PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

OPTIMIZATION OF SEMANTIC WEB QUERIES USING SPARQL PATTERN TREES

ANDRÉS IGNACIO LETELIER NAGEL

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
MARCELO ARENAS S.

Santiago de Chile, September 2013

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
SCHOOL OF ENGINEERING

OPTIMIZATION OF SEMANTIC WEB QUERIES USING SPARQL PATTERN TREES

ANDRÉS IGNACIO LETELIER NAGEL

Members of the Committee:
MARCELO ARENAS S.
JUAN L. REUTTER D.
JORGE PÉREZ R.
ANDRÉS GUESALAGA M.

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, September 2013

To everyone who doesn't have a thesis dedicated to them.

ACKNOWLEDGEMENTS

Thanks to my parents, for their constant support and for always believing in me.

Thanks to my advisor, Marcelo Arenas, for his never ending patience.

Thanks to Sebastian Skritek and Reinhard Pichler from TU Wien, for their hospitality and for helping me with the heavy lifting.

Thanks to Jorge Pérez from Universidad de Chile, for all his support, advice and wisdom.

Thanks to Soledad Carrión, for always having the greatest disposition to help.

Thanks to Sean Plott, for teaching me that if at first you don't succeed, try and try again.

Thanks to my office mates at the university, for laughing at joke in the dedication.

And many thanks to Claudia. Without her encouragement, support and prescription pad, this thesis would have never been completed.

Contents

List of Tables

List of Figures

ABSTRACT

SPARQL is the standard query language for Semantic Web data. Since data on the Web is inherently incomplete, it is crucial for users to be able to obtain partial answers without having the query evaluation fail, and to restrict answers to those that are relevant for them. While the filtering feature is standard in classical query languages, such as SQL for relational databases, the optionality feature is one of the most complicated constructors in SPARQL, and makes this language significantly different from others. Thus, common optimization techniques for relational queries can no longer be applied directly and need to be revised.

In this work we study optimization techniques specifically for SPARQL, focusing primarily on the *optionality* and *filtering* features by extending the notion of *pattern trees*. We mainly restrict ourselves to the fragment of well-designed SPARQL queries, which has been previously shown to have good properties and behavior. We first extend the definition of pattern trees to capture the class of well-designed graph patterns composed of AND, OPT and FILTER operators. We then add an additional transformation rule based on the FILTER operator. Finally, we propose a strategy for query optimization by using transformation rules for pattern trees, and show their usefulness by studying their applicability and effect over several publicly available datasets of Semantic Web data.

Keywords: SPARQL, RDF, Semantic Web, optimization, rewriting, database models

RESUMEN

SPARQL es el lenguaje de consulta estándar para datos en la Web Semántica. Dado que la información en la Web es inherentemente incompleta, es crucial para los usuarios poder obtener respuestas parciales sin que la evaluación de la consulta fracase, y poder restringir sus respuestas a aquellas que les parecen relevantes. Mientras que la facultad de filtrar resultados es estándar en lenguajes de consulta clásicos, como SQL en bases de datos relacionales, la característica de opcionalidad es uno de los operadores más complejos de SPARQL, y hace a este lenguaje significativamente distinto de otros. Por lo tanto, las técnicas habituales de optimización para consultas relacionales no se pueden aplicar directamente, y necesitan ser revisadas.

En este trabajo estudiamos técnicas de optimización específicas para SPARQL. Nos enfocamos principalmente en las características de *opcionalidad* y *filtrado* de SPARQL, extendiendo la noción de *pattern trees*. Nos restringimos al fragmento de consultas de SPARQL conocidas como "bien diseñadas", que han demostrado previamente tener buenas propiedades y comportamiento al evaluarse. En primer lugar, extendemos la definición de *pattern trees* para capturar la clase de patrones bien diseñados compuestos por los operadores AND, OPT y FILTER. Luego presentamos una nueva regla de transformación basada en el operador FILTER. Finalmente, proponemos una estrategia para optimizar consultas usando las reglas de transformación para *pattern trees*, mostrando su utilidad al estudiar su aplicabilidad y efectividad sobre varios conjuntos de datos de Web Semántica públicamente disponibles.

Palabras Claves: SPARQL, RDF, Web semántica, optimización, reescritura, modelos de bases de datos

Chapter 1. INTRODUCTION

1.1. Background

The term "Semantic Web" refers to a web of data that can be readable and processable by machines and not just by humans. While the term was coined by Tim Berners Lee, it has largely been an initiative of the World Wide Web Consortium (W3C). In 1999, the *Resource Description Framework* (RDF) was published as a W3C Recommendation, designed as a data model for representing information about World Wide Web resources (Lassila & Swick, 1999). Jointly with its release, the natural problem of querying RDF data was raised, and thus the question of how to manage RDF data has been in the focus of the Semantic Web community. This problem still remains partially unsolved, but an important first step was the release of SPARQL as a W3C recommendation (Prud'Hommeaux & Seaborne, 2008). With time, this has become the standard query language for RDF. Since then, the amount of RDF data published on the Web has grown constantly, as shown by the popularity of initiatives like the Open Linked Data project (Berners-Lee, 2006; Bizer, Heath, & Berners-Lee, 2009), and even being adopted by the US and UK governments in their respective Open Government Data initiatives (*DATA.GOV.UK*, 2013; *Data.gov*, 2013).

This increase in notoriety has drawn the attention of the research community, as RDF and SPARQL offer many new and interesting problems to tackle. Several research efforts are being directed towards understanding the fundamental properties of the language, as well as developing new techniques to handle these problems (Pérez, Arenas, & Gutierrez, 2006; Polleres, 2007; Abadi, Marcus, Madden, & Hollenbach, 2007; Weiss, Karras, & Bernstein, 2008; Sidirourgos, Goncalves, Kersten, Nes, & Manegold, 2008; Angles & Gutierrez, 2008; Schmidt, Hornung, Küchlin, Lausen, & Pinkel, 2008; Pérez, Arenas, & Gutierrez, 2009; Neumann & Weikum, 2010; Schmidt, Meier, & Lausen, 2010; Arenas & Pérez, 2011; Letelier, Pérez, Pichler, & Skritek, 2012b, in press).

RDF and SPARQL will be formally introduced in Chapter 2. However, at is core, an RDF dataset is a set of triples of the form (s, p, o). This syntax is used to describe a directed graph with named arcs; for this reason, an RDF dataset is interchangeably referred to as an RDF graph. SPARQL is in essence a graph pattern matching language. Its fundamental component, the SPARQL triple pattern, is an RDF triple which can have variables instead of labels. One could see an RDF graph as merely being a set of tuples; then, if one restricts SPARQL to conjunctions of triple patterns (which are basically ternary atoms), then one can see that basic SPARQL queries are essentially relational conjunctive queries. Thus, one can connect this fragment of SPARQL to the decades of work behind conjunctive queries.

In a classical setting there are two closely associated problems: static analysis and optimization of queries. Static analysis refers to the study of queries without any knowledge of the dataset they will be evaluated over. Problems in this area include the idea of query containment (which means deciding whether the answers of one query will always be contained in the answers of another query, when evaluated over the same dataset), and equivalence (which means deciding whether two different queries will always return the same results when evaluated over the same dataset). On the other hand, query optimization refers to the study of how to more efficiently evaluate a given query. When evaluating a query written in a language like SQL, the query is first *parsed*, that is, turned into a parse tree representing the structure of the query in a useful way. The parse tree is then transformed into an expression tree of relational algebra, which is termed a *logical query plan*, or just query plan for short. In picking a query plan, one has opportunities to apply many different algebraic operations, with the goal of hopefully producing the best query plan (Garcia-Molina, Ullman, & Widom, 2009, p. 759). In reality, obtaining an optimal query plan is unrealistic; it is more important to avoid the worst plans and find a good plan. This is a problem which has been studied for several decades, and optimization techniques based on query plans for relational queries are already part of the folklore of the field (Garcia-Molina et al., 2009; Ramakrishnan & Gehrke, 2003).

Unfortunately, before Letelier et al. (2012b), there had been little work regarding the static analysis and optimization of SPARQL queries. While there have been exceptions (Serfiotis, Koffina, Christophides, & Tannen, 2005; Schmidt et al., 2010; Chekol, Euzenat, Genevès, & Layaïda, 2011; Stocker, Seaborne, Bernstein, Kiefer, & Reynolds, 2008), most of them focus on the previously mentioned fragment of SPARQL, taking the conjunctive query approach. However, it is SPARQL's ability to work with *partial* information which makes it an interesting language: since data on the web is inherently incomplete, it is essential for users to be able to request optional information. For example, if one were to request the names, phone numbers and addresses of a group of people using conjunctive queries, and one person's phone number was unknown, then there would be no information regarding that person. It is SPARQL's OPT operator which enables users to request optional information: in the previous example, if users were to ask for the names of people, and optionally their address, and optionally their phone number, then the answer to the query would include all known names in the dataset, along with addresses and phone numbers whenever they are available. SPARQL also offers a *filtering* feature through its FILTER operator. Through it, users can choose which results they consider relevant, and exclude those that are not from the answer to their query.

Unfortunately, when one goes beyond the basic fragment of SPARQL, things get considerably more complicated (Pérez et al., 2009; Arenas & Pérez, 2011). The OPT operator has proven to be particularly complex. Furthermore, its use in practice is substantial (Gallego, Fernández, Martínez-Prieto, & de la Fuente, 2011; Picalausa & Vansummeren, 2011), which makes its study essential. Pérez et al. (2009) showed how the combined complexity of the evaluation problem for SPARQL (deciding whether a set of variable mappings is part of the result set of the evaluation of a given SPARQL query over a given dataset) rises dramatically, from PTIME-membership for the most basic fragment, up to PSPACE-completeness when including the OPT feature. Nevertheless, the same work also defined a natural and well-behaved fragment of SPARQL with OPT and FILTER operators: the class of *well-designed* SPARQL queries. It was shown in Pérez

et al. (2009) that the combined complexity of the evaluation problem for well-designed SPARQL queries is coNP-complete, which is much more tractable than the general case of SPARQL queries with the OPT operator.

Letelier et al. (2012b) attacked the problem of static analysis of well-designed SPARQL queries which include the OPT operator, focusing mostly on query containment and equivalence. To study these problems, the authors created a new tool for SPARQL analysis: a tree representation of SPARQL queries called *pattern tree*. Among other things, Letelier et al. (2012b) showed that pattern trees can be viewed as query plans for well-designed SPARQL queries. Furthermore, that work introduced an algebra of pattern trees, composed of four transformation rules. However, the authors excluded the FILTER operator from their analysis. In this thesis we extend the work done in Letelier et al. (2012b) using it as our foundation, but adding the FILTER operator to capture the entire class of well-designed SPARQL queries.

1.2. Summary of contributions

The first half of this thesis deals with extending the notion of pattern trees, previously defined in Letelier et al. (2012b). We have modified it to include the filtering feature of SPARQL, namely the FILTER operator. Furthermore, we have extended several propositions, lemmas and theorems along with their proofs. In particular, we have shown that pattern trees are expressive enough to represent the entirety of well-designed SPARQL graph patterns, meaning that any well-designed SPARQL graph pattern can be transformed into a pattern tree. We show how this particular pattern tree can be constructed in polynomial time from a given SPARQL query, and how a pattern tree can be turned into a SPARQL graph pattern. We also define how pattern trees can be evaluated to obtain the same result set as one would obtain from evaluating an equivalent query using SPARQL's usual semantics.

We have also extended the algebra of pattern trees by minimally adapting the four transformation rules presented in Letelier et al. (2012b), in such a way that they behave

nicely in the presence of FILTER operators, but retain their previous behavior in their absence. Additionally, we have created a fifth transformation rule, which takes advantage of the FILTER operator to improve query evaluation time.

For the second half of this work, we have implemented a SPARQL query engine based on pattern trees, called the Top-down engine, along with the five transformation rules, by using the Jena framework (Carroll et al., 2004). We have used this implementation to test the applicability of the transformation rules for query optimization, over three publicly available datasets containing over forty million real world queries.

In particular, we have shown massive decreases in query execution time when applying each transformation rule, and done an in-depth analysis of the effect of our new rule.

1.3. Thesis outline and structure

In Chapter 2, we present a formalization for the Resource Description Framework (RDF) and the SPARQL query language.

Chapter 3 formally introduces the extended version of pattern trees, well-designed and quasi well-designed pattern trees. It then shows our extension of the existing transformation rules for quasi well-designed pattern trees, and presents our new, additional transformation rule.

Chapter 4 presents a strategy for well-designed query optimization using the transformation rules introduced in the previous chapter. The technique is tested over an extensive dataset, and the overall effect of each transformation rule is discussed.

Finally, Chapter 5 summarizes the results of this thesis and proposes future lines of research to further extend this work.

Chapter 2. PRELIMINARIES

2.1. Resource Description Framework

Assume there are three pairwise, disjoint, infinite sets $\mathbf{U}$, $\mathbf{B}$ and $\mathbf{L}$ (URIs, Blank nodes and Literals). A triple $(s, p, o) \in (\mathbf{U} \cup \mathbf{B}) \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{B} \cup \mathbf{L})$ is called an *RDF triple*. In this tuple, s is the *subject*, p the *predicate* and o the *object*. An *RDF graph* is a finite set of RDF triples. For an in depth description and formalization of RDF, see the work by Gutiérrez, Hurtado, and Mendelzon (2004).

