we only consider basic graph patterns, and we found it useful to divide our study into three parts: Basic graph patterns with single property path pattern, with double property path patterns, and with multiple property path patterns.

Our transformation rules are also concerned about eliminating a single variable at a time. For this purpose, only this variable will be represented as variable (with a `?` symbol), while everything else will be represented with an abstract symbol (`S1`, `S2`, ... for subjects, `P1`, `P2`, ... for property path expressions and `O1`, `O2`, ... for objects) which can be anything allowed by SPARQL 1.1 syntax.

We define a function ζ_x to be an $?x$ eliminating function. It takes a graph pattern as an argument and returns a transformed graph pattern without $?x$ if such transformation is possible. This function will be defined according to a set of transformation rules that will be studied on case basis.

In general, given a basic graph pattern, any property path pattern in it that does not contain $?x$ will be copied as it is. This allows us to define our first intuitive transformation rule.

Transformation Rule 1. *Given a basic graph pattern P not containing $?x$, we define $\zeta_x(P) = P$.*

3.2.1 Single PP Pattern with a Non-Distinguished Variable

A BGP of the form:

```
{?x P1 O1}
```

will be rewritten as:

```
{O1 ^P1/P1 O1}
```

Thus eliminating the variable ?x, and preserving the meaning of the BGP.

In the previous transformation example, the variable we want to eliminate ($?x$) was in the subject position. In general, the variable may instead appear in the object position. In following sections, it will even become more complex when dealing with more than one property path pattern where the variable may appear in a mixture of subject and object positions, and tremendously increase the number of cases we are dealing with. For simplicity of our transformation process, we define our elementary transformation rules only for the case where variables appear in the subject position. In a latter section, we show that we can always adjust the variable position to become in the subject position, and thus allowing our elementary transformation rules to be applied globally (for all cases).

To describe the previous transformation procedure (and other following transformation procedures), we define a supplementary transformation function ζ' which will be the core of our variable eliminating function (ζ_x).

Supplementary Transformation Rule 1. *Given a basic graph pattern P of a single property path pattern $\{\langle s,p,o\rangle\}$, we define $\zeta'(P)$:*

$$\frac{P = \{\langle s,p,o\rangle\}}{\zeta'(P) = \{\langle o,\ \hat{}p/p,\ o\rangle\}}$$

Our supplementary transformation function (ζ') is a direct application of a set of transformation rules which do not take into account the complete composition of its input. The semantics of our transformation is given by our propositions, and these semantics are taken into consideration in the final definition of our variable eliminating function (ζ_x).

Proposition 3.2.1. *Given a GRDF graph G, a variable sequence $\vec{B}$, and a basic graph pattern P. Let P_1 be any property path, and O_1 be any RDF term. If P consists of a single property path pattern of the form $\{\langle ?x,P_1,O_1\rangle\}$ and $?x \notin \vec{B}$, then $\mathcal{A}(\vec{B},G,P) = \mathcal{A}(\vec{B},G,\zeta'(P))$.*

Proof. By definition of the property path sequence operator, $\{\langle s,p/q,o\rangle\}$ is equivalent to $\{\langle s,p,?x\rangle,\langle ?x,q,o\rangle\}$ for an arbitrary variable $?x$ [11], and by definition of the property path inverse operator, $\{\langle s,\hat{}p,o\rangle\}$ is equivalent to $\{\langle o,p,s\rangle\}$ [11]. The basic graph pattern we consider $\{\langle ?x,P_1,O_1\rangle\}$ can be rewritten as $\{\langle ?x,P_1,O_1\rangle,\langle ?x,P_1,O_1\rangle\}$, i.e. by repeating the same property path pattern, then rewritten as $\{\langle O_1,\hat{}P_1,?x\rangle,\langle ?x,P_1,O_1\rangle\}$, then rewritten as $\{\langle O_1,\hat{}P_1/P_1,O_1\rangle\}$ which is our transformed form, and thus showing that our transformation preserves the meaning of the graph pattern. Since $?x$ is not a part of the result set (given by the condition $?x \notin \vec{B}$), then $\mathcal{A}(\vec{B},G,\{\langle ?x,P_1,O_1\rangle\}) = \mathcal{A}(\vec{B},G,\zeta'(\{\langle ?x,P_1,O_1\rangle\}))$. □

3.2.2 Double PP Patterns Sharing a Non-Distinguished Variable

A BGP of the form:

```
{
  ?x P1 O1.
  ?x P2 O2
}
```

will be rewritten as:

```
{
 O1 ^P1/P2 O2
}
```

Supplementary Transformation Rule 2. *Given a basic graph pattern P of two property path patterns $\{\langle s_1, p_1, o_1\rangle, \langle s_2, p_2, o_2\rangle\}$, we define $\zeta'(P)$.*

$$\frac{P = \{\langle s_1, p_1, o_1\rangle, \langle s_2, p_2, o_2\rangle\}}{\zeta'(P) = \{\langle o_1,\ \hat{}p_1/p_2,\ o_2\rangle\}}$$

Proposition 3.2.2. *Given a GRDF graph G, a variable sequence $\vec{B}$, and a basic graph pattern P. Let P_1 and P_2 be any property paths, and O_1 and $O2$ be any RDF terms. If P consists of two property path patterns of the form $\{\langle ?x, P_1, O_1\rangle, \langle ?x, P_2, O_2\rangle\}$ and $?x \notin \vec{B}$, then $\mathcal{A}(\vec{B}, G, P) = \mathcal{A}(\vec{B}, G, \zeta'(P))$.*

Proof. By definition of the property path sequence operator, $\{\langle s, p/q, o\rangle\}$ is equivalent to $\{\langle s, p, ?x\rangle, \langle ?x, q, o\rangle\}$ for an arbitrary variable $?x$ [11], and by definition of the property path inverse operator, $\{\langle s, \hat{}p, o\rangle\}$ is equivalent to $\{\langle o, p, s\rangle\}$ [11]. The basic graph pattern we consider $\{\langle ?x, P_1, O_1\rangle, \langle ?x, P_2, O_2\rangle\}$ can then be rewritten as $\{\langle O_1, \hat{}P_1, ?x\rangle, \langle ?x, P_2, O_2\rangle\}$, then rewritten as $\{\langle O_1, \hat{}P_1/P_2, O_2\rangle\}$ which is our transformed form, and thus showing that our transformation preserves the meaning of the graph pattern. Since $?x$ is not a part of the result set (given by the condition $?x \notin \vec{B}$), then $\mathcal{A}(\vec{B}, G, P) = \mathcal{A}(\vec{B}, G, \zeta'(P))$ if $P = \{\langle ?x, P_1, O_1\rangle, \langle ?x, P_2, O_2\rangle\}$. $\square$

3.2.3 Multiple PP Patterns Sharing a Non-Distinguished Variable

A basic graph pattern with multiple property path patterns here means that it consists of 3 or more property path patterns sharing the same non-distinguished variable.

Consider the following basic graph pattern with 3 property path patterns sharing $?x$:

```
{
  ?x P1 O1.
  ?x P2 O2.
  ?x P3 O3
}
```

We can define another basic graph pattern as follows:

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3 O3.
  O3 ^P3/P1 O1
}
```

Our previous transformation proposal does not exactly conserve the semantics of the original basic graph pattern. We found out that it is not possible to eliminate $?x$ from multiple property path patterns containing $?x$ without loosing semantics. For this moment, we adopt this transformation for multiple property path patterns. In a latter section, we will introduce constraints on the queried graph which will make our transformation sound and complete.

The previous transformation is based on linking each property path pattern with the one following it in a cycle, ending by linking the last property path pattern to the first. This can be generalized to more than 3 property path patterns in the same way.

Supplementary Transformation Rule 3. *Given a basic graph pattern P of n property path patterns $\{\langle s_1, p_1, o_1\rangle, ..., \langle s_n, p_n, o_n\rangle\}$ where $n \in \mathbb{N} : n \geq 3$, we define $\zeta'(P)$.*

$$\frac{P = \{\langle s_1, p_1, o_1\rangle, ..., \langle s_n, p_n, o_n\rangle\}}{\zeta'(P) = \left\{\langle o_i,\ \hat{}p_i/p_{(i \bmod n)+1},\ o_{(i \bmod n)+1}\rangle \mid i \in \{1, \dots, n\}\right\}}$$

In a latter section about constrained datasets, we will give the semantics of the previous transformation with respect to these constraints.

Now we can give the complete definition to our *supplementary transformation function (ζ')* which transforms a given basic graph pattern according to the three different cases discussed previously.

Definition 3.2.1 (Supplementary transformation function (ζ')). *Given a basic graph pattern P, and number $n \in \mathbb{N} : n \geq 3$, we define the supplementary transformation function $\zeta'(P)$:*

- $\frac{P=\{\langle s,p,o\rangle\}}{\zeta'(P)=\{\langle o,\ \hat{}p/p,\ o\rangle\}}$,
- $\frac{P=\{\langle s_1,p_1,o_1\rangle,\langle s_2,p_2,o_2\rangle\}}{\zeta'(P)=\{\langle o_1,\ \hat{}p_1/p_2,\ o_2\rangle\}}$,
- $\frac{P=\{\langle s_1,p_1,o_1\rangle,...,\langle s_n,p_n,o_n\rangle\}}{\zeta'(P)=\left\{\langle o_i,\ \hat{}p_i/p_{(i \bmod n)+1},\ o_{(i \bmod n)+1}\rangle \mid i\in\{1,...,n\}\right\}}$

As said earlier, the *supplementary transformation function (ζ')* is a direct application of a set of transformation rules which do not take into account the complete composition of its input. Yet it is the core for defining our *variable eliminating function (ζ_x)*, taking into account the complete composition.

In the next section, we perform a case study in order to provide a complete definition for the *variable eliminating function (ζ_x)*, by using the *supplementary transformation function (ζ')* as necessary.

3.3 Query Transformation

Our purpose in this section is to first provide a complete definition for our *variable eliminating function (ζ_x)*. We carry out a case study for this purpose, detailing the different cases the function will possibly encounter. The *variable eliminating function (ζ_x)* takes a graph pattern as input, and eliminates the variable $?x$ where this elimination is possible, by utilizing our *supplementary transformation function (ζ')* defined previously.

We then define a *query transformation function (ζ)*. This function takes a query as input, and applies the *variable eliminating function (ζ_x)* on its graph pattern, for each of the non-distinguished variables in it.

3.3.1 Case Study

Presence of Property Path Patterns without the Non-Distinguished Variable

The *supplementary transformation function (ζ')*, takes a basic graph pattern, and semantically assumes the presence of the variable in all its property path patterns, and thus our transformation proposal was based on eliminating such variable.

To further generalize our transformation process, let us consider the following basic graph pattern example:

```
{
  ?x P1 O1.
  S2 P2 O2
}
```

where `S2` and `P2` are any RDF terms but not `?x`. In such case we can apply our *supplementary transformation function (ζ')* on the first property path pattern, while copy the second property path pattern as it is.

Thus, it is desirable to make separation between property path patterns that contain `?x` in a basic graph pattern, and those that do not contain `?x`. This allows us to define the transformation rules separately for each part. For this purpose we define a *basic graph pattern partitioning function (α_x)*. This function takes a basic graph pattern as input, and returns a basic graph pattern consisting of only the property path patterns that contain `?x`. The set of other property path patterns are given by another function $\bar{\alpha}_x$.

Definition 3.3.1 (Basic graph patterns partitioning functions α_x and $\bar{\alpha}_x$)**.** *Given a basic graph pattern P and a variable $?x$, we define the partitioning functions α_x and $\bar{\alpha}_x$:*
$\alpha_x(P) = \{\langle s,p,o\rangle \mid \langle s,p,o\rangle \in P,\ s \text{ is } ?x \text{ or } o \text{ is } ?x\}$
$\bar{\alpha}_x(P) = \{\langle s,p,o\rangle \mid \langle s,p,o\rangle \in P,\ s \text{ and } o \text{ are not } ?x\}$

Variable Position

The *supplementary transformation function (ζ')*, takes a basic graph pattern, and semantically assumes the presence of the variable in subject positions of the property path patterns, and thus our transformation proposal was based on eliminating all RDF terms in the subject position.

In general, the variable may be in the object position instead, or a mixture of subject and object positions may occur, and thus tremendously increasing the number of cases we are dealing with.

For simplicity of our transformation process, we defined our *supplementary transformation function (ζ')* only for the case where variables appear in the subject position. In this section we show that we can always adjust the variable position to become in the subject position, and thus allowing our *supplementary transformation function (ζ')* to be applied globally (for all cases).

With the use of the property path inverse operator ($\hat{}P$ instead of P), any variable in one position can be switched to the other.

In a basic graph pattern, for every property path pattern of the form:

```
S1 P1 ?x
```

(where the non-distinguished variable ?x is in the object positon), we replace it with another property path pattern of the form:

```
?x ^P1 S1
```

Having all the non-distinguished variables rearranged to be in the subject position, we can then apply our *supplementary transformation function (ζ')*.

For this purpose, we use a *variable position adjustment function (δ_x)* for a variable $?x$ in a basic graph pattern, defined as follows:

Definition 3.3.2 (Variable position adjustment function (δ_x))**.** *Given a basic graph pattern P and a variable $?x$, we define $\delta_x(P)$:*
- $\delta_x(P) = \{\delta_x(t) \mid t \textit{ is a triple in } P\}$
- $\delta_x(\langle s, p, o\rangle) = \langle o, \hat{}p, s\rangle$ (if o is $?x$ and s is not $?x$)
- $\delta_x(\langle s, p, o\rangle) = \langle s, p, o\rangle$ (otherwise)

Reflexive Property Path Patterns

We say that a property path pattern is reflexive if its subject and object nodes are the same element. We also say that a property path pattern is reflexive on $?x$ if this element is $?x$.

Definition 3.3.3 (Reflexive Property Path Pattern on a Variable)**.** *Given a variable $?x$, a property path pattern $\langle s, p, o\rangle$ is called reflexive on $?x$ if s and o are $?x$.*

The existence of a reflexive property path pattern on a variable in a basic graph pattern has an adverse effect on the transformation process, i.e. we do not get the expected result of eliminating this variable.

For example consider the following basic graph pattern of a single and reflexive property path pattern on $?x$:

```
{ ?x P1 ?x }
```

Using our *supplementary transformation function (ζ')* will give us the following transformation:

```
{ ?x ^P1/P1 ?x }
```

This transformation does not violate any of our previous propositions (particularly it does not violate *proposition 3.2.1*). The real problem in the previous example is that the variable $?x$ is not eliminated by the application of the *supplementary transformation function (ζ')*, mainly because we are dealing with a reflexive property path pattern.

To generalize, in a basic graph pattern, if all the property path patterns containing $?x$ are reflexive, then the elimination of this variable is not possible by applying the transformation rules, and thus we keep such variable.

On the contrary, the presence of at least one non-reflexive property path pattern on a variable $?x$ in a basic graph pattern is sufficient to solve the previous problem. In such case, although the application of the *supplementary transformation function (ζ')* may keep a residual of the variable $?x$ in the transformed basic graph pattern, the application of the *supplementary transformation function (ζ')* recursively will eventually lead to the complete elimination of $?x$.

Example 3.3.1. Consider the following basic graph pattern:

```
{
  ?x P1 ?x.
  ?x P2 O2.
  ?x P3 O3
}
```

We apply *supplementary transformation rule 3* that deals with multiple property path patterns in a basic graph pattern. After the first transformation we get the following basic graph pattern:

```
{
  ?x ^P1/P2 O2.
  O2 ^P2/P3 O3.
  O3 ^P3/P1 ?x.
}
```

We adjust the variable position (using δ_x):