In our work we focus only on *bound* RDF graphs; that is, RDF graphs that do not contain blank nodes. We also do not make any distinctions between URIs and Literals. Therefore, in our context, an RDF triple is a tuple in $\mathbf{U} \times \mathbf{U} \times \mathbf{U}$, and an RDF graph (or graph for short) is a finite set of RDF triples. The active domain of an RDF graph G, denoted by $\mathrm{dom}(G)$ with $\mathrm{dom}(G) \subseteq \mathbf{U}$ is the set of uniform resource identifiers (URIs) actually appearing in G.

2.2. SPARQL

SPARQL (Prud'Hommeaux & Seaborne, 2008) is the standard query language for RDF. It is essentially a graph pattern matching language. A SPARQL query is of the form $H \leftarrow B$, where B, the *body* of the query, is a complex RDF graph pattern expression that may include RDF triples with variables, conjunctions, disjunctions, optional parts, and constraints over the values of the variables. On the other hand, H, the *head* of the query, is an expression that indicates how to construct the answer to the query. To produce the answer of evaluating a query Q against a dataset D, the body of Q is matched against D to produce a set of bindings. These are then processed with the information in the head of Q, using classical relational operators like projection and distinct, to obtain the final result of the query. This work only deals with the handling of the body of the query.

We next formalize the *graph pattern matching facility* of SPARQL, which forms the core of the language. Assume the existence of an infinite set $\mathbf{V}$ of variables disjoint from $\mathbf{U}$. We denote variables in $\mathbf{V}$ by using a question mark, as in $?X$. A SPARQL triple pattern is then a tuple $t \in (\mathbf{U} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{V}) \times (\mathbf{U} \cup \mathbf{V})$.

Complex graph patterns are constructed from triple patterns by using operators AND, OPT, UNION, and FILTER. In this work we focus on the SPARQL fragment composed of the operators AND, OPT and FILTER. Formally, SPARQL graph patterns are recursively defined as follows:

(i) a triple pattern t is a graph pattern,

(ii) if P_1 and P_2 are graph patterns, then $(P_1 \ \mathrm{AND} \ P_2)$ and $(P_1 \ \mathrm{OPT} \ P_2)$ are graph patterns, and

(iii) if P is a graph pattern and R is a *built-in* condition, then $(P \ \mathrm{FILTER} \ R)$ is a graph pattern.

A SPARQL *built-in* condition is constructed using elements from the set $\mathbf{U} \cup \mathbf{V}$ and constants, logical connectives $(\neg, \wedge, \vee)$, inequality symbols $(<, \leq, >, \geq)$, the equality symbol $(=)$, unary predicates like bound, isBlank and isURI, and other features detailed in Prud'Hommeaux and Seaborne (2008). In this paper we restrict ourselves to the fragment where the built-in condition is a Boolean combination of terms constructed by using $=$ and bound, that is:

(i) If $?X, ?Y \in \mathbf{V}$ and $c \in \mathbf{U}$, then $\mathrm{bound}(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions.

(ii) If R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \wedge R_2)$ and $(R_1 \vee R_2)$ are built-in conditions.

For a triple pattern t, we write $\mathrm{vars}(t)$ to denote the set of variables occurring in t; for a built-in condition R we write $\mathrm{vars}(R)$ for the set of variables mentioned in R, and for

a graph pattern P we write $\mathrm{vars}(P)$ for the set of variables that occur in the triples and built-in conditions that compose P.

To define the semantics of SPARQL graph patterns, we follow closely the definitions proposed in Pérez et al. (2009). A mapping μ is a partial function $\mu : \mathbf{V} \rightarrow \mathbf{U}$. The domain of μ, denoted by $\mathrm{dom}(\mu)$, is the set of all variables from $\mathbf{V}$ for which μ is defined. Given a triple pattern t and a mapping μ such that $\mathrm{vars}(t) \subseteq \mathrm{dom}(\mu)$, we denote by $\mu(t)$ the RDF triple obtained by replacing the variables in t according to μ. Given two mappings μ_1 and μ_2, we say that μ_1 and μ_2 are *compatible*, denoted by $\mu_1 \sim \mu_2$, if for every $?X \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ it holds that $\mu_1(?X) = \mu_2(?X)$. Notice that, for compatible mappings μ_1 and μ_2, we have that $\mu_1 \cup \mu_2$ is also a mapping and is such that $(\mu_1 \cup \mu_2)(?X)$ is $\mu_1(?X)$ if $?X \in \mathrm{dom}(\mu_1)$, or $\mu_2(?X)$ otherwise. Also notice that the mapping with empty domain, denoted by $\mu_\emptyset$, is compatible with any mapping.

Before defining the semantics of SPARQL graph patterns, we define some operations between sets of mappings that resemble relational operators over sets of tuples. Let M_1 and M_2 be sets of mappings. We define the *join* and the *left-outer join* between M_1 and M_2 as follows:

$$M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}$$

$$M_1 \mathbin{⟕} M_2 = (M_1 \bowtie M_2) \cup \{\mu \in M_1 \mid \forall \mu' \in M_2 : \mu \not\sim \mu'\}.$$

Given a SPARQL built-in condition R and a partial mapping μ, we say that μ *satisfies* R (shortened to $\mu \models R$) if:

(i) R is $\mathrm{bound}(?X)$ and $?X \in \mathrm{dom}(\mu)$,

(ii) R is $?X = c$, $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) = c$,

(iii) R is $?X = ?Y$, $?X, ?Y \in \mathrm{dom}(\mu)$ and $\mu(?X) = \mu(?Y)$,

(iv) R is $(\neg R_1)$, R_1 is a built-in condition and it is not the case that $\mu \models R_1$,

(v) R is $(R_1 \wedge R_2)$, R_1 and R_2 are built-in conditions and it is the case that $\mu \models R_1$ and $\mu \models R_2$, and

(vi) R is $(R_1 \vee R_2)$, R_1 and R_2 are built-in conditions and it is the case that $\mu \models R_1$ or $\mu \models R_2$.

We now have all the necessary prerequisites to formalize the evaluation of a SPARQL graph pattern over an RDF graph G as a function $[\![\cdot]\!]_G$, which given a graph pattern returns a set of mappings. Formally, $[\![P]\!]_G$ is defined recursively as follows (Pérez et al., 2009):

(i) If P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{vars}(t) \text{ and } \mu(t) \in G\}$.

(ii) If $P = (P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.

(iii) If $P = (P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \mathbin{⟕} [\![P_2]\!]_G$.

(iv) If $P = (P_1 \text{ FILTER } R)$, then $[\![P]\!]_G = \{\mu | \mu \in [\![P_1]\!]_G \text{ and } \mu \models R\}$.

We say that two patterns P_1 and P_2 are equivalent, denoted by $P_1 \equiv P_2$, if for every RDF graph G, it holds that $[\![P_1]\!]_G = [\![P_2]\!]_G$. Notice that mappings explicitly refer to the variable names. Hence, unlike the case of conjunctive queries, the actual names of the variables matter, since two graph patterns containing different sets of variables can never be equivalent. In Pérez et al. (2009), the authors show several algebraic properties for graph patterns. In particular they show that AND is commutative and associative, which allows us to drop parentheses from sequences of AND operators.

Note that we described the set-semantics of SPARQL, while the W3C Recommendation defines a bag-semantics for query answering (Prud'Hommeaux & Seaborne, 2008). Nevertheless, for the fragment considered in this paper (allowing only for AND, OPT and FILTER), both semantics coincide (Pérez et al., 2006). We have therefore restricted ourselves only to the set-semantics of the language.

Example 2.1 (From Pérez et al. (2009))**.** *Consider an RDF graph G storing incomplete information about professors in a university with the following triples, and the SPARQL graph pattern P_1:*

{ *(R_1, name, paul),* *(R_1, phone, 777-3426),*
(R_2, name, john), *(R_2, email, john@acd.edu),*
(R_3, name, george), *(R_3, webPage, www.george.edu),*
(R_4, name, ringo), *(R_4, email, ringo@acd.edu),*
(R_4, webPage, www.starr.edu), *(R_4, phone, 888-4537)* *}*

$$P_1 = \big(\big((?A, \textit{name}, ?N) \text{ OPT } (?A, \textit{email}, ?E)\big) \text{ OPT } (?A, \textit{webPage}, ?W)\big)$$

Intuitively, we are trying to retrieve the name of the resources in G. Optionally, for the resources that have an email we retrieve the email, and, optionally, for the resources that have a Web page we retrieve the Web page. When we evaluate P_1 over G we obtain the set of mappings $[\![P_1]\!]_G = \{\mu_1, \mu_2, \mu_3, \mu_4\}$ where

$\mu_1 = \{?A \to R_1, ?N \to \textit{paul}\}$,
$\mu_2 = \{?A \to R_2, ?N \to \textit{john}, ?E \to \textit{john@acd.edu}\}$,
$\mu_3 = \{?A \to R_3, ?N \to \textit{george}, ?W \to \textit{www.george.edu}\}$,
$\mu_4 = \{?A \to R_4, ?N \to \textit{ringo}, ?E \to \textit{ringo@acd.edu}\}$,
$\mu_4 = \{?W \to \textit{www.starr.edu}\}$.

Also, consider now pattern P_2 given by the following expression:

$$P_2 = \big((?A, \textit{name}, ?N) \text{ OPT } \big((?A, \textit{email}, ?E) \text{ OPT } (?A, \textit{webPage}, ?W)\big)\big)$$

In this case the evaluation of P_2 over G is the set of mappings $[\![P_2]\!]_G = \{\mu_1, \mu_2, \mu_3, \mu_4\}$, where

$\mu_1 = \{?A \to R_1, ?N \to \textit{paul}\}$,
$\mu_2 = \{?A \to R_2, ?N \to \textit{john}, ?E \to \textit{john@acd.edu}\}$,
$\mu_3 = \{?A \to R_3, ?N \to \textit{george}\}$,
$\mu_4 = \{?A \to R_4, ?N \to \textit{ringo}, ?E \to \textit{ringo@acd.edu}$,
$?W \to \textit{www.starr.edu}\}$.

Notice that we obtain no information for the Web page of george, since in P_2 that information is retrieved only for the resources that have an email.

Now consider the following expression:

$P_3 = \big(((?A, \textit{name}, ?N)\ \text{OPT}\ (?A, \textit{phone}, ?P))\ \text{FILTER}\ ?N = \textit{paul}\big)$

The evaluation of P_3 over G is the set of mappings $[\![P_2]\!]_G = \{\mu_1\}$, where $\mu_1 = \{?A \to R_1, ?N \to \textit{paul}, ?P \to \textit{777-3426}\}$.

2.3. Well-designed patterns

The patterns that mostly interests us are known as *well-designed patterns* (Pérez et al., 2009). A filter expression $(P\ \text{FILTER}\ R)$ is *safe* if every every variable $?X$ appearing in R also appears in P. A pattern P is said to be well-designed if:

(i) every filter expression in P is safe, and

(ii) for every sub pattern $P' = (P_1\ \text{OPT}\ P_2)$ of P and every variable $?X$ occurring in P, it holds that if $?X$ occurs inside P_2 and outside P', then $?X$ also occurs inside P_1.

Example 2.2 (From Pérez et al. (2009))**.** *Consider the SPARQL graph pattern*

$$P = \big((?X, \textit{name}, \textit{john})\ \text{OPT}\ ((?Y, \textit{name}, \textit{mick})\ \text{OPT}\ (?X, \textit{email}, ?Z))\big)$$

Notice how $(?X, \textit{email}, ?Z)$ is giving optional information for $(?X, \textit{name}, \textit{john})$, but in P appears giving optional information for $(?Y, \textit{name}, \textit{mick})$. Notice how graph G from Example 2.1 includes the triples $(R_2, \textit{name}, \textit{john})$ and $(R_2, \textit{email}, \textit{john@acd.edu})$, but the evaluation of $[\![P]\!]_G$ only returns the set $\{\{?X \to R_2\}\}$, since $[\![(?Y, \textit{name}, \textit{mick})]\!]_G = \emptyset$ (as mick was in a different band), without giving information for the email of john.

Patterns P_1, P_2 and P_3 from Example 2.1 are well-designed. Interestingly, all patterns in this class have been shown to have several desirable properties. In particular, Pérez et al. (2009) showed that the complexity of the evaluation problem for well-designed patterns is much lower than for the general language. Furthermore, that work showed several rewrite rules which were suggested to be useful for optimization procedures. This idea was taken further in Letelier et al. (2012b), where the authors created an algebra of graph patterns by introducing the notion of pattern trees, which is further discussed in Chapter 3.

Chapter 3. PATTERN TREES

This chapter is based heavily on previous work by Letelier et al. (2012b). In that paper, most of the definitions, propositions, lemmas and theorems presented here were present, but did not consider the existence of the FILTER operator. To maintain consistency and to make this work easier to understand to readers familiar with these concepts, we have expanded all of them while keeping our modified declarations as close as possible to the original ones. Those cases where the wording was taken directly from the previous work are appropriately marked.