```
{
  ?x ^P1/P2 O2.
  O2 ^P2/P3 O3.
  ?x ^(^P3/P1) O3
}
```

Considering only the two property path patterns containing $?x$ (using α_x), we apply *supplementary transformation rule 2* (other property path patterns will be copied as they are):

```
{
  O2 ^(^P1/P2)/^(^P3/P1) O3.
  O2 ^P2/P3 O3
}
```

Such multiple transformations can be defined recursively by our *variable eliminating function* ζ_x according to the following transformation rule. *We note out, in consideration of the following transformation rule, that the definition of a basic graph pattern as a set of property path patterns allows us to perform set operations on it such as the set union operator ($\cup$).*

Transformation Rule 2. *Given a basic graph pattern P, and a variable $?x$, we define:*
- $\zeta_x(\emptyset) = \emptyset$ ($\emptyset$, i.e. the empty set)
- $\zeta_x(P) = P$ (if all PP patterns containing $?x$ are reflexive)
- $\zeta_x(P) = \zeta_x(\zeta'(\delta_x(\alpha_x(P)))) \cup \bar{\alpha}_x(P)$ (otherwise)

In our previous definition, the recursive step implements nested calls of functions $\zeta_x(\zeta'(\delta_x(\alpha_x(P))))$ that can be described in steps as follows:

- We get the subset of property path patterns that contain $?x$ using the partitioning function α_x.
- We adjust the variable positions to the subject position using δ_x.
- We apply the supplementary transformation rules defined by ζ'.
- We call our variable eliminating function ζ_x again to deal with residuals of the variable $?x$ if they exist from the previous transformation.

The role of the first and the second step is to produce a basic graph pattern that is semantically compatible with the *supplementary transformation function* ζ'. Notice that in the definition of ζ' we only considered property path patterns in an abstract form $\langle s, p, o\rangle$. On the other hand, the transformation semantics are given by the propositions following each supplementary transformation rule. The first two steps here take care about preparing the conditions expressed in these propositions.

Non-Basic Graph Patterns

Everything discussed earlier applies on a basic graph pattern query level. There is a wider range of query fragments, including the UNION, AND, and OPT fragments which allow combining basic graph patterns in various ways in order to form the query graph pattern.

These new fragments do not really matter for our purpose. From a semantic point of view, SPARQL graph pattern matching is defined in terms of combining the results from matching basic graph patterns appearing in the query. Our transformation strategy is as so applied per basic graph pattern, and operations like UNION, AND, and OPT are conserved in position. This is applicable because in our transformation proposal, every basic graph pattern transformation can only result in a basic graph pattern.

Transformation Rule 3. *Given two graph patterns P_1 and P_2, we define:*

- $\zeta_x(P_1 \texttt{ UNION } P_2) = \zeta_x(P_1) \texttt{ UNION } \zeta_x(P_2)$
- $\zeta_x(P_1 \texttt{ AND } P_2) = \zeta_x(P_1) \texttt{ AND } \zeta_x(P_2)$
- $\zeta_x(P_1 \texttt{ OPT } P_2) = \zeta_x(P_1) \texttt{ OPT } \zeta_x(P_2)$

FILTER Constraint on a Non-Distinguished Variable

A graph pattern can use a FILTER operator in order to add constraints to the query. A FILTER operator may reference a non-distinguished variable to add constraints on this variable. Once the non-distinguished variable is eliminated, the reference to this variable becomes impossible, and thus we loose the ability to maintain this constraint in our query. A non-distinguished variable that appears in a filter constraint will not be eliminated from its corresponding graph pattern. This allows value assignments to this variable to occur in order maintain the semantics of the constraint.

Transformation Rule 4. *Given a graph pattern P and a filter constraint K, we define:*

$$\zeta_x(P \texttt{ FILTER } K) = \begin{cases} P \texttt{ FILTER } K & \text{(if } K \text{ contains } ?x\text{)} \\ \zeta_x(P) \texttt{ FILTER } K & \text{(otherwise)} \end{cases}$$

3.3.2 Variable Eliminating Function

Based on the discussions and results obtained earlier in this document, we can now provide a complete definition for the *variable eliminating function (ζ_x)*. This function takes a graph pattern as an input, and it returns a transformed graph pattern by eliminating the variable $?x$ from the original graph pattern where this elimination is possible.

The definition of ζ_x will be given in an inductive way and by reducing to the *supplementary transformation function (ζ')* in some cases. The definition also uses the *variable position adjustment function (δ_x)*, and the *partitioning functions* α_x and $\bar{\alpha}_x$.

The definition is a combination of the transformation rules defined previously. For a variable $?x$, we define the variable eliminating function ζ_x inductively as follows:

Definition 3.3.4 (Variable Eliminating Function (ζ_x))**.** *Let P be a basic graph pattern, P_1 and P_2 two graph patterns, K a filter constraint, we define the variable eliminating function ζ_x inductively:*

- $\zeta_x(\emptyset) = \emptyset$ ($\emptyset$, i.e. the empty set)
- $\zeta_x(P) = P$ (if all PP patterns containing $?x$ in P are reflexive)
- $\zeta_x(P) = \zeta_x(\zeta'(\delta_x(\alpha_x(P)))) \cup \bar{\alpha}_x(P)$ (otherwise)

- $\zeta_x(P_1 \texttt{ UNION } P_2) = \zeta_x(P_1) \texttt{ UNION } \zeta_x(P_2)$
- $\zeta_x(P_1 \texttt{ AND } P_2) = \zeta_x(P_1) \texttt{ AND } \zeta_x(P_2)$
- $\zeta_x(P_1 \texttt{ OPT } P_2) = \zeta_x(P_1) \texttt{ OPT } \zeta_x(P_2)$

- $\zeta_x(P_1 \texttt{ FILTER } K) = P_1 \texttt{ FILTER } K$ (if K contains $?x$)
- $\zeta_x(P_1 \texttt{ FILTER } K) = \zeta_x(P_1) \texttt{ FILTER } K$ (otherwise)

3.3.3 Query Transformation Function

The *variable eliminating function (ζ_x)* is defined abstractly for eliminating any variable from a graph pattern.

Considering a SPARQL query, we are interested in applying this function for each of the non-distinguished variables in the query. The SPARQL query defines a variable sequence $\vec{B}$ whose elements are the set of distinguished variables. Other variables appearing in the query graph pattern are the non-distinguished variables, each of which the *variable eliminating function (ζ_x)* should be applied for.

We denote by ζ our *query transformation function.* It takes a query as an input, and it returns another transformed query by eliminating the non-distinguished variables in its graph pattern where this elimination is possible.

Definition 3.3.5 (Query Transformation Function (ζ)). *Given a query $Q(\vec{B}, P)$, where $\vec{B}$ is a variable name sequence, and P is the query graph pattern. Let $?x_1$, $?x_2$, ..., $?x_n$ be the set of non-distinguished variables in Q. We define the query transformation function ζ:*

$$\zeta(Q(\vec{B}, P)) = Q(\vec{B}, P')$$

where

$$P' = \zeta_{x_1}(\zeta_{x_2}(\dots\zeta_{x_n}(P)))$$

We mention that our *query transformation function (ζ)* is typically defined for SELECT query forms, with a projection defined by the variable sequence ($\vec{B}$) of the `SELECT` clause.

Semantically, the notion of distinguished/non-distinguished variables can be generalized to other query forms. In the CONSTRUCT query form, the variables of the graph template of the `CONSTRUCT` clause can be considered the set of distinguished variables. In the DESCRIBE query form, the variable sequence that may follow the `DESCRIBE` clause can be considered the set of distinguished variables. In the ASK query form, there are no distinguished variables. In all cases, the non-distinguished variables are the other variables appearing in the query graph pattern.

3.3.4 Transformation Complexity

Complexity of Eliminating a Variable

The complexity of eliminating a variable from a basic graph pattern is quadratic $O(n^2)$, where n is the size of the basic graph pattern (number of property path patterns). The main reason for quadratic complexity is the recursion required due to the potential presence of reflexive property path patterns. The recursion may occur $(n-1)$ times in the worst case, and the complexity of each recursive step is $O(n)$, and can be described as follows:

- Application of the *partitioning functions*, α_x and $\bar{\alpha}_x$, has $O(n)$ complexity.
- Application of the *variable position adjustment function (δ_x)* has $O(n)$ complexity.
- Application of the *supplementary transformation function (ζ')* has $O(n)$ complexity, where each transformation results in a new basic graph pattern with the same number of property path patterns as the original basic graph pattern, and transformation can be done sequentially on fly.

UNION, OPTIONAL, AND, and FILTER operators has no additional computational complexity on the transformation process. These operations will be conserved in position in the new query.

Proposition 3.3.1. *Complexity of the application of the variable eliminating function (ζ_x) is $O(n^2)$, where n is the number of property path patterns in the query graph pattern.*

Complexity of Query Transformation

The *query transformation function (ζ)* is an application of the *variable eliminating function (ζ_x)*, which has $O(n^2)$ complexity, for each of the non-distinguished variables in the query. The number of non-distinguished variables is bounded to $2n$, where n is the size of the query (number of property path patterns). Thus, the complexity of query transformation is cubic in the size of query.

Proposition 3.3.2. *Complexity of the application of the query transformation function (ζ) is $O(n^3)$, where n is the number of property path patterns in the query graph pattern.*

3.3.5 Graphical Representation Summary

In the following figure, we illustrate a variable decomposition in a basic graph pattern.

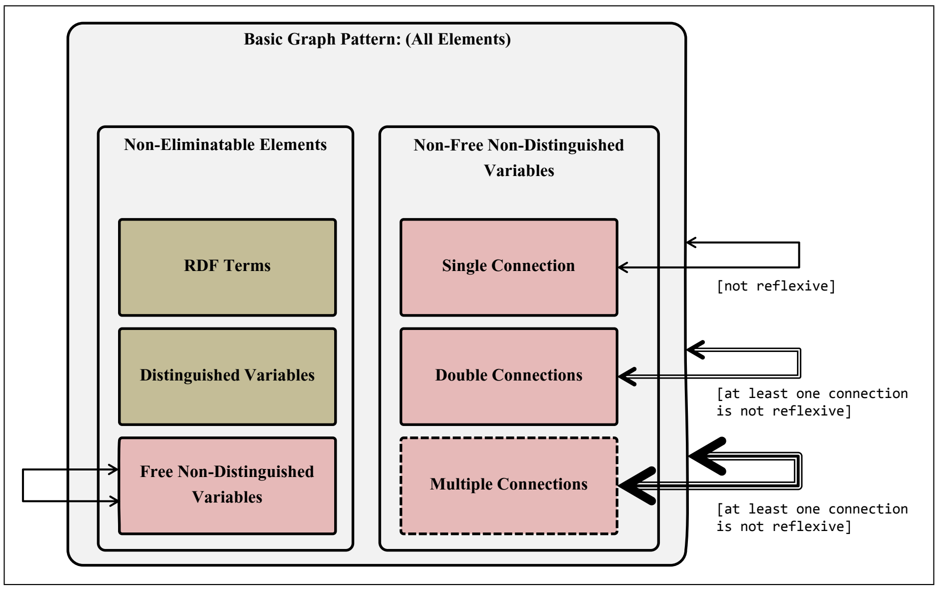
Basic Graph Pattern: (All Elements)
Non-Eliminatable Elements
RDF Terms
Distinguished Variables
Free Non-Distinguished Variables
Non-Free Non-Distinguished Variables
Single Connection
Double Connections
Multiple Connections
[not reflexive]
[at least one connection is not reflexive]
[at least one connection is not reflexive]

Figure 3.1: Elements decomposition in a basic graph pattern

Figure 3.1 allows us to demonstrate the different cases we can deal with for the purpose of transformation of a basic graph pattern.

The elements of a basic graph pattern are mainly divided into two sets: the set of *non-eliminatable elements*, and the set of *non-free non-distinguished variables.*

The set of *non-eliminatable elements* intuitively includes the *RDF terms* and *distinguished variables.*

Non-distinguished variables are divided into four sets:

- **Free variables:** A variable belongs to this set if it has connections only to itself, i.e. all property path pattern containing this variable are reflexive. Such variable cannot be eliminated, and thus it belongs to the bigger set of *non-eliminatable elements*.

- **Non-free single connected variables:** A variable belongs to this set if it has only a single connection to another element in the basic graph pattern. These variables can be eliminated, and we deal with them using *supplementary transformation rule 1*.

- **Non-free double connected variables:** A variable belongs to this set if it has two connections to elements, at least one of which is not itself. These variables can be eliminated, and we deal with them using *supplementary transformation rule 2*.

- **Non-free multiple connected variables:** A variable belongs to this set if it has three or more connections to elements, at least one of which is not itself. Normally, these variables cannot be eliminated while conserving the complete semantics of the basic graph pattern. In the next section we will define constraints on the queried graph that allow us to deal with such variables. If these constraints are met in the queried graph, these variables can then be eliminated, and we deal with them using *supplementary transformation rule 3*.

3.4 Constrained Datasets

Our transformation proposal is defined to be used with constrained data graphs (i.e. queried graphs). The transformation is sound and complete if these constraints are met. In general our transformation is complete but not sound.

Considering an arbitrary data graph, our transformation is not sound because by applying *supplementary transformation rule 3*, which deals with 3 or more property path patterns containing the variable we want to eliminate in a basic graph pattern, we loose some of the semantics of the original basic graph pattern. We found out that it is not possible to suggest a transformation rule that eliminates a variable contained in 3 or more property path patterns without loosing semantics.

In this section, we first carry out a set of transformation suggestions that all fail to completely conserve the semantics, giving the intuition why such transformation is not possible. We then give the semantics of our transformation choice, and show that it is sound and complete when some constraints, that we will define, are met.

3.4.1 Various Transformation Suggestions

Consider the following basic graph pattern with 3 property path patterns sharing $?x$:

```
{
  ?x P1 O1.
  ?x P2 O2.
  ?x P3 O3
}
```

The previous basic graph pattern will match subgraphs in the data graph that have a pattern similar to that of *figure 3.2*.

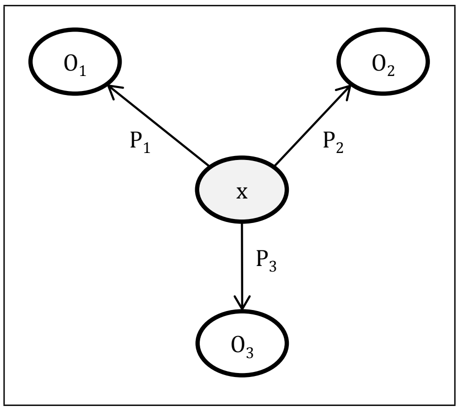

Figure 3.2
where x is any RDF term, and $O1, O2, O3, P1, P2$ and $P3$ are the RDF terms referenced explicitly by the query's graph pattern.

Again our goal is to eliminate ?x. We suggest different transformation proposals, by investigating the expressive power of different SPARQL fragments, and thus showing the flaw of each of such suggestions.

1. **Single Property Path Pattern:** A single property path pattern can only match a path (a sequence of properties) in a data graph, i.e. it can neither match tree nor a graph (unless they are as simple as being considered paths). In analogy, it is the same as a *word regular expression* can only catch strings (a sequence of characters). In our case, as can be seen in *figure 3.2*, our pattern is not a path.

   The deficiency we are talking about in this fragment is the expressive power of a single property path expression. What I call a backward operator can solve the problem, but such an operator is not an option in property path expressions according to SPARQL 1.1 specifications. What I mean by backward operator is an operator that allows one to go back to a previous node, that will practically allow one to define multiple path branches from this node (every time you define a branch and then go back to this node to define another branch). Not to be confused, the property path inverse operator (`^P`) is not a backward operator (it has a different action effect). For example the property path expression (`P.^P`) does not exactly mean that we move through (`P`) and then we move back to our initial node, rather it means that after moving the first step through (`P`), we then move to any node that has a (`P`) relation wtih the current node (there may exist such nodes different from the initial node).

   If we consider the graph pattern of *figure 3.2* (our case), and the following property path pattern trying to express it:

   ```
   {O1 ^P1/P2/^P2/P3 O3}
   ```

   This property path pattern matches the graph pattern of *figure 3.2*, but also matches other patterns. First, it misses to mention `O2` explicitly. Assuming that we can emphasize the presence of `O2` somewhere else in the query, the previous property path pattern matches the graph pattern of *figure 3.2*, but also matches the following graph patterns (*figure 3.3*).

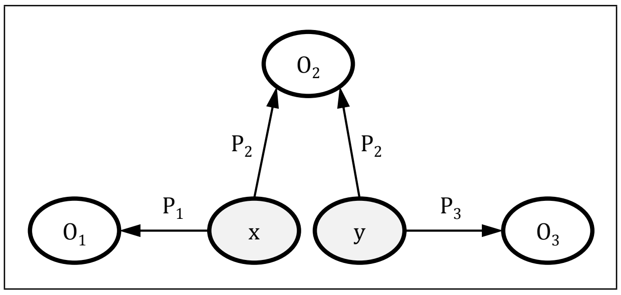

Figure 3.3

2. **AND (.):** The AND operator allows to express the presence of multiple graph patterns simultaneously. The problem is that there is no guarantee that we reference the same path in two different property path patterns. For example if we consider the following basic graph pattern:

```
{
  O1 ^P1/P2 O2.
  O1 ^P1/P3 O3
}
```

There is no guarantee that the first step (`^P1`) is the same step (leads to the same node) in the two property path pattens of the previous basic graph pattern. For example, the previous basic graph pattern can match a graph pattern like this (*figure 3.4*):

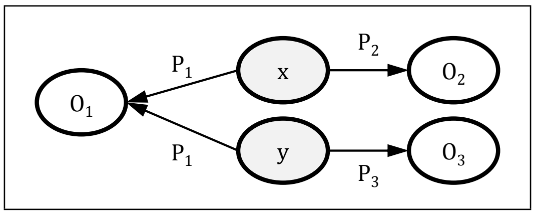

Figure 3.4

3. **UNION:** Given the union of two graph patterns $GP1$ and $GP2$:

```
{GP1} UNION {GP2}
```

The semantics of the union is that the set of solutions of $GP1$ and the set of solutions of $GP2$ will be combined together in a bigger set containing the solutions of both graph patterns.

If each of $GP1$ and $GP2$ cannot be strict enough to represent the pattern of *figure 3.2*, then the UNION operator is not useful; it only appends additional solutions.

In our previous transformation suggestions we were not able to be strict enough to represent the pattern of *figure 3.2*.

4. **OPTIONAL:** Given two graph patterns $GP1$ and $GP2$, we write:

   ```
   {GP1} OPTIONAL {GP2}
   ```

   Such a graph pattern will conserve all the solution mappings raised by $GP1$ (the left hand side of the OPTIONAL operator). If $GP1$ cannot be strict enough to represent the pattern of *figure 3.2*, then the OPTIONAL operator is not useful. In our previous transformation suggestions we were not able to be strict enough to represent the pattern of *figure 3.2*.

5. **FILTER:** With FILTER operator, we can define constraints on the variables of the graph pattern. In the graph pattern of *figure 3.2*, our purpose is to eliminate $?x$, and thus we will not able to define a constraint on it because we lack a reference to it.

We give the following conjecture.

Conjecture 3.4.1. Given a basic graph pattern of the form $\{\langle ?x, P_1, O_1\rangle, \langle ?x, P_2, O_2\rangle, \ldots, \langle ?x, P_n, O_n\rangle\}$, where $n \in \mathbb{N} : n \geq 3$, it is not possible to eliminate $?x$ while conserving the complete semantics of the original basic graph pattern.

3.4.2 Defining the Constraints

In our *supplementary transformation function (ζ')*, we adopt *supplementary transformation rule 3* for dealing with 3 or more property path patterns sharing a non-distinguished variable.

Consider the following basic graph pattern with 3 property path patterns sharing $?x$:
We call this basic graph pattern $\mathbf{GP_1}$

```
{
  ?x P1 O1.
  ?x P2 O2.
  ?x P3 O3
}
```

Using the *supplementary transformation function (ζ')*, we get:
We call this basic graph pattern $\mathbf{GP_2}$

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3 O3.
  O3 ^P3/P1 O1
}
```

The previous transformation is based on linking each property path pattern with the one following it in a cycle, ending by linking the last property path pattern to the first. It does not exactly conserve the semantics of the original basic graph pattern, yet it possesses a vast majority of the its characteristics.

GP_1 matches subgraphs in a data graph that have a pattern similar to that of *figure 3.2*. In addition to subgraph pattern of *figure 3.2*, GP_2 also matches subgraph patterns of *figure 3.5*.

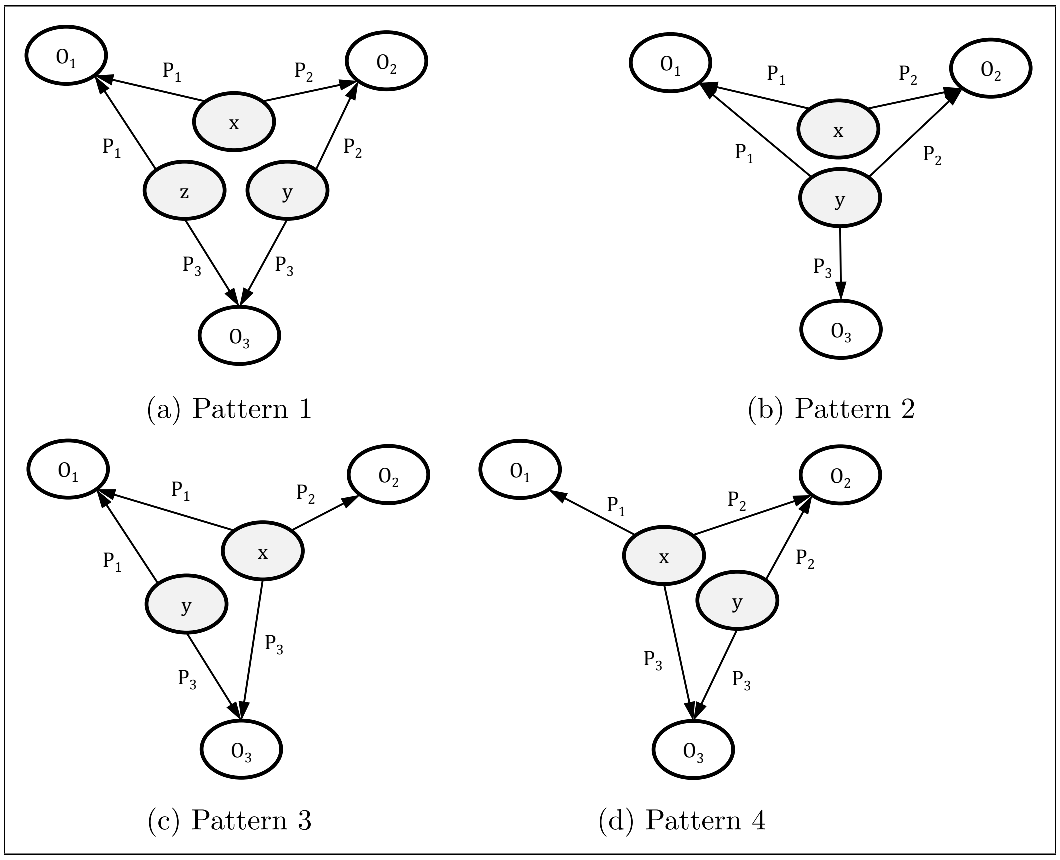

Figure 3.5

Given an acyclic data graph data graph, we can query this data graph using GP_2, with a guarantee that we will not match the false results of *figure 3.5*.