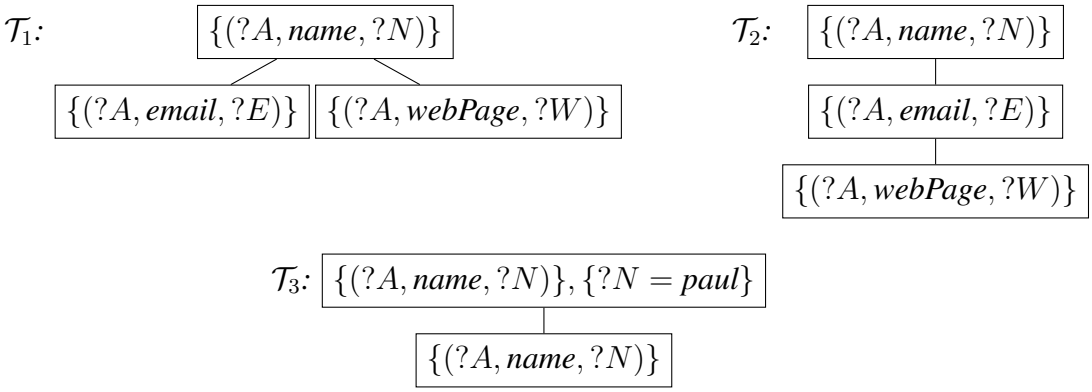
3.1. Definitions

A *rooted tree* is defined as a tuple $T = (V, E, r)$, where V is a set of nodes, E is a set of edges, and $r \in V$ is the root of the tree. We assume trees to be unordered and undirected. For any node n we write T_n to denote the subtree of T rooted at n, composed by all the descendants of n in the tree. With this definition we can now redefine the tree representation of SPARQL graph patterns by adding a FILTER label to each node.

Definition 3.1 (Pattern tree)**.** *A* pattern tree $\mathcal{T}$ *is a triple* $\mathcal{T} = (T, \mathcal{P}, \mathcal{F})$*, where* $T = (V, E, r)$ *is a rooted tree,* $\mathcal{P} = \left(P_n\right)_{n \in V}$ *is a labeling of the nodes of* $\mathcal{T}$ *such that* P_n *is a nonempty set of triple patterns, and* $\mathcal{F} = \left(F_n\right)_{n \in V}$ *is a labeling of the nodes of* $\mathcal{T}$ *such that* F_n *is a set of SPARQL built-in conditions for every* $n \in V$*.*

Notice how a pattern tree can be represented by a graphical tree with its nodes labeled appropriately. In the case of nodes with an empty set of built-in conditions, we simply omit that label. This is illustrated in Example 3.1.

Example 3.1. *The following are representations of the pattern trees which intuitively correspond to the queries introduced in Example 2.1.*



We will now establish a syntactic relationship between pattern trees and SPARQL graph patterns. Given a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ and a node $n \in V$, we denote by $\mathrm{vars}(P_n)$ the set of variables mentioned in the triples of P_n, by $\mathrm{vars}(F_n)$ the set of variables mentioned in the built-in conditions of F_n, and by $\mathrm{vars}(\mathcal{T})$ the set $\bigcup_{n \in V}(\mathrm{vars}(P_n) \cup \mathrm{vars}(F_n))$. We can now redefine the transformation function $\textsc{Tr}(\cdot, \cdot, \cdot)$, which is used to transform a pattern tree into a SPARQL graph pattern.

Consider a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ and a set Σ of ordering functions $\{\sigma_n | n \in V\}$, such that $\sigma_n(1)$ is the first node in the ordering, $\sigma_n(2)$ is the second one, and so on. Also, given sets $P = \{t_1, t_2, \ldots, t_k\}$ and $F = \{R_1, R_2, \ldots, R_\ell\}$, let $and(P)$ be the SPARQL pattern $(t_1 \text{ AND } t_2 \text{ AND } \ldots \text{ AND } t_k)$ and $conj(F)$ be the built-in condition $(R_1 \wedge R_2 \wedge \ldots \wedge R_\ell)$. Finally, let $\textsc{tr}(P, F) = and(P)$ if F is empty, or $(and(P) \text{ FILTER } conj(F))$ otherwise. We can now define the transformation of the sub tree of $\mathcal{T}$ rooted at n given the order Σ. Assuming that n has k children in $\mathcal{T}$, then $\textsc{Tr}(\mathcal{T}, n, \Sigma)$ is defined as the graph pattern expression

$$\big(\cdots \big(\big(\textsc{tr}(P_n, F_n) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n(1), \Sigma)\big) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n(2), \Sigma)\big) \cdots \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n(k), \Sigma)\big),$$

and if n has no children, then $\textsc{Tr}(\mathcal{T}, n, \Sigma) = \textsc{tr}(P_n, F_n)$. Finally, given a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ and an ordering Σ for $\mathcal{T}$, we define $\textsc{Tr}(\mathcal{T}, \Sigma)$ as $\textsc{Tr}(\mathcal{T}, r, \Sigma)$.

Example 3.2. *Applying* $\textsc{Tr}(\cdot,\cdot,\cdot)$ *to the pattern trees in Example 3.1, ordering sibling nodes from left to right, we get the following SPARQL graph patterns:*

$$\begin{aligned}
\textsc{Tr}(\mathcal{T}_1,\Sigma_1) &= \big(\big((?A, \textit{name}, ?N)\ \text{OPT}\ (?A, \textit{email}, ?E)\big)\ \text{OPT}\ (?A, \textit{webPage}, ?W)\big)\\
\textsc{Tr}(\mathcal{T}_2,\Sigma_2) &= \big((?A, \textit{name}, ?N)\ \text{OPT}\ \big((?A, \textit{email}, ?E)\ \text{OPT}\ (?A, \textit{webPage}, ?W)\big)\big)\\
\textsc{Tr}(\mathcal{T}_3,\Sigma_3) &= \big(((?A, \textit{name}, ?N)\ \text{FILTER}\ ?N = \textit{paul})\ \text{OPT}\ (?A, \textit{phone}, ?P)\big)
\end{aligned}$$

Notice how $\textsc{Tr}(\mathcal{T}_1,\Sigma_1) = P_1$ *and* $\textsc{Tr}(\mathcal{T}_2,\Sigma_2) = P_2$*, but* $\textsc{Tr}(\mathcal{T}_3,\Sigma_3) \neq P_3$*. However, since* P_3 *is well-designed, it can easily be shown that* $\textsc{Tr}(\mathcal{T}_1,\Sigma_1)$ *is equivalent to* P_3 *(Pérez et al., 2009).*

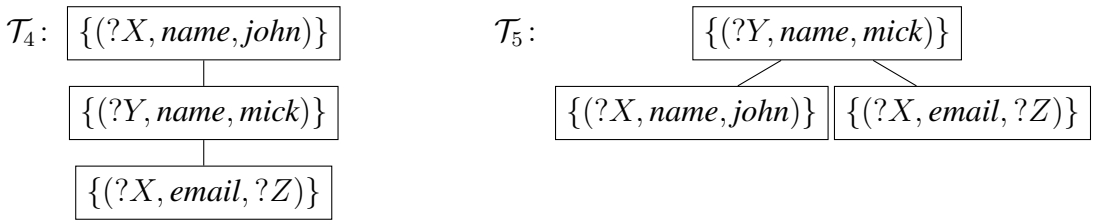
3.2. Semantics of well-designed pattern trees

Having established a syntactic relationship between pattern trees and SPARQL graph patterns, we are interested in defining the evaluation of a pattern tree over an RDF graph. Notice that several different SPARQL patterns can be obtained from a pattern tree depending on the ordering functions used in the transformation. Thus, we cannot directly define the evaluation of a pattern tree $\mathcal{T}$ by using the evaluation of an arbitrary transformation of $\mathcal{T}$.

In this section we extend the Definition of the well-designed condition for pattern trees, which will help define the semantics of pattern trees. In particular, it will allow us to choose an arbitrary transformation of a pattern tree in order to evaluate it. We begin with the definition of the well-designedness condition for pattern trees. Given a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n\in V}, (F_n)_{n\in V})$, we say that a node n is *safe* if $\text{vars}(F_n) \subseteq \text{vars}(P_n)$. Notice that if this condition is not satisfied, then by the semantics of SPARQL defined in Section 2.2 we have that, for any graph G, $[\![P_n\ \text{FILTER}\ F_n]\!]_G = \emptyset$. Thus, in accordance with the definition of well-designed SPARQL patterns given in Pérez et al. (2009), we add this safety condition over every node in the tree to the previous definition of a well-designed pattern tree.

Definition 3.2. *A pattern tree* $\mathcal{T} = ((V, E, r), (P_n)_{n\in V}, (F_n)_{n\in V})$ *is* well-designed *if for every* $n \in V$, $\mathrm{vars}(F_n) \subseteq \mathrm{vars}(P_n)$, *and for every variable* $?X$ *occurring in* $\mathcal{T}$, *the set* $\{n \in V | ?X \in \mathrm{vars}(P_n)\}$ *induces a connected subtree of* $\mathcal{T}$.

Example 3.3. *The pattern trees from Example 3.1,* $\mathcal{T}_1$, $\mathcal{T}_2$ *and* $\mathcal{T}_3$, *are all well-designed. The following pattern trees are not:*



$\mathcal{T}_6$: $\{(?X, name, ?N)\}, \{?M = john\}$

Tree $\mathcal{T}_4$ *is based on the SPARQL graph pattern* P *from example 2.2 Note how variable* $?X$ *induces a disconnected subgraph in both* $\mathcal{T}_4$ *and* $\mathcal{T}_5$, *while using variable* $?M$ *instead of* $?N$ *in the built-in condition in* $\mathcal{T}_6$ *makes the query not safe.*

The similarity between the well-designedness condition for pattern trees and for graph patterns makes the following proposition still hold. However, despite the wording being the same, the proof given in Letelier et al. (2012b) no longer holds due to the extended definitions, and is completed in Appendix A.1

PROPOSITION 3.1 (From Letelier et al. (2012b)). *Let* $\mathcal{T}$ *be a well-designed pattern tree, and* Σ *an arbitrary set of ordering functions for* $\mathcal{T}$. *Then* $\mathrm{TR}(\mathcal{T}, \Sigma)$ *is a well-designed graph pattern.*

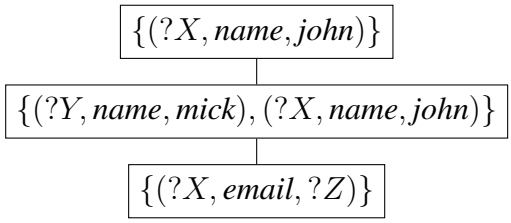
We now move onto the relaxation of the well-designedness condition.

Definition 3.3. *A pattern tree* $\mathcal{T} = ((V, E, r), (P_n)_{n\in V}, (F_n)_{n\in V})$ *is a* quasi well-designed pattern tree *(shortened to QWDPT) if for every node* $n \in V$ *it is the case that* $\mathrm{vars}(F_n) \subseteq \mathrm{vars}(P_n)$, *and for every pair of nodes* $u, v \in V$ *and each variable* $?X \in \mathrm{vars}(P_u) \cap \mathrm{vars}(P_v)$ *there exists a node* n *that is a common ancestor of* u *and* v *in* $\mathcal{T}$, *such that* $?X \in \mathrm{vars}(P_n)$.

Once again, we have merely added a safety condition in every node. The work from Letelier et al. (2012b) depended heavily on the notion of QWDPTs to obtain several of its results; however, since we are mostly focused on using pattern trees to rewrite queries and not to test for containment or equivalence, this definition is mostly included for the sake of completeness.

A QWDPT can be turned into a well-designed pattern tree by duplicating triples from some nodes down into their descendants, so that every variable in the tree ends up inducing a connected subtree, thus satisfying Definition 3.2.

Example 3.4. *Consider the tree $\mathcal{T}_4$ from Example 3.3. The tree can be transformed into a well-designed pattern tree by copying the triple $(?X, \textit{name}, \textit{john})$ into the root's child node:*



However, neither $\mathcal{T}_5$ nor $\mathcal{T}_6$ can be turned into a well-designed pattern tree, since they are not quasi well-designed.

We say that a pattern tree $\mathcal{T}' = ((V', E', r'), (P'_n)_{n \in V'}, (F'_n)_{n \in V'})$ was derived from a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ by duplicating a triple to a child, denoted by $\mathcal{T} \hookrightarrow \mathcal{T}'$, if $(V', E', r') = (V, E, r)$ (that is, the underlying trees are the same), and there exist a node $u \in V$, a triple $t \in P_u$, and a child v of u, such that $P'_v = P_v \cup \{t\}$, and $P_n = P'_n$ for all $n \neq v$. We denote by $\hookrightarrow^*$ the reflexive and transitive closure of $\hookrightarrow$, that is, $\mathcal{T} \hookrightarrow^* \mathcal{T}'$ if $\mathcal{T} = \mathcal{T}'$ or there exists a sequence $\mathcal{T}_1 \hookrightarrow \mathcal{T}_2 \hookrightarrow \ldots \hookrightarrow \mathcal{T}_m$ with $\mathcal{T}_1 = \mathcal{T}$ and $\mathcal{T}_m = \mathcal{T}'$. This leads us into the next definition, taken verbatim from Letelier et al. (2012b).

Definition 3.4 (From Letelier et al. (2012b)). *Let $\mathcal{T}$ be a QWDPT. The set of SPARQL graph patterns defined by $\mathcal{T}$ is defined as*

$$\textsc{Sem}(\mathcal{T}) = \{\textsc{Tr}(\mathcal{T}', \Sigma) \mid \Sigma \textit{ is an ordering for } \mathcal{T}', \mathcal{T} \hookrightarrow^* \mathcal{T}' \textit{ and } \mathcal{T}' \textit{ is well-designed}\}.$$

To define the result of evaluating a QWDPT $\mathcal{T}$ over an RDF graph G, we first show that all queries in $\textsc{Sem}(\mathcal{T})$ are equivalent. Using this property, we then define the evaluation of $\mathcal{T}$ to be exactly the same as that of an arbitrarily chosen query from $\textsc{Sem}(\mathcal{T})$. Once again, the following lemmas and theorem are taken straight from Letelier et al. (2012b), but due to the extended definitions the proofs no longer apply, and are completed in Appendixes A.2 and A.3.

Lemma 3.1. *Let $\mathcal{T}$ be a well-designed pattern tree, Σ_1 and Σ_2 be two arbitrary orderings for $\mathcal{T}$, and $P_1 = \textsc{Tr}(\mathcal{T}, \Sigma_1)$ and $P_2 = \textsc{Tr}(\mathcal{T}, \Sigma_2)$ be the graph patterns obtained by transforming $\mathcal{T}$ with Σ_1 and Σ_2, respectively. Then $P_1 \equiv P_2$.*

Lemma 3.2. *Let $\mathcal{T}$ be a QWDPT, let Σ be an ordering for $\mathcal{T}$, and let $\mathcal{T}_1$ and $\mathcal{T}_2$ be well-designed pattern trees such that $\mathcal{T} \hookrightarrow^* \mathcal{T}_1$ and $\mathcal{T} \hookrightarrow^* \mathcal{T}_2$. If $P_1 = \textsc{Tr}(\mathcal{T}_1, \Sigma)$ and $P_2 = \textsc{Tr}(\mathcal{T}_2, \Sigma)$, then $P_1 \equiv P_2$.*

Putting these two Lemmas together, we get the following result:

Theorem 3.1. *Let $\mathcal{T}$ be a QWDPT. Then all graph patterns in $\textsc{Sem}(\mathcal{T})$ are equivalent, i.e., for any two graph patterns $P_1, P_2 \in \textsc{Sem}(\mathcal{T})$, it holds that $P_1 \equiv P_2$.*

The proof follows directly from Lemma 3.1 and 3.2. This finally allows us to define the semantics of a QWDPT via an arbitrary triple pattern from $\textsc{Sem}(\mathcal{T})$.

Definition 3.5. *Let $\mathcal{T}$ be a QWDPT and G an RDF graph. Then the evaluation of $\mathcal{T}$ over G, denoted by $[\![\mathcal{T}]\!]_G$, is defined as the set of mappings $[\![P]\!]_G$ for an arbitrary $P \in \textsc{Sem}(\mathcal{T})$.*

Theorem 3.1 allows us to choose any representative in $\text{SEM}(\mathcal{T})$ for a given QWDPT $\mathcal{T}$ and evaluate it. In particular, if $\mathcal{T}$ is already well-designed, we may simply fix the order of the child nodes of each node and evaluate this SPARQL pattern.

Given two QWDPTs $\mathcal{T}_1$ and $\mathcal{T}_2$, we say that $\mathcal{T}_1$ and $\mathcal{T}_2$ are equivalent, denoted by $\mathcal{T}_1 \equiv \mathcal{T}_2$, if for every RDF graph G it holds that $[\![\mathcal{T}_1]\!]_G = [\![\mathcal{T}_2]\!]_G$. Similarly, a QWDPT $\mathcal{T}$ is equivalent to a SPARQL graph pattern P, denoted by $\mathcal{T} \equiv P$, if for every RDF graph G it holds that $[\![\mathcal{T}]\!]_G = [\![P]\!]_G$. Notice that Definition 3.5 plus Proposition 3.1 imply that for every QWDPT $\mathcal{T}$ there exists a well-designed graph pattern P such that $\mathcal{T} \equiv P$. The last result of this section states that the opposite also holds, and thus, QWDPTs can represent the entire class of well-designed SPARQL graph patterns.

We end this section by extending the following proposition and its proof, which show how to construct a well-designed pattern tree from a SPARQL graph pattern.

PROPOSITION 3.2 (From Letelier et al. (2012b)). *For every well-designed graph pattern P, there exists a QWDPT $\mathcal{T}$ such that $P \equiv \mathcal{T}$. Moreover, given a well-designed graph pattern, an equivalent QWDPT can be constructed in polynomial time.*

PROOF. In Pérez et al. (2009) it was shown that every well-designed graph pattern is equivalent to a pattern in OPT-normal form, which is defined as follows:

A pattern P is in OPT-normal form if:

(i) P is constructed by using only the AND and FILTER operators, or

(ii) $P = (P_1 \ \text{OPT} \ P_2)$, where P_1 and P_2 are in OPT-normal form.

Given a pattern P in OPT-normal form we describe an algorithm to construct a well-designed pattern tree.

If $P = (t_1 \ \text{AND} \ t_2 \ \text{AND} \ \cdots \ \text{AND} \ t_k \ \text{FILTER} \ R)$ then we create a pattern tree with a single node and labels $\{t_1, \ldots, t_k\}$ and $\{R\}$ respectively.

If $P = (P_1 \ \text{OPT} \ P_2)$ then we construct a pattern tree $\mathcal{T}_1$ from P_1, a pattern tree $\mathcal{T}_2$ from P_2 and then construct a pattern tree $\mathcal{T}$ from $\mathcal{T}_1$ and $\mathcal{T}_2$, by considering $\mathcal{T}_1$ and $\mathcal{T}_2$

together, adding the root of $\mathcal{T}_2$ as a child of the root of $\mathcal{T}_1$, and setting the root of $\mathcal{T}_1$ as the root of the obtained tree $\mathcal{T}$. It is not difficult to show that the obtained pattern tree $\mathcal{T}$ is well-designed and that there exists an ordering Σ for $\mathcal{T}$ such that $\text{TR}(\mathcal{T}, \Sigma) = P$ and thus, $\mathcal{T} \equiv P$. □

3.3. Evaluating pattern trees

In this section we extend the procedural semantics for QWDPTs given in Letelier et al. (2012b).

We say that a mapping μ_1 is *subsumed by* μ_2, denoted by $\mu_1 \sqsubseteq \mu_2$, if $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ and for every $?X \in \text{dom}(\mu_1)$ it holds that $\mu_1(?X) = \mu_2(?X)$. We write $\mu_1 \sqsubset \mu_2$ whenever $\mu_1 \sqsubseteq \mu_2$ and $\mu_1 \neq \mu_2$. Also, given a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ we denote by $\text{redux}(\mathcal{T})$ the conjunction (AND) of the transformation through TR of all the nodes in V. For example, if $V = \{n_1, \cdots, n_\ell\}$, then

$$\text{redux}(\mathcal{T}) = \Big(\Big(\text{TR}(P_{n_1}, F_{n_1})\Big) \text{ AND } \cdots \text{ AND } \Big(\text{TR}(P_{n_\ell}, F_{n_\ell})\Big)\Big).$$

Note that this is equivalent to transforming the pattern tree into a SPARQL pattern and changing all OPT operators to AND operators.

Lemma 3.3 (From Letelier et al. (2012b))**.** *Let $\mathcal{T}$ be a QWDPT with root r, and G an RDF graph. A mapping μ is in $[\![\mathcal{T}]\!]_G$ if and only if*

(i) *$\mu \in [\![\text{redux}(\mathcal{T}')]\!]_G$ for a subtree $\mathcal{T}'$ of $\mathcal{T}$ rooted at r, and*

(ii) *there are no mapping ν and subtree $\mathcal{T}''$ of $\mathcal{T}$ rooted at r, such that $\mu \sqsubset \nu$ and $\nu \in [\![\text{redux}(\mathcal{T}'')]\!]_G$.*

Basically, Lemma 3.3 states that any mapping μ in the evaluation of a QWDPT $\mathcal{T}$ must be *maximal* (with respect to $\sqsubseteq$) for some subtree $\mathcal{T}'$ of $\mathcal{T}$, such that it cannot be further extended by another subtree $\mathcal{T}''$ of $\mathcal{T}$. The proof given for this lemma in Letelier

et al. (in press) is based on Proposition 4.5 from Pérez et al. (2009), which applies to well-designed graph patterns with FILTER. Therefore, this is the only case in which the exact same proof applies without having to be modified.

This gives way to the following algorithm for evaluating QWDPTs, called the *Top-down* evaluation. This definition is the basis for the Top-down engine used in Chapter 4.

Definition 3.6 (Top-down evaluation, from Letelier et al. (2012b))**.** *Consider an RDF graph G, a QWDPT $\mathcal{T} = ((V, E, r), (P_n)_{n \in V})$, and a set M of mappings. For $n \in V$, we define the evaluation of $\mathcal{T}_n$ (the complete subtree of $\mathcal{T}$ rooted at n) given M over G, denoted by* $\mathrm{ext}(M, n, G)$ *as follows. If n is a leaf, then*

$$\mathrm{ext}(M, n, G) = M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G,$$

and, otherwise, if $n_1, \ldots, n_k$ are the child nodes of n, then

$$\mathrm{ext}(M, n, G) = M_1 \bowtie M_2 \bowtie \cdots \bowtie M_k,$$

where $M_i = (M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G) \rtimes \mathrm{ext}(M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G, n_i, G)$. We define the top-down evaluation *of $\mathcal{T}$ over G, denoted by $[\![\mathcal{T}]\!]^{td}_G$, as*

$$[\![\mathcal{T}]\!]^{td}_G = \mathrm{ext}(\{\mu_\emptyset\}, r, G),$$

where $\mu_\emptyset$ is the mapping with the empty domain.

Theorem 3.2 (From Letelier et al. (2012b))**.** *Let $\mathcal{T}$ be a QWDPT and G an RDF graph. Then $[\![\mathcal{T}]\!]_G = [\![\mathcal{T}]\!]^{td}_G$.*

3.4. Transformation of QWDPTs

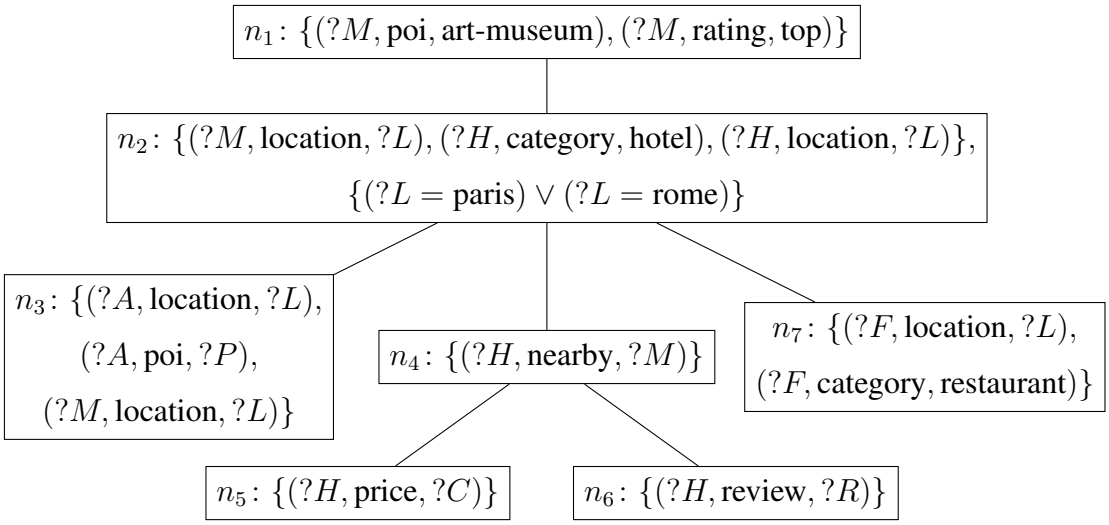
We now proceed to extend the transformation rules provided in (Letelier et al., 2012b) for pattern trees. As noted in that paper, one of the advantages of QWDPTs is that they allow us to define equivalence preserving transformation rules based on the structure of

the tree, while previous work has based these transformations on the properties of the SPARQL operators (Pérez et al., 2009; Schmidt et al., 2010).