Proposition 3.4.1. *Given a GRDF graph G, a variable sequence $\vec{B}$, and a basic graph pattern P. Let P_i be any property path, and O_i be any RDF term, where i is a natural number. If P consists of $n \in \mathbb{N} : n \geq 3$ property path patterns of the form $\{\langle ?x, P_1, O_1\rangle, \ldots, \langle ?x, P_n, O_n\rangle\}$, $?x \notin \vec{B}$, and G is acyclic, then $\mathcal{A}(\vec{B}, G, P) = \mathcal{A}(\vec{B}, G, \zeta'(P))$.*

Proof. Given a basic graph pattern of the form:

$\{\langle ?x, P_1, O_1\rangle, \langle ?x, P_2, O_2\rangle, \cdots, \langle ?x, P_n, O_n\rangle\}$

our supplementary transformation function (ζ') gives the following form:

$\{\langle O_1, \hat{}P_1/P_2, O_2\rangle, \langle O_2, \hat{}P_2/P_3, O_3\rangle, \cdots, \langle O_n, \hat{}P_n/P_1, O_1\rangle\}$

By definition of the property path sequence operator, $\{\langle s, p/q, o\rangle\}$ is equivalent to $\{\langle s, p, ?x\rangle, \langle ?x, q, o\rangle\}$ for an arbitrary variable $?x$ [11], and by definition of the property path inverse operator, $\{\langle s, \hat{}p, o\rangle\}$ is equivalent to $\{\langle o, p, s\rangle\}$ [11]. The transformed basic graph pattern can be rewritten as:

$$\{\langle ?x_1, P_1, O_1\rangle, \langle ?x_1, P_2, O_2\rangle, \langle ?x_2, P_2, O_2\rangle, \cdots, \langle ?x_n, P_n, O_n\rangle, \langle ?x_n, P_1, O_1\rangle\}$$

For the case where all the variables $?x_1$, $?x_2$, ..., $?x_n$ are assigned to the same graph element when query graph patterns matching is done, this case is equivalent to the our original basic graph pattern, and thus our transformation is complete.

Also possible to be rewritten as:

$$\{\langle O_1, \hat{}P_1, ?x_1\rangle, \langle ?x_1, P_2, O_2\rangle, \langle O_2, \hat{}P_2, ?x_2\rangle, \cdots, \langle O_n, \hat{}P_n, ?x_n\rangle, \langle ?x_n, P_1, O_1\rangle\}$$

For the case where the variables are not assigned the same graph element, it is the case where false positives rise in the results and makes our transformation unsound. In this case, it can be seen from the latest rewriting form that there is a linkage between each property path pattern and the one following it (i.e. the object of the first is the subject of the second), and ending with a linkage between the last property path pattern and the first, and thus forcing a cycle pattern. Therefore, if the queried data graph is acyclic, then our transformation is sound and complete, because the acyclic data graph suppresses such results. □

Another characteristic that we can benefit from in the graph patterns of *figure 3.5*, is the non-functionality of the properties P_1, P_2, and P_3, while there functionality in *figure 3.2*.

Definition 3.4.1 (Functional Property). *Given a GRDF graph G, and a property P, P is functional with respect to the domain of elements* $\{S_1, S_2, \cdots, S_n\}$ *iff* $\forall i \in \{1, 2, \ldots, n\}$*:*
– $\langle S_i, P, O_i\rangle \in G$ *for some element* O_i
– $\forall\langle S_i, P, o\rangle \in G, o$ *is* O_i.

If in the data graph, $\hat{}P_1$ is a functional property with respect to the domain $\{O_1\}$, $\hat{}P_2$ is a functional property with respect to the domain $\{O_2\}$, and $\hat{}P_3$ is a functional property with respect to the domain $\{O_3\}$, we can query this data graph using GP_2, with a guarantee that we will not match the false results of *figure 3.5*.

Proposition 3.4.2. *Given a GRDF graph G, a variable sequence $\vec{B}$, and a basic graph pattern P. Let P_i be any property path, and O_i be any RDF term, where i is a natural number. If P consists of $n \in \mathbb{N} : n \geq 3$ property path patterns of the form $\{\langle ?x, P_1, O_1\rangle, \ldots, \langle ?x, P_n, O_n\rangle\}$, $?x \notin \vec{B}$, and for all properties P_i in G where $i \in \{1, \ldots, n\}$, P_i is a functional property with respect to the domain $\{O_i\}$, then $\mathcal{A}(\vec{B}, G, P) = \mathcal{A}(\vec{B}, G, \zeta'(P))$.*

Proof. Given a basic graph pattern of the form:

$$\{\langle ?x, P_1, O_1\rangle, \langle ?x, P_2, O_2\rangle, \cdots, \langle ?x, P_n, O_n\rangle\}$$

our supplementary transformation function (ζ') gives the following form:

$$\{\langle O_1, \hat{}P_1/P_2, O_2\rangle, \langle O_2, \hat{}P_2/P_3, O_3\rangle, \cdots, \langle O_n, \hat{}P_n/P_1, O_1\rangle\}$$

By definition of the property path sequence operator, $\{\langle s, p/q, o\rangle\}$ is equivalent to $\{\langle s, p, ?x\rangle, \langle ?x, q, o\rangle\}$ for an arbitrary variable $?x$ [11], and by definition of the property path inverse operator, $\{\langle s, \hat{}p, o\rangle\}$ is equivalent to $\{\langle o, p, s\rangle\}$ [11]. The transformed basic graph pattern can be rewritten as:

$\{\langle O_1, \hat{}P_1, ?x_1\rangle, \langle O_2, \hat{}P_2, ?x_1\rangle, \langle O_2, \hat{}P_2, ?x_2\rangle, \cdots, \langle ?O_n, \hat{}P_n, ?x_n\rangle, \langle O_1, \hat{}P_1, ?x_n\rangle\}$

If $\hat{}P_1$, $\hat{}P_2$, ..., $\hat{}P_n$ are functional properties with respect to the domains $\{O_1\}$, $\{O_2\}$, ..., $\{O_n\}$ respectively, then we ensure that all the variables $?x_1$, $?x_2$, ..., $?x_n$ will be assigned the same value in a result, which is semantically equivalent to our original basic graph pattern, and thus making our transformation sound and complete. $\square$

Both *Proposition 3.4.1* and *Proposition 3.4.2* deal with the same situation (i.e. situation of *transformation rule 3*) and reach into the same result, but the first considers cycle checking, while the second considers functional property checking in order to achieve the soundness of our propositions. These two considerations are independent. For example it is possible in our previous demonstrative example that the properties P1, P2 and P3 are functional, but the graph is cyclic, and vice versa. Either the satisfaction of the conditions of *proposition 3.4.1*, or the satisfaction of the conditions of *proposition 3.4.2* is sufficient for our graph patterns to be sound and complete with respect to the query answering problem.

Constraints Checking Complexity

- Cycle checking can be decided by DFS (Depth First Search) which has $O(E + V)$ time complexity (i.e. linear in the size of the graph), where E is the number of edges, and V is the number of nodes of the graph. [21]

- Checking for the functionality of all properties with respect to each domain of a single element in a graph has $O(E + V)$ time complexity (i.e. linear in the size of the graph), where E is the number of edges, and V is the number of nodes of the graph.

  This can be done by visiting each statement (triple) of the RDF graph, and counting for each couple of an RDF node and a property the number of objects they map to. A property is functional with respect to a domain of a single RDF node if the number of objects corresponding to them as a couple is exactly one.

We also mention that constraint checking is required only once and can be done ahead of time, as long as the data graph is not modified. No matter then how many times we want to query this data graph and with what queries, a second constraint checking is not required.

3.5 Dealing with Non-Constrained Datasets

In previous sections, we defined our *query transformation function (ζ')* in order to eliminate non-distinguished variables from a SPARQL query, and we explained that our transformation is complete, but not sound. It becomes sound if some constraints on the dataset are met.