Before presenting the transformation rules, we need to introduce some additional notation. Let $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ be a pattern tree, and n a node in V. We define the *branch of* n *in* $\mathcal{T}$, denoted by $\mathrm{branch}(n, \mathcal{T})$, as the unique path from r to n, given as the sequence of nodes $n^1, \ldots, n^k$ with $n^1 = r$ and $n^k = n$. If it is clear from the context, we may drop the name of the pattern tree and simply write $\mathrm{branch}(n)$. We denote by $P_{\mathrm{branch}(n,\mathcal{T})}$ the set of triple patterns $\bigcup_{i=1}^{k} P_{n^i}$. Given two sets P_1 and P_2 of triple patterns, a *homomorphism* h from P_1 into P_2, written $h\colon P_1 \to P_2$, is a mapping $h\colon \mathrm{vars}(P_1) \to \mathbf{U} \times \mathbf{V}$ s.t. for all triple patterns $t \in P_1$ it holds that $h(t) \in P_2$, where $h(t)$ denotes the triple obtained from t by replacing all variables $?X \in \mathrm{vars}(t)$ by $h(?X)$ and leaving URIs unchanged. It is further convenient to introduce the following notation to speak about variables occurring in some P_n.

Definition 3.7. *Let $\mathcal{T} = ((V, E, r), \mathcal{P})$ be a pattern tree and $n, \hat{n} \in V$ s.t. $\hat{n}$ is the parent node of n. Then the* new variables *at n are defined as* $\mathrm{newvars}(n) = \mathrm{vars}(P_n) \setminus \mathrm{vars}(P_{\mathrm{branch}(\hat{n})})$. *For the case of the root r, we define* $\mathrm{newvars}(r)$ *as* $\mathrm{vars}(P_r)$.

Along with the rules, we will present a running example, which is a modified version of the one presented in Letelier et al. (in press). Consider the following well-designed pattern tree $\mathcal{T}$:

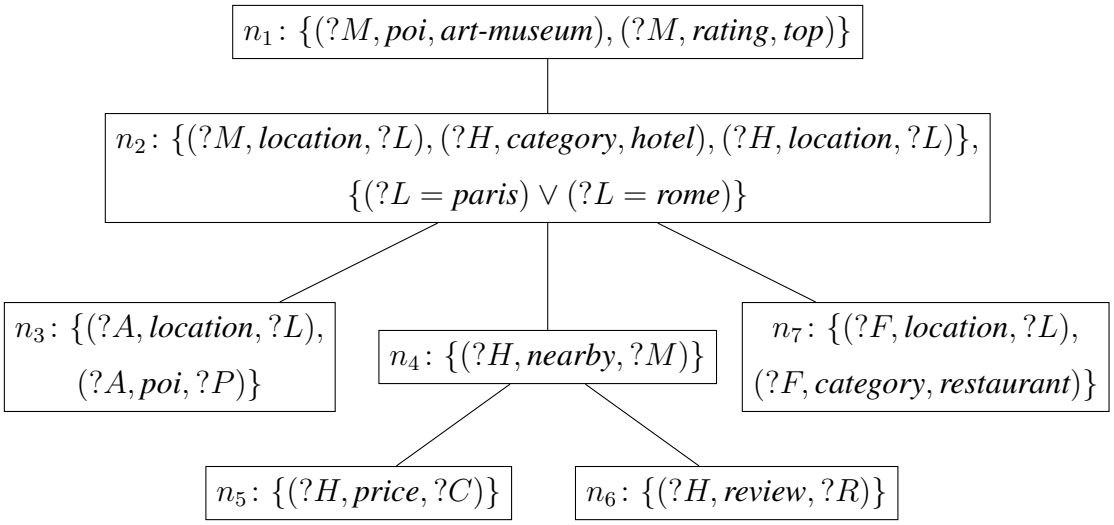


Node n_1 specifies that the user is searching for art museums (assigned to variable $?M$) with a good rating. Node n_2 looks for hotels located near each museum, but only for those museums located in Paris or Rome. Node n_3 requests additional points of interest in the same location as the hotel. Node n_4 tells the user if the hotel is close to the museum, and if so, requests the price and the review for the hotel in nodes n_5 and n_6. Finally, node n_7 searches for restaurants in the same city as the hotel.

Rule R1 (*deletion of redundant triples*): Let $n \in V$. If there exists a triple $t \in P_n$ s.t. $\text{vars}(t) \cap \text{vars}(F_n) = \emptyset$ and $t \in P_{n'}$ for some ancestor n' of n, then delete t from P_n, i.e. $P'_n = P_n \setminus \{t\}$. If $P'_n = \emptyset$, delete n and turn its child nodes into children of the parent of n.

We have modified rule R1 to ensure that no node can become unsafe because of it, and thus it cannot break the quasi well-designedness of the tree.

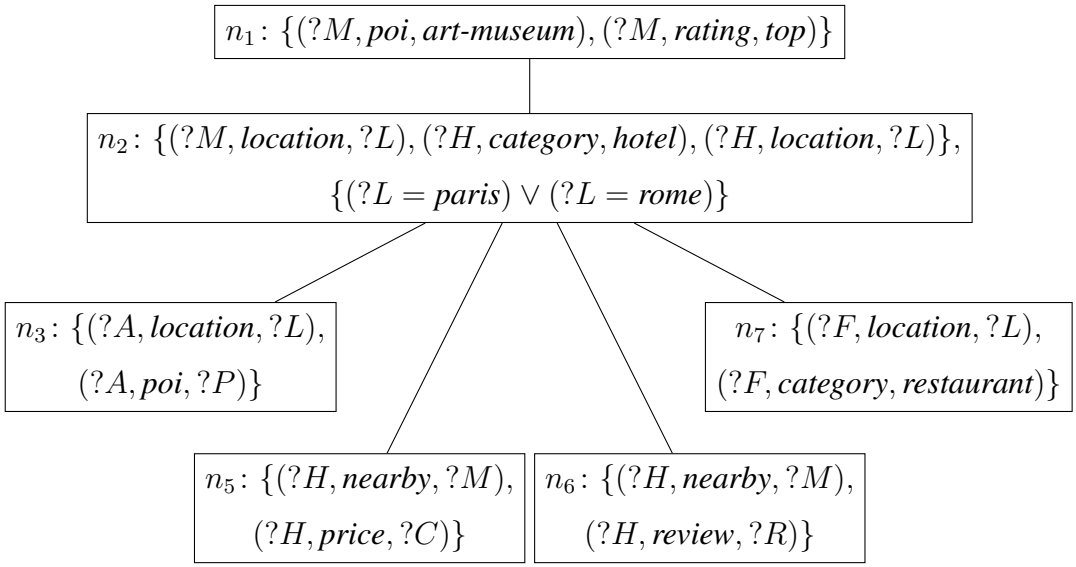
Example 3.5. *Consider the QWDPT $\mathcal{T}$. Since the triple $(?M, \text{location}, ?L)$ in node n_3 also appears in node n_2, we can safely eliminate it to obtain the following pattern tree.*



Rule R2 (*deletion of unproductive nodes*): Let $n, \hat{n} \in V$ s.t. $\hat{n}$ is the parent of n, and let $n_1, \ldots, n_k \in V$ be the children of n. If $\mathrm{newvars}(n) = \emptyset$, then merge n into each of its children and make each n_i a child of $\hat{n}$. I.e. let $P'_{n_i} = P_{n_i} \cup P_n$ and $F'_{n_i} = F_{n_i} \cup F_n$ for $i = \{1, \ldots, k\}$, $V' = V \setminus \{n\}$, and $E' = (E \setminus \{(\hat{n}, n), (n, n_1), \ldots, (n, n_k)\}) \cup \{(\hat{n}, n_1), \ldots, (\hat{n}, n_k)\}$. If n has no child node, then applying this rule is equivalent to deleting n.

With rule R1 we added a restriction to its use. However, we have given rule R2 a new effect: when merging a node downwards onto its children, the built-in conditions are passed along with the triple patterns.

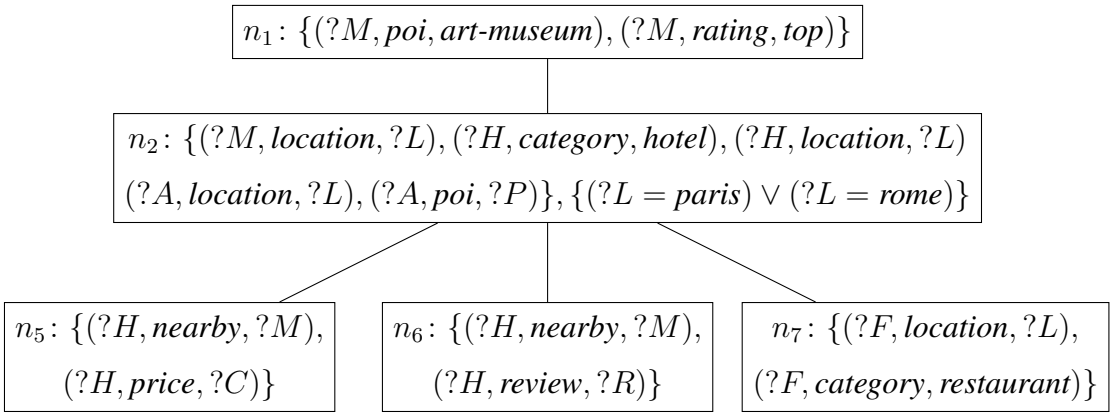
Example 3.6. *Consider the resulting tree from Example 3.5. Note how node n_4 does not introduce any new variables, since $?H$ and $?M$ are both present in node n_2. Thus, we can copy the contents of P_{n_4} into P_{n_5}, eliminate node n_4 and make nodes n_5 and n_6 children of n_2:*



Rule R3 (*homomorphism upwards*): Let $n, \hat{n} \in V$ be nodes s.t. $\hat{n}$ is the parent of n, and let $n_1, \ldots, n_k \in V$ be the children of n. If there exists a homomorphism $h\colon P_n \to P_{\text{branch}(\hat{n})}$ with $h(?X) = ?X$ for all variables $?X \in \text{vars}(P_n) \cap \text{vars}(P_{\text{branch}(\hat{n})})$, and $\text{vars}(F_n) \cap \text{vars}(\text{branch}(\hat{n})) = \emptyset$, then *merge* n into $\hat{n}$, i.e. let $P'_{\hat{n}} = P_{\hat{n}} \cup P_n$, $F'_{\hat{n}} = F_{\hat{n}} \cup F_n$, $V' = V \setminus \{n\}$ (remove n) and $E' = (E \setminus \{(\hat{n}, n), (n, n_1), \ldots, (n, n_k)\}) \cup \{(\hat{n}, n_1), \ldots, (\hat{n}, n_k)\}$ (turn n's child nodes into children of $\hat{n}$).

Rule R3 has both a new restriction to its application and a new effect: since merging the node upwards now copies its set of built-in conditions upwards, care must be taken that this does not affect the rest of the tree.

Example 3.7. *Consider node n_3 in the resulting tree from Example 3.6. Now consider the homomorphism $h\colon \{?A \to ?M, ?L \to ?L, ?P \to \textit{art-museum}\}$, which maps each of the triples in n_3 onto a triple in n_1 or n_2. Also note that n_3 has no built-in conditions, and so the additional condition for rule R3 is trivially satisfied. Therefore, we can merge n_3 into n_2 to obtain the following tree:*



Rule R4 (*parallelization*): Consider nodes $\hat{n}, n, n' \in V$ s.t. $\hat{n}$ is the parent of n, and n is the parent of n'. If there exists a homomorphism $h\colon P_n \to P_{n'} \cup P_{\text{branch}(\hat{n})}$ with $h(?X) = ?X$ for all variables $?X \in \text{vars}(P_n) \cap \text{vars}(P_{\text{branch}(\hat{n})})$, then add F_n to $F_{n'}$ (i.e. make $F'_{n'} = F_{n'} \cup F_n$) and turn n' from a child of n into a child of $\hat{n}$, if the resulting pattern tree is quasi well-designed. I.e. $V' = V$, $E = (E \setminus \{(n, n')\}) \cup \{(\hat{n}, n')\}$, if $\mathcal{T}'$ is still quasi well-designed.

Our running example does not have a rule R4 application. In fact, very few real queries do, as will be seen in Chapter 4. However, this rule has been included for the sake of completeness. In this case, it is necessary to copy some built-in expressions to ensure that results obtained from the descendants of n' still get filtered appropriately.

For the fifth rule we require an additional definition.