In this section we propose a method in order to deal with non-constrained datasets. Our proposal benefits from our previous transformation rules, but re-introduces the non-distinguished variables in a way that promises a lower computation time than the original query, and only witnesses the same computation time in the worst case. Using this method, the new query is sound and complete on every dataset.

Our previously defined *query transformation function (ζ')*, gives new queries whose results are complete with respect to the original queries, and we are going to benefit from this characteristic. For a transformed query, we add a `FILTER` that will eliminate undesirable solutions, and thus make the query sound.

Example: Consider a basic graph pattern of the following form:

BGP$_0$

```
{
  ?x P1 O1.
  ?x P2 O2.
  ?x P3 O3.
  S4 P4 O4
}
```

We will use the previous transformation rules to get a new basic graph pattern without non-distinguished variables:

BGP$_1$

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3 O3.
  O3 ^P3/P1 O1.
  S4 P4 O4
}
```

then we add a `FILTER`:

BGP$_2$

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3 O3.
  O3 ^P3/P1 O1.
  S4 P4 O4

  FILTER EXISTS
  {
    ?x P1 O1.
    ?x P2 O2.
    ?x P3 O3
  }
}
```

equivalent to:

```
{
  [BGP 1]

  FILTER EXISTS
  {
    [PP patterns of BGP 0
    that contain
    non-distinguished
    variables]
  }
}
```

`EXISTS` is a filter expression, provided in SPARQL 1.1, that tests whether a graph pattern matches in the queried graph; it does not generate any additional bindings. It takes a graph pattern and returns true/false depending on whether the pattern matches the dataset given the bindings in the current group graph pattern, and the dataset. (It returns true if pattern matches and returns false otherwise.)

Variables in the `EXISTS`'s graph pattern that are bound in the current solution mapping take the value that they have from the solution mapping. Variables in the `EXISTS`'s graph pattern that are not bound in the current solution mapping take part in pattern matching.

First the solver will evaluate the solutions for the graph pattern BGP_1. Then given the solution bindings resulting from this evaluation, it uses them to check the validity of the `EXISTS`'s graph pattern.

Notice that the elements `O1`, `O2`, and `O3` are abstract names for the object position, i.e. they are not necessarily IRIs, they can be distinguished variables. The first evaluation step will assign values to the variables among `O1`, `O2`, and `O3`, but not to `?x` (or any other non-distinguished variable). The next evaluation step (filter step) will use the assigned values from the previous step, and try to find the suitable assignments for `?x` only.

We already discussed how the evaluation of BGP_1 (non-distinguished variables eliminated) is faster than BGP_0 (original basic graph pattern). Then the filter step should benefit from the assigned values rather than trying all possible combination of the existing variables.

Assuming that all of the elements `O1`, `O2`, and `O3` are distinguished variables, the worst case is when every possible combination of 3 nodes of the data graph is a solution for BGP_1. In such case the filter step will have exactly the same computation time as if there were no assignments (which means it has to try every combination of 3 nodes).

In BGP_2, the `EXISTS`'s graph pattern consists only of property path patterns of BGP_0 that contain non-distinguished variables. Property path patterns that do not contain non-distinguished variables do not add any new information, they are already matched in the main pattern.

Although the new query form seems to be bigger in size (number of property path patterns), but this is not the only deciding factor for the computation time of a query. The number of times you check these triples also must be taken into account, which is equal to the number of mappings tried during the evaluation. As we reduced the number of variables in each of the 2 steps of evaluation, the computation time in each of them should decrease exponentially as the number of mappings variables to RDF terms decreases exponentially.

3.5.1 Nested FILTERs

In the previous example of filtering we considered a single non-distinguished variable `?x`. In case of presence of more than one non-distinguished variable, exactly the same strategy works as done with single non-distinguished variable (we add a single `FILTER`).

Nesting `FILTER`s in the case of multiple non-distinguished variables further improves the performance of query solving because it decreases the number of mappings that would be applied by the solver while checking the `EXISTS`'s graph pattern. We allow only one non-distinguished variable in the `EXISTS`'s graph pattern of each nested `FILTER`.

Example: Consider the following basic graph pattern:

BGP$_0$

```
{
  ?x P1 O1.
  ?x P2 O2.
  ?x P3?y
}
```

Eliminating `?x` gives the following basic graph pattern:

BGP$_1$

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3 ?y.
  ?y ^P3/P1 O1
}
```

Then eliminating `?y` gives the following basic graph pattern:

BGP$_1$

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3/^P3/P1 O1
}
```

Now we can add a simple filter as follows:

BGP$_3$

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3/^P3/P1 O1

  FILTER EXISTS
  {
    ?x P1 O1.
    ?x P2 O2.
    ?x P3?y
  }
}
```

equivalent to:

```
{
  [BGP 2]

  FILTER EXISTS
  {
    [PP patterns of BGP 0
    that contain
    non-distinguished
    variables]
  }
}
```

or nested filters (which is more promising) as follows:

BGP_4 *(Final)*

```
{
  O1 ^P1/P2 O2.
  O2 ^P2/P3/^P3/P1 O1

  FILTER EXISTS
  {
    O2 ^P2/P3 ?y.
    ?y ^P3/P1 O1

    FILTER EXISTS
    {
      ?x P1 O1.
      ?x P2 O2.
      ?x P3?y
    }
  }
}
```

equivalent to:

```
{
  [BGP 2]

  FILTER EXISTS
  {
    [PP patterns of BGP 1
    that contain ?y]

    FILTER EXISTS
    {
      [PP patterns of BGP 0
      that contain ?x]