Definition 3.8. *Consider a pattern tree node n, its labeling P_n and a built-in condition R. Let $R' = (c_1 \wedge \cdots \wedge c_k)$ be another built-in condition in conjunctive normal form that is equivalent to F. We define the* maximum propagation of R onto n*, written as* $\text{maxprop}(R, n)$*, as the set $\{c_i | i \in 1 \ldots k \wedge \text{vars}(c_i) \subseteq \text{vars}(P_n)\}$.*

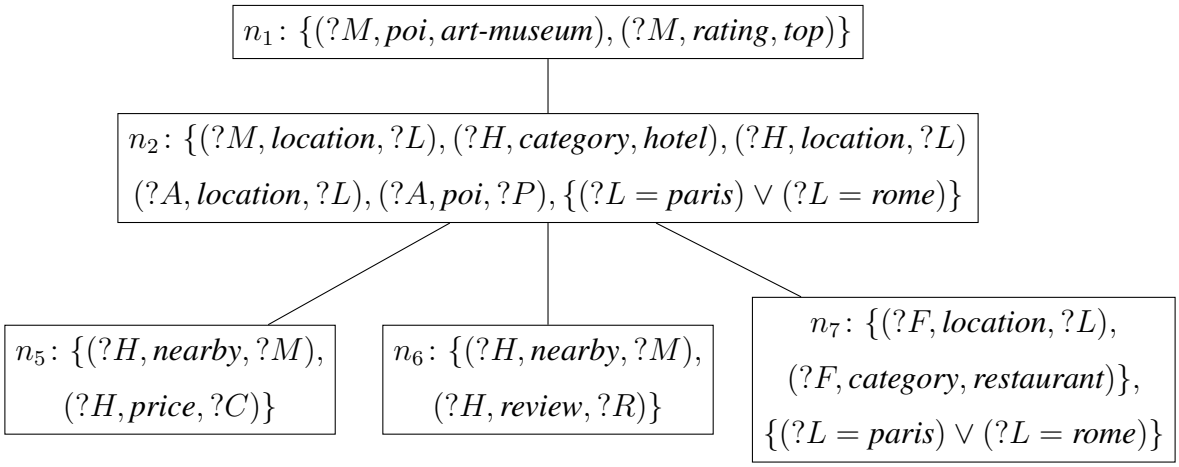
For a set $F = \{R_1, \ldots, R_\ell\}$ of built-in conditions, let M_i be the maximum propagation of R_i onto n. We define the maximum propagation of F onto n as the set $\bigcup_{i \in 1 \ldots \ell} M_i$.

We can now present our new transformation rule.

Rule R5 (*filter propagation*): Let $\hat{n}, n \in V$ be nodes s.t. $\hat{n}$ is the parent of n. Let M be the maximum propagation of $F_{\hat{n}}$ onto n. Then make $F'_n = F_n \cup M$.

Basically, rule R5 passes down as much filtering information as it can down the tree. Ideally, this allows lower nodes in the tree to filter out partial results which would end up being eliminated higher up in the tree after being joined with other partial results, speeding up evaluation time.

Example 3.8. *Consider the label in F_{n_2} and node n_7 in pattern tree $\mathcal{T}$ from Example 3.7. We can copy F_{n_2} into node n_7 to obtain our final well-designed pattern tree:*



Having presented the final rule, we can now introduce the following theorem.

Theorem 3.3. *Let $\mathcal{T}$ be a QWDPT and $\mathcal{T}'$ the pattern tree that results from applying either rule R1, or R2, or R3, or R4, or R5 to $\mathcal{T}$. Then $\mathcal{T}'$ is a QWDPT such that $\mathcal{T} \equiv \mathcal{T}'$.*

The proof for Theorem 3.3 is given in Appendix A.5, where the correctness of each rule is proven separately. This theorem allows us to freely transform any given QWDPT by applying rules R1 through R5 in any arbitrary order, which leads us into Chapter 4.

Chapter 4. QUERY OPTIMIZATION

We have thus far added the FILTER operator to the pattern tree structure and shown that QWDPTs still behave as one would intuitively expect (meaning that theorems and lemmas still hold). We now study the potential of these rules for query optimization.

Note that in this section we are working with real world SPARQL queries. There are a few syntactical differences between these and the mathematical constructs that we've used in the previous sections; for example, the real world version of the graph pattern "$(?S,\ ?P,\ ?O)$ AND $(?O,\ ?P,\ ?S)$" would be "$\{?S\ ?P\ ?O\ .\ ?O\ ?P\ ?S\}$". Care must be taken when going from one to the other, but in general the conversion is relatively straightforward.

4.1. Implementation

In Letelier et al. (2012b) and Letelier, Pérez, Pichler, and Skritek (2012a), the authors created a Java implementation of pattern trees, the *Top-down* evaluation algorithm, and the transformation rules R1 to R4 over the Jena framework and its ARQ main query engine (*Apache Jena*, 2013). We have updated this implementation to use Jena 2.10 and to include the FILTER operator on the Top-down evaluation, based on Definition 3.6, and transformation rules R1 through R4, as well as added rule R5.

The Jena-ARQ main query is a fairly modern and mainstream iterator based engine that has no cache. This makes it extremely useful for testing, as it prevents multiple run times of the same query from varying wildly as caches get warmed up, which makes isolating variables and repeating tests very easy and ensures consistent results. Also, unlike Virtuoso, 4-store and other engines, Jena-ARQ doesn't stop returning results after any given time or result limit is hit. Both Virtuoso and 4-store fail silently whenever this happens, thus making query evaluation times seem shorter than they really are. This makes them unsuitable for testing purposes.

On the other hand, the Top-down query engine is based on Jena-ARQ's reference engine. Instead of quickly returning an iterator and calculating new results as requested, it follows strictly the mathematical definitions of operators, completely calculating sets before doing operations on them, and thus returning the entire result set at once. It is designed to be correct, not efficient. Because of this, comparisons between both query engines are unfair, but nevertheless might be interesting. Furthermore, since rule applications were designed with the top down evaluation in mind, this engine should show the most improvement when evaluating rewritten queries.

4.2. Dataset

The USEWOD 2013 Data Challenge (Berendt, Hollink, Luczak-Rösch, Möller, & Vallet, 2013) provides a large dataset of several years worth of server logs from DBpedia,[1] Semantic Web Dog Food[2] (SWDF), Linked Open Geo Data[3] and Bio2RDF.[4]

In all, the logs contain almost 45 million query requests. Out of these, approximately 5 million are not valid SPARQL queries. For our test case, we require SELECT queries that contain at least one OPT operator, no UNION operator, no sub queries, and projection only on the top level (the SELECT statement). We call these queries *usable*.

Out of all the usable queries, we can only work on those that are well-designed. Out of these, we extract those over which we can apply at least one rule based transformation. We call these queries *transformable*. Finally, since these are real query logs, there are many identical queries, or queries that differ only in the parameters given to the LIMIT and OFFSET operators. This is due to the fact that most real world query engines limit the number of results they return, so users wanting to get the full result set must spread their operation over many queries. Therefore, we consider these queries identical and eliminate repeated queries.

[1] http://www.dbpedia.org/
[2] http://data.semanticweb.org/
[3] http://linkedgeodata.org/About
[4] http://bio2rdf.org/

	Bio2RDF	DBpedia	LGD	SWDF
Total	192,057	28,929,586	1,929,671	13,891,099
Valid	192,008	24,263,214	1,519,725	13,675,469
Usable	0	2,625,104	435,771	300,135
Well-designed	0	1,587,554	45,346	261,985
Transformable	0	23,454	14,025	30,050
Unique queries	0	6,690	151	117

TABLE 4.1. Number of queries per dataset

Each step eliminates a considerable portion of the available set of queries. The applicability of rules varies greatly over different datasets, leaving very few test cases in all of them except for DBpedia. The number of queries after each step is summarized in Table 4.1. We have therefore chosen DBpedia as our main test dataset, leaving us with 6,695 queries which can be transformed and compared. However, for the sake of completeness, we have also run the queries from Linked Open Geo Data and Semantic Web Dog Food.

4.3. Methodology

We first extract the graph pattern part of the query and construct its canonical pattern tree. We then exhaustively try to apply rules R1, R2, R3, R4 and R5 in that order. Finally, we run four tests: the original query over the Jena-ARQ engine, the rewritten query over the Jena-ARQ engine, the original query's tree pattern over the Top-down engine, and the modified tree pattern over the Top-down engine.

For the dataset, we have loaded the complete English version of DBpedia 3.8,[5] SWDF[6] and LGD[7] onto an instance of Virtuoso, which constitutes approximately 600 million unique triples. Ideally we would run each query over the entire dataset, but memory constraints make this impractical. To work around this, we take advantage of SPARQL's CONSTRUCT operator in order to create a smaller, tractable dataset.

[5] http://downloads.dbpedia.org/3.8/en/
[6] http://data.semanticweb.org/dumps/
[7] http://downloads.linkedgeodata.org/releases/

Given a SPARQL SELECT query Q, we refer by $graphPattern(Q)$ to the graph pattern of Q. Given a graph pattern P, we refer by $triples(P)$ to the set of all triple patterns mentioned in P. We can then use a number n and Algorithm 1 to build a CONSTRUCT query which, when evaluated over a large database D, generates a second database D_Q. Intuitively, D_Q includes only those triples in D that are useful for answering Q, in the sense that if we were to remove them from D there would be at least one less result in $[\![Q]\!]_D$.

Assuming that n is large enough and that Q is well-designed, evaluating Q over D_Q will create the same results as when evaluating Q over D. As long as n is larger than the number of triples in Q and that $[\![Q]\!]_D$ is not empty, then $[\![Q]\!]_{D_Q}$ will include at least one result. Also note that since $D_Q \subseteq D$ and Q is well-designed, $[\![Q]\!]_{D_Q} \subseteq [\![Q]\!]_D$. Finally, note that the inclusion of the LIMIT statement keeps the size of D_Q manageable, while including the pattern $?S\ ?P\ ?O$ in the template of the CONSTRUCT query and the UNION statement ensures that D_Q is large enough to make the calculation of $[\![Q]\!]_{D_Q}$ nontrivial for most queries.

Algorithm 1: Construct(Q, $size$)

Input: Q: query, $size$: number
$P \leftarrow graphPattern(Q)$;
$out \leftarrow$ "CONSTRUCT{";
foreach t *in* $triples(P)$ **do**
  $out \leftarrow$ `append(`*out, t*`)` ;
end
out ← `append(`*out, "?S ?P ?O }"*`)`;
out ← `append(`*out,* "WHERE *{ {"*`)` ;
out ← `append(`*out, P*`)`;
out ← `append(`*out, "}* UNION *{ ?S ?P ?O } } LIMIT "*`)`;
out ← `append(`*out, size*`)`;
return *out* ;

Thus, our testing for each unique, useful and transformable query Q proceeds as follows:

(i) Parse the query Q;

(ii) using Virtuoso, evaluate Construct(Q, 10000) over DBpedia, LGD or SWDF to create database D_Q;

(iii) generate Q's canonical pattern tree, $\mathcal{T}$, by transforming the body of Q as shown in Proposition 3.2;

(iv) exhaustively apply every possible transformation rule to $\mathcal{T}$ to create $\mathcal{T}'$;

(v) create a query Q' by transforming $\mathcal{T}'$ into a SPARQL query and applying all result modifiers from the head of Q;

(vi) evaluate Q and Q' over D_Q using Jena's ARQ main query engine;

(vii) evaluate $\mathcal{T}$ and $\mathcal{T}'$ over D_Q using the Top-down evaluation from Letelier et al. (2012b).

The queries were run on an Amazon EC2 general purpose m3.xlarge instance. The machine has 15 GB of RAM and a 64-bit, 4-core processor with 13 ECU units,[8] with one EC2 Compute Unit being the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.[9] For each evaluation there was a time limit of 120,000 milliseconds. Query execution time was measured by reading the system clock before and after the execution. Additionally, to ensure that all results are actually generated, the program iterates through the entire result set before considering each evaluation completed.

4.4. Results and analysis

Tables 4.2, 4.3 and 4.4 summarize the average running time of all queries before and after transformations are applied for both query engines, separated by rule application. It also shows how many queries had to be halted after being allowed to run for two minutes of real time.

[8] http://aws.amazon.com/ec2/instance-types/#instance-details

[9] http://aws.amazon.com/ec2/faqs/#What_is_an_EC2_Compute_Unit_and_why_did_you_introduce_it

4.4.1. Rules R1 through R3

The first thing one might notice with regards to rule applicability is how seldom are two separate rules applicable. This makes the possibly interesting comparison of which rules have the strongest effect on query execution time over a single query, statistically insignificant. Worse, rule R4 is never applied, thus giving no information over its use. However, the fact that rule R4 simply shifts nodes around without actually reducing the number of nodes, triples or OPT operators suggests that this rule is not the best suited for query optimization.

Rule R1 has very few applications, even when considering its use in conjunction with other queries. This is to be expected, since the repetition of triples within sub-nodes should be a fairly evident mistake from the query designer.

Rule R2 provides by far the strongest improvement, stemming from the fact that the rule eliminates an entire node, which includes its triples and an OPT operator. In the Top-down engine, this is so extreme that it even beats the Jena-ARQ main engine *after* optimization. This makes this rule the most likely explanation for the strong reduction in time whenever both R1 and R2 are applied. Unfortunately the rule is only applicable in around 1% of those queries where rule applications are possible, which makes these transformations a rare but worthwhile find.

Rule R3 also provides a marked increase in both engines. While the increase might not appear too great in the Jena-ARQ case, note that the number of queries that hit the time limit is reduced from 12 to 0, showing that the average time hides the usefulness of the rule in making difficult queries tractable and how much better Jena-ARQ is at handling inner joins than left joins.

It is interesting as well that, despite the exhaustive search for homomorphisms within each query, their fairly small size makes rule application very fast in practice, especially when compared to the average gain in query execution time. Rule R3, which has the slowest transformation time at 0.4 milliseconds, is marginal when compared to the provided

improvement in both engines. The opposite side of this is rule R5, which when considering transformation time ends up improving very little in the Top-down engine and actually being worse over Jena-ARQ.