    }
  }
}
```

In BGP_4, the first filter consists of property path patterns of BGP_1 that contain `?y`. We know that property path patterns of BGP_1 only contains non-distinguished variable `?y` (`?x` was eliminated).

The second filter consists of property path patterns from BGP_0 that contain `?x`. These property path patterns can also contain `?y`, but there values are inherited from the `EXISTS`'s graph pattern mappings of the previous filter.

We modify our *variable eliminating function (ζ_x)* in order to cope with the addition of `FILTER`s as described above. We call our modified function the *pseudo variable eliminating function ($\mathring{\zeta}_x$).*

Definition 3.5.1 (Pseudo Variable Eliminating Function ($\mathring{\zeta}_x$))**.** *Let P be a basic graph pattern, P_1 and P_2 two graph patterns, K a filter constraint, we define the pseudo variable eliminating function $\mathring{\zeta}_x$ inductively:*

- $\mathring{\zeta}_x(\emptyset) = \emptyset$ ($\emptyset$, i.e. the empty set)
- $\mathring{\zeta}_x(P) = P$ (if all PP patterns containing $?x$ in P are reflexive)
- $\mathring{\zeta}_x(P) = (\zeta_x(\zeta'(\delta_x(\alpha_x(P)))) \cup \bar{\alpha}_x(P))$ `FILTER EXISTS` $\alpha_x(P)$ (otherwise)

- $\mathring{\zeta}_x(P_1 \; \textit{UNION} \; P_2) = \mathring{\zeta}_x(P_1) \; \textit{UNION} \; \mathring{\zeta}_x(P_2)$
- $\mathring{\zeta}_x(P_1 \; \textit{AND} \; P_2) = \mathring{\zeta}_x(P_1) \; \textit{AND} \; \mathring{\zeta}_x(P_2)$
- $\mathring{\zeta}_x(P_1 \; \textit{OPT} \; P_2) = \mathring{\zeta}_x(P_1) \; \textit{OPT} \; \mathring{\zeta}_x(P_2)$

- $\mathring{\zeta}_x(P_1 \; \textit{FILTER} \; K) = P_1 \; \textit{FILTER} \; K$ (if K is not an `EXISTS` expression, K contains $?x$)
- $\mathring{\zeta}_x(P_1 \; \textit{FILTER} \; K) = \mathring{\zeta}_x(P_1) \; \textit{FILTER} \; K$ (otherwise)

We also modify our *query transformation function (ζ)*. The purpose of the new definition is just to emphasize the usage of $\mathring{\zeta}_x$ instead of ζ_x. We call our modified function the *pseudo query transformation function ($\mathring{\zeta}$).*

Definition 3.5.2 (Pseudo Query Transformation Function (ζ))**.** *Given a query $Q(\vec{B}, P)$, where $\vec{B}$ is a variable name sequence, and P is the query graph pattern. Let $?x_1$, $?x_2$, ...,*

$?x_n$ be the set of non-distinguished variables in Q. We define the pseudo query transformation function $\mathring{\zeta}$:

$$\mathring{\zeta}(Q(\vec{B}, P)) = Q(\vec{B}, P')$$

where

$$P' = \mathring{\zeta}_{x_1}(\mathring{\zeta}_{x_2}(\ldots \mathring{\zeta}_{x_n}(P)))$$

The *pseudo query transformation function ($\mathring{\zeta}$)* will take care about filter nesting. For each variable it eliminates in the sequence, it adds a `FILTER` at its corresponding level.

3.6 Conclusion

The purpose of our work is to eliminate non-distinguished variables from a SPARQL query by taking advantage property paths. Such elimination is important because it guarantees lower computation complexity.

In this chapter, we studied different forms and fragments of SPARQL queries, and we accordingly proposed a *query transformation function.* It takes a SPARQL query as input, and returns a transformed SPARQL query by eliminating non-distinguished variables where such elimination is possible. Our transformation is based on rewriting SPARQL queries with property paths.

For an arbitrary dataset, our transformation function is complete but unsound. The form of the transformed queries allows us to define constraints on the queried data graph, that suppresses undesirable results. Our transformation function is sound and complete if the queried data graph is acyclic, or if certain properties, corresponding to their usage in the query, are functional in the queried data graph.

To deal with non-constrained datasets (any possible dataset), we proposed an alternative solution that benefits from our transformation function, but re-introduces the non-distinguished variables in a way that promises a lower computation time than the original query, and only witnesses the same computation time in the worst case. Our proposed solution is based on using the filter expression `EXISTS`. This filter expression takes a graph pattern, other than the query graph pattern, and checks if it matches the dataset, using the bindings from the current solution mapping. It accepts the solution mapping if the graph pattern matches. The idea is to suppress other undesirable solutions using this filter expression.

Chapter 4

Conclusion

The purpose of our work is to eliminate non-distinguished variables from a SPARQL query by taking advantage property paths. Such elimination is important because it guarantees lower computation complexity.

It may not be intuitive for users to write queries without non-distinguished variables. Such variables, which are source of extra complexity, allow users to write meaningful queries more easily. For this reason we suggest automatic transformation of queries, on fly, before their execution.

In this document, we first studied and reported on the Semantic Web technologies: RDF and SPARQL.

The Resource Description Framework (RDF) is a framework for expressing information about resources in the World Wide Web. RDF can be expressed in a variety of formats. We have presented an abstract syntax for RDF, then we provided a detailed study about its semantics, and an inference mechanism based on the abstract syntax.

We also reported on SPARQL - a query language for accessing RDF published data. We have presented the syntax and the different fragments of this query language. We then provided its semantics, and an algebraic manipulation for it. We also introduced one of its recent extensions, concerning property paths, which has been proposed by different studies and was recently introduced in the latest version, SPARQL 1.1.

The syntax and semantics of these technologies are the necessary building blocks for our work in this document.

For the purpose of our work, we studied different forms and fragments of SPARQL queries, and we accordingly proposed a *query transformation function*. This function takes a SPARQL query as input, and returns a transformed SPARQL query by eliminating non-distinguished variables where such elimination is possible. Our transformation is based on rewriting SPARQL queries with property paths.

For an arbitrary dataset, our transformation function is complete but unsound. The form of the transformed queries allows us to define constraints on the queried data graph, that suppresses undesirable results. Our transformation function is sound and complete if the queried data graph is acyclic, or if certain properties, corresponding to their usage in the query, are functional in the queried data graph.

To deal with non-constrained datasets (any possible dataset), we proposed an alternative solution that benefits from our transformation function, but re-introduces the non-distinguished variables in a way that promises a lower computation time than the original query, and only witnesses the same computation time in the worst case. Our proposed solution is based on using the filter expression `EXISTS`. This filter expression takes a graph pattern, other than the query graph pattern, and checks if it matches the dataset, using the bindings from the current solution mapping. It accepts the solution mapping if the graph pattern matches. The idea is to suppress other undesirable solutions using this filter expression.

The results of our work is a potential and important starting point for further future works. We suggest that a following work be about static analysis of SPARQL queries. The completeness of our transformation function in general, and its soundness and completeness in specific environments are basic ideas, over which we can build further SPARQL query analysis.

Bibliography

[1] Faisal Alkhateeb. *Querying RDF(S) with Regular Expressions.* PhD thesis, Université Joseph Fourier, Grenoble (FR), 2008.

[2] Faisal Alkhateeb, Jean-François Baget, and Jérôme Euzenat. Extending SPARQL with regular expression patterns (for querying RDF). *J. Web Sem.*, 7(2):57–73, 2009.

[3] Faisal Alkhateeb and Jérôme Euzenat. Answering SPARQL queries modulo RDF Schema with paths. *CoRR*, abs/1311.3879, 2013.

[4] Jean-François Baget. RDF entailment as a graph homomorphism. In *Proceedings of the 4th International Conference on The Semantic Web*, ISWC'05, pages 82–96, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Michel Chein and Marie-Laure Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs.* Advanced Information and Knowledge Processing. Springer, 2008.

[6] Olivier Corby, Rose Dieng, and Cédric Hébert. A conceptual graph model for W3C Resource Description Framework. In *Proceedings of the Linguistic on Conceptual Structures: Logical Linguistic, and Computational Issues*, ICCS '00, pages 468–482, London, UK, UK, 2000. Springer-Verlag.

[7] Martin Duerst and Michel Suignard. RFC 3987: Internationalized Resource Identifiers (IRIs). RFC 3987 (Proposed Standard), see `http://www.ietf.org/rfc/rfc3987.txt`, January 2005.

[8] Jérôme Euzenat. Semantic web semantics. Lecture notes, INRIA & LIG, Montbonnot, France, January 2014. `http://exmo.inria.fr/teaching/sw/poly/semwebsem.pdf`.

[9] Fabien Gandon and Guus Schreiber. RDF 1.1 XML syntax. W3C recommendation, W3C, February 2014. `http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/`.

[10] Claudio Gutierrez, Carlos Hurtado, and Alberto O. Mendelzon. Foundations of semantic web databases. In *Proceedings of the Twenty-third ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '04, pages 95–106, New York, NY, USA, 2004. ACM.

[11] Steven Harris and Andy Seaborne. SPARQL 1.1 query language. W3C recommendation, W3C, March 2013. `http://www.w3.org/TR/2013/REC-sparql11-query-20130321/`.

[12] Patrick Hayes. RDF semantics. W3C recommendation, W3C, February 2004. `http://www.w3.org/TR/2004/REC-rdf-mt-20040210/`.

[13] Markus Lanthaler, David Wood, and Richard Cyganiak. RDF 1.1 concepts and abstract syntax. W3C recommendation, W3C, February 2014. `http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/`.

[14] Eric Miller and Frank Manola. RDF primer. W3C recommendation, W3C, February 2004. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[15] Peter Patel-Schneider and Patrick Hayes. RDF 1.1 semantics. W3C recommendation, W3C, February 2014. `http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/`.

[16] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[17] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. nSPARQL: A navigational language for RDF. *Web Semantics: Science, Services and Agents on the World Wide Web*, 8(4), 2010.

[18] Axel Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 787–796, New York, NY, USA, 2007. ACM.

[19] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`.

[20] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of SPARQL query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[21] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[22] Herman J. ter Horst. Completeness, decidability and complexity of entailment for RDF schema and a semantic extension involving the OWL vocabulary. *Web Semant.*, 3(2-3):79–115, October 2005.