Overall, the strategy offers a considerable improvement in query execution time even after factoring in transformation times, as shown in Figures 4.3 and 4.4, and it is therefore a sound query optimization technique.

			Top-down				Jena-ARQ			
			Original query		Modified query		Original query		Modified query	
Rules	Count	Transform time (ms)	Time (ms)	# Halted	Time (ms)	# Halted	Time (ms)	# Halted	Time (ms)	# Halted
R1	4	0	12.25	0	0.75	0	7.75	0	1	0
R1R2	5	0.4	32756.6	1	25248.8	1	26490.6	1	24855.6	1
R1R3	1	0	129	0	5737	0	21	0	114	0
R1R5	1	0	1	0	1	0	2	0	1	0
R2	63	0.126984	2429.06	0	7.66667	0	277.079	0	8.34921	0
R3	249	0.405622	6427.32	14	3592.87	2	131.948	12	127.341	0
R4	0	—	—	—	—	—	—	—	—	—
R5	6367	0.199937	3.41228	1	3.31522	0	0.474635	1	0.464897	0
Combined	6690	0.206876	289.854	16	156.682	3	27.7788	14	23.8552	1

TABLE 4.2. Experimental results for DBpedia

			Top-down				**Jena-ARQ**			
			Original query		Modified query		Original query		Modified query	
Rules	Count	Transform time (ms)	Time (ms)	# Halted	Time (ms)	# Halted	Time (ms)	# Halted	Time (ms)	# Halted
R1	0	—	—	—	—	—	—	—	—	—
R2	0	—	—	—	—	—	—	—	—	—
R3	151	0.549669	17596.9	0	62.457	0	45.6291	0	16.0397	0
R4	0	—	—	—	—	—	—	—	—	—
R5	0	—	—	—	—	—	—	—	—	—
Combined	151	0.549669	17596.9	0	62.457	0	45.6291	0	16.0397	0

TABLE 4.3. Experimental results for Linked Open Geo Data

			Top-down				Jena-ARQ			
			Original query		Modified query		Original query		Modified query	
Rules	Count	Transform time (ms)	Time (ms)	# Halted	Time (ms)	# Halted	Time (ms)	# Halted	Time (ms)	# Halted
R1	5	0.2	26.8	0	9.4	0	11	0	4.6	0
R1R2	4	0	24892	0	68.75	0	787.5	0	21	0
R1R3	5	0.4	1	0	1	0	2	0	1.4	0
R2	5	0.2	15546.8	0	31.4	0	96.8	0	15.8	0
R3	26	0.153846	23542.3	5	16313.8	0	21.1154	0	14.9231	0
R4	0	—	—	—	—	—	—	—	—	—
R5	72	1.66667	883.722	0	876.792	0	18	0	27.6389	0
Combined	117	1.07563	7169.88	0	4098.92	0	46.5882	0	21.605	0

TABLE 4.4. Experimental results for Semantic Web Dog Food

4.4.2. Rule R5

Despite being the most applicable, Rule R5 appears at first to be somewhat of a disappointment. On DBpedia its overall average effect is barely noticeable, while on SWDF it varies greatly between query engines, being a considerable improvement on the Top-down engine, but damaging query evaluation time on Jena-ARQ. This is to be expected to some extent in Jena-ARQ due to its iterator based approach: pruned partial mappings on the left side of an OPT operator should also prune results inside it when looking for compatible mappings. This means that in some cases, adding FILTER expressions to the text only results in having to iterate an additional time over a given dataset, which might have a significant cost in time. However, since the Top-down engine evaluates basic graph patterns before left joining, filtering results between these operations ought to greatly reduce the number of compatibility tests required, and thus evaluation time. Therefore, we have further studied this particular case.

One would expect a "difficult" query to have a greater optimization potential than an "easy" one. However, query execution time is extremely short for DBpedia, taking less than 4 milliseconds on the Top-down engine and less than 0.5 milliseconds on Jena-ARQ on average. This suggests that the actual effect of the rule is hidden by a very high number of very fast queries. In fact, out of the 6,440 queries for which rule R5 is applicable, only 97 take more than 10 milliseconds on the Top-down engine and 66 on Jena-ARQ meaning that relatively few intermediate results are involved, and thus not many mappings can be pruned by pushing FILTER operators down the pattern tree.

To make our results more significant, we ran the same queries with rule R5 applications with a much larger database of 30,000 triples, in order to see more queries taking more than 10 milliseconds. This resulted in 124 queries on the Top-down engine and 104 on Jena-ARQ.

	Top-down	Jena-ARQ
Number of queries	124	104
Average time for unmodified queries (ms)	1140.27	58.2692
Average time for modified queries (ms)	1113.94	44.3654
Average time improvement (ms)	26.3226	13.9038
Total percentual improvement	2.30846%	23.8614%
Number of queries which improve	101	80
Percentage of queries which improve	81.4516%	76.9231%

TABLE 4.5. Effect of applying rule R5 to queries which take more than 10 milliseconds when evaluated over 30000 triples

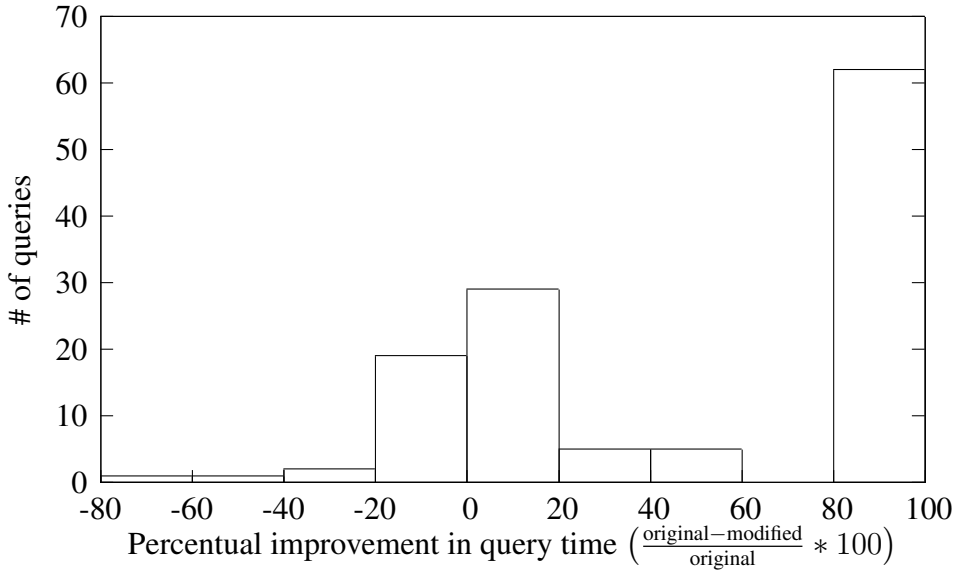


FIGURE 4.1. Distribution of percentual improvement on evaluation for rule R5 on Top-down engine, over "slow" queries

For these queries, we have much better results, which are illustrated in Figures 4.1 and 4.2, which show the distribution of percentual improvement in query execution time. A brief summary of results is shown in table 4.5.

There is a noticeable improvement now over the Jena-ARQ query engine. Notice how most queries now show at least some improvement, with many of them showing an almost 100% decrease in execution time. Unfortunately, in the relatively few cases where the transformation impacts negatively, there is a sharp increase in time, which partly negates

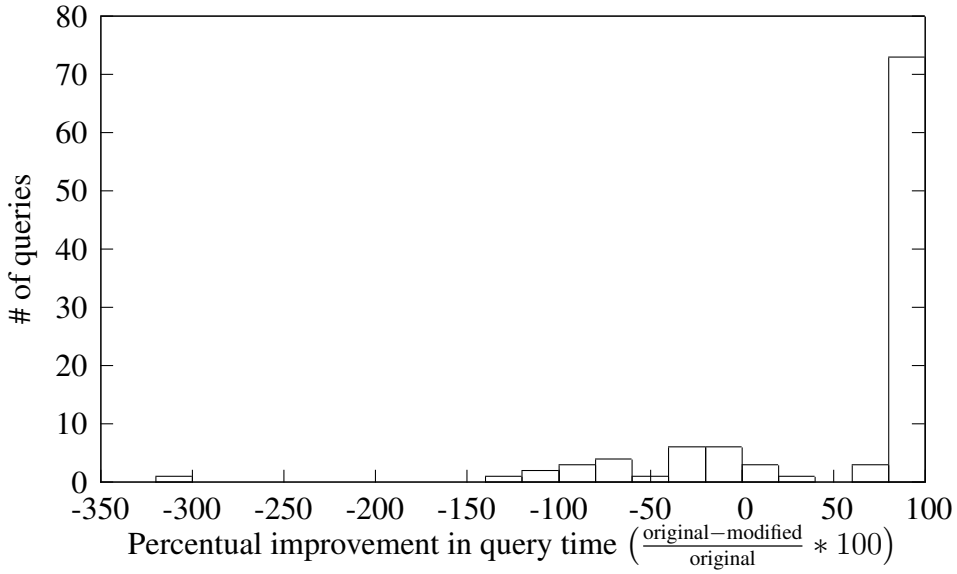


FIGURE 4.2. Distribution of percentual improvement on evaluation time for rule R5 on Jena-ARQ engine, over "slow" queries

the overall effect of the transformation. This is particularly illustrated in Figure 4.2, where one query quadruples in execution time. All of this suggests that while blindly applying rule R5 does indeed improve average execution times for slower queries, the change in query time heavily depends on the query.

One possible solution to this problem could be to estimate the size of the result set of each node before and after applying the transformation. Much work has been done in this area (Markl et al., 2005; Ioannidis & Poosala, 1995; Quilitz & Leser, 2008), and mature query engines like IBM's DB2, Informix, Microsoft SQL Server, Oracle 8, and Sybase Ase all use several techniques to estimate query characteristics, such as result size and cost (Ramakrishnan & Gehrke, 2003, p. 485). In fact, this technique has already been well studied and applied on the relational equivalent of the FILTER operator, the WHERE clause. It has been shown how one can model the effect of the WHERE clause on the result size by associating a reduction factor with each term, which is the ratio of the expected result size of the input considering only the selection represented by the term. The actual size of the result can be estimated as the input size times the product

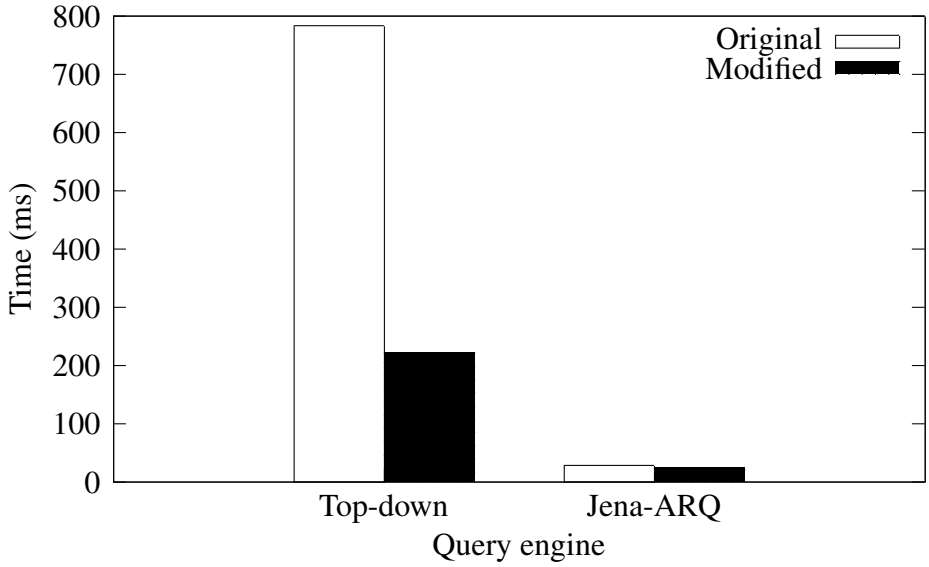


FIGURE 4.3. Query evaluation time improvement for all rules

of the reduction factors for all the terms in the where clause. This estimate reflects the simplifying but useful assumption that the conditions tested by each term are statistically independent (Ramakrishnan & Gehrke, 2003, p. 485).

These techniques can be easily applied to our case: if one could reliably estimate the number of partial mappings pruned by pushing each FILTER operator down the query, and it would be fairly simple to decide whether to apply rule R5 or not. However, this is beyond the scope of this thesis, and is merely proposed as future work.

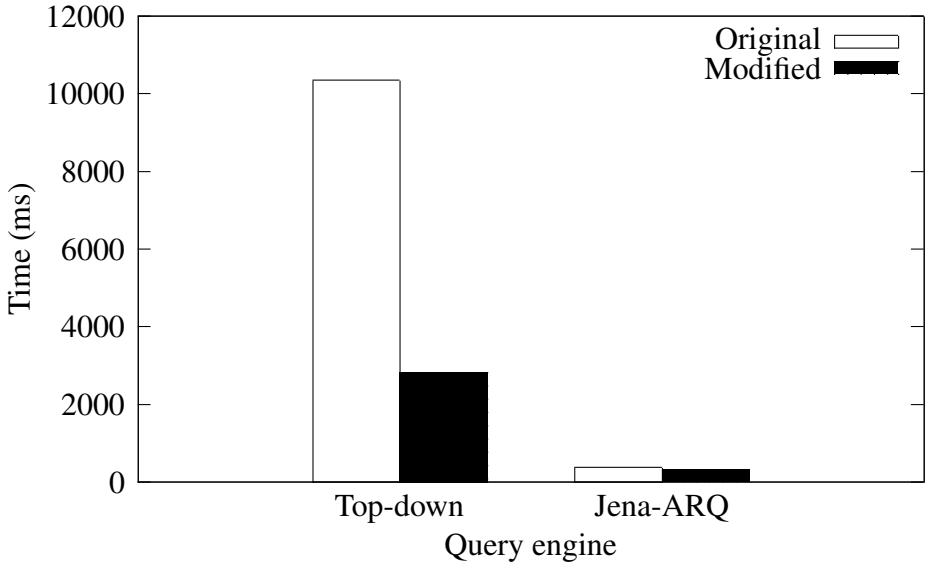


FIGURE 4.4. Query evaluation time improvement for all rules except R5

Chapter 5. CONCLUSIONS AND FUTURE RESEARCH

5.1. General remarks

In this work we have extended the notion of pattern trees (Letelier et al., 2012b) to support queries which include the filtering feature of SPARQL, showing ways to build, evaluate and transform them in a way that is fully compatible with previous work. Along with this, we have presented a successful strategy for optimizing well-designed SPARQL graph patterns, demonstrating how well pattern tree transformation rules work as a logical query plan optimizer. Furthermore, we have implemented our results over the Jena platform, allowing us to handle massive sets of query logs programmatically. To complement this, we have developed a technique which allows us to extract a relevant subset of an RDF graph for a given SPARQL query, which makes testing possible under strict memory constraints. Finally, we have proposed a strategy to apply the new FILTER based transformation rule selectively, to improve its optimization potential.

5.2. Future research topics

There are several goals towards which this work might be extended.

Possibly the most immediate line of work would be to study how to combine the use of rule R5 with algorithms that predict its effectiveness, to decide when it is worth applying. This is most likely a direct application of previous work directed towards the WHERE clause of relational databases, and thus should not be difficult to adapt to our work.

Another interesting idea would be to combine the work done here, to add the FILTER operator, with the one done in Letelier et al. (in press) to include the use of projection. This would further increase the expressive power of QWDPTs, thus capturing a larger fragment of SPARQL for which we are currently unequipped to deal with.

Finally, as part of our future work, we would like to study the main problem from Letelier et al. (2012b), of query containment and equivalence in the presence of FILTER

operators, as this is both an interesting problem and the foundation for future query optimization techniques.

References

Abadi, D. J., Marcus, A., Madden, S., & Hollenbach, K. J. (2007). Scalable semantic web data management using vertical partitioning. In *VLDB* (p. 411-422).

Angles, R., & Gutierrez, C. (2008). The expressive power of SPARQL. In *ISWC* (p. 114-129).

Apache Jena. (2013). http://jena.apache.org/.

Arenas, M., & Pérez, J. (2011). Querying semantic web data with SPARQL. In *PODS* (p. 305-316).

Berendt, B., Hollink, L., Luczak-Rösch, M., Möller, K. H., & Vallet, D. (2013). USEWOD2013 – 3rd International Workshop on Usage Analysis and the Web of Data. In *10th ESWC – semantics and big data, Montpellier, France.*

Berners-Lee, T. (2006). *Linked data – design issues.* http://www.w3.org/DesignIssues/LinkedData.html.

Bizer, C., Heath, T., & Berners-Lee, T. (2009). Linked data - the story so far. *International Journal of Semantic Web and Information Systems*, *5*(3), 1-22.

Carroll, J. J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., & Wilkinson, K. (2004). Jena: implementing the semantic web recommendations. In S. I. Feldman, M. Uretsky, M. Najork, & C. E. Wills (Eds.), *WWW (alternate track papers & posters)* (p. 74-83). ACM.

Chekol, M., Euzenat, J., Genevès, P., & Layaïda, N. (2011). PSPARQL query containment. In *DBPL.*

Data.gov. (2013). http://www.data.gov.

Data.gov.uk. (2013). http://data.gov.uk.

Gallego, M. A., Fernández, J. D., Martínez-Prieto, M. A., & de la Fuente, P. (2011). An empirical study of real-world SPARQL queries. *CoRR*, *abs/1103.5043*.

Garcia-Molina, H., Ullman, J. D., & Widom, J. (2009). *Database systems - the complete book (2. ed.)*. Pearson Education.

Gutiérrez, C., Hurtado, C. A., & Mendelzon, A. O. (2004). Foundations of semantic web databases. In C. Beeri & A. Deutsch (Eds.), *PODS* (p. 95-106). ACM.

Ioannidis, Y. E., & Poosala, V. (1995, May). Balancing histogram optimality and practicality for query result size estimation. *SIGMOD Rec.*, *24*(2), 233–244. doi: 10.1145/568271.223841

Lassila, O., & Swick, R. (1999). Resource description framework (RDF) model and syntax. *W3C Recommendation*. http://www.w3.org/TR/PR-rdf-syntax.

Letelier, A., Pérez, J., Pichler, R., & Skritek, S. (2012a). SPAM: A SPARQL Analysis and Manipulation Tool. *PVLDB*, *5*(12), 1958-1961.

Letelier, A., Pérez, J., Pichler, R., & Skritek, S. (2012b). Static analysis and optimization of semantic web queries. In M. Benedikt, M. Krötzsch, & M. Lenzerini (Eds.), *PODS* (p. 89-100). ACM.

Letelier, A., Pérez, J., Pichler, R., & Skritek, S. (in press). Static analysis and optimization of semantic web queries. In *TODS.*

Markl, V., Megiddo, N., Kutsch, M., Tran, T. M., Haas, P. J., & Srivastava, U. (2005). Consistently estimating the selectivity of conjuncts of predicates. In *VLDB* (p. 373-384).

Neumann, T., & Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *VLDB J.*, *19*(1), 91-113.

Pérez, J., Arenas, M., & Gutierrez, C. (2006). Semantics and complexity of SPARQL. In *ISWC* (p. 30-43).

Pérez, J., Arenas, M., & Gutierrez, C. (2009). Semantics and complexity of SPARQL. *TODS*, *34*(3).

Picalausa, F., & Vansummeren, S. (2011). What are real SPARQL queries like? In *SWIM* (p. 7).

Polleres, A. (2007). From SPARQL to rules (and back). In *WWW* (pp. 787–796).

Prud'Hommeaux, E., & Seaborne, A. (2008). SPARQL query language for RDF. *W3C Recommendation*.

Quilitz, B., & Leser, U. (2008). Querying distributed rdf data sources with SPARQL. In S. Bechhofer, M. Hauswirth, J. Hoffmann, & M. Koubarakis (Eds.), *ESWC* (Vol. 5021, p. 524-538). Springer.

Ramakrishnan, R., & Gehrke, J. (2003). *Database management systems (3. ed.)*. McGraw-Hill.

Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., & Pinkel, C. (2008). An experimental comparison of rdf data management approaches in a SPARQL benchmark scenario. In *International semantic web conference* (p. 82-97).

Schmidt, M., Meier, M., & Lausen, G. (2010). Foundations of sparql query optimization. In *Proceedings of the 13th international conference on database theory* (pp. 4–33). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1804669.1804675` doi: 10.1145/1804669.1804675

Serfiotis, G., Koffina, I., Christophides, V., & Tannen, V. (2005). Containment and minimization of RDF/S query patterns. In *International semantic web conference* (p. 607-623).

Sidirourgos, L., Goncalves, R., Kersten, M. L., Nes, N., & Manegold, S. (2008). Column-store support for RDF data management: not all swans are white. *PVLDB*, *1*(2), 1553-1563.

Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., & Reynolds, D. (2008). SPARQL basic graph pattern optimization using selectivity estimation. In J. Huai et al. (Eds.), *WWW* (p. 595-604). ACM.

Weiss, C., Karras, P., & Bernstein, A. (2008). Hexastore: sextuple indexing for semantic web data management. *PVLDB*, *1*(1), 1008-1019.

APPENDIX

APPENDIX A. ADDITIONAL PROOFS

A.1. Proof of Proposition 3.1

PROOF. Let $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ be a well-designed pattern tree, Σ an arbitrary order of $\mathcal{T}$, and P be the graph pattern $\text{TR}(\mathcal{T}, \Sigma)$. We next show that if P is not a well-designed graph pattern, then $\mathcal{T}$ is not a well-designed pattern tree. Suppose that P is not well-designed. We have two possibilities:

- there exists a subpattern $P' = (P_1 \ \text{OPT} \ P_2)$ of P, and a variable $?X$ which occurs in P_2 and outside P' in P, but does not occur in P_1. Notice that in the definition of TR, every OPT operator is included when transforming a particular node $n \in V$ and one of its children. Thus assume that the OPT operator in subpattern $P' = (P_1 \ \text{OPT} \ P_2)$ is created when transforming node n and its children. Then if $\{n_1, \ldots, n_k\}$ are the children of n, by following the definition of TR we know that there exists a value $i \in \{1, \ldots, k\}$ such that

$$P_1 = \Big(\cdots \Big(\Big(\Big(\Big(\text{TR}(P_n, F_n) \Big) \ \text{OPT} \ \text{TR}(\mathcal{T}, \sigma_n(1), \Sigma) \Big) \ \text{OPT} \ \text{TR}(\mathcal{T}, \sigma_n(2), \Sigma) \Big) \cdots \Big) \ \text{OPT} \ \text{TR}(\mathcal{T}, \sigma_n(i-1), \Sigma) \Big), \tag{A.1}$$

and

$$P_2 \ = \ \text{TR}(\mathcal{T}, \sigma_n(i), \Sigma) \tag{A.2}$$

Notice that since $?X$ appears in P_2, then we know that $?X$ appears in the subtree rooted at $\sigma_n(i)$. Moreover, since $?X$ does not appear in P_1, we have that $?X$ does not appear in P_n (and also it does not appear in any of the subtrees rooted at $\sigma_n(1), \ldots, \sigma_n(i-1)$). Finally, the fact that $?X$ also appears "outside" P', implies that there exists a node v with $v \neq n$ and that is not in the subtree rooted at $\sigma_n(i)$, such that $?X$ appears in P_v. This is enough to conclude that $\mathcal{T}$

violates the connected condition in Definition 3.2 for variable $?X$, and thus $\mathcal{T}$ is not a well-designed pattern tree.

- there exists a subpattern $P' = (P_1 \text{ FILTER } R)$ of P and a variable $?X$ that occurs inside R but not in P_1. From the definition of TR, every FILTER operator is created when evaluating some node $n \in V$. Assume that P' is created when transforming node n and its children. Thus, by following the definition of TR, we know that $\text{TR}(\mathcal{T}, n, \Sigma)$ introduced the subpattern $(and(P_n) \text{ FILTER } conj(F_n))$, with the variable $?X$ occurring inside F_n but not in P_n. Since $\text{vars}(F_n) \not\subseteq \text{vars}(P_n)$, $\mathcal{T}$ violates the safety condition in Definition 3.2, and thus it is not a well-designed pattern tree.

This completes our proof. □

A.2. Proof of Lemma 3.1

PROOF. This lemma is a direct consequence of Proposition 3.1 and the soundness of the following rewrite rule for well-designed patterns:

$$((P_1 \text{ OPT } P_2) \text{ OPT } P_3) \longrightarrow ((P_1 \text{ OPT } P_3) \text{ OPT } P_2). \tag{A.3}$$

It was proven in (Pérez et al., 2006) that if P is a well-designed graph pattern, and P' is obtained from P by applying the above rule to some subpattern of P, then P' is also well-designed and $P \equiv P'$. Thus assume that $\Sigma_1 = \{\sigma_n^1 \mid n \in V\}$ and $\Sigma_2 = \{\sigma_n^2 \mid n \in V\}$ are arbitrary orderings for $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V}))$, and let n be a node in $\mathcal{T}$ with set of children $\{n_1, \ldots, n_k\}$. Rule (A.3) ensures that, by applying the rule to subpatterns of the form

$$\big(\big(P_1' \text{ OPT } \text{TR}(\mathcal{T}, \sigma_n^1(i), \Sigma_1)\big) \text{ OPT } \text{TR}(\mathcal{T}, \sigma_n^1(j), \Sigma_1)\big),$$

we can go from pattern

$$\textsc{Tr}(\mathcal{T}, n, \Sigma_1) \;=\; \Big(\cdots \Big(\Big(\Big(\textsc{Tr}(P_n, F_n) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n^1(1), \Sigma_1) \Big) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n^1(2), \Sigma_1) \Big) \;\cdots\; \Big) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n^1(k), \Sigma_1) \Big),$$

to pattern

$$\textsc{Tr}(\mathcal{T}, n, \Sigma_2) \;=\; \Big(\cdots \Big(\Big(\Big(\textsc{Tr}(P_n, F_n) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n^2(1), \Sigma_2) \Big) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n^2(2), \Sigma_2) \Big) \;\cdots\; \Big) \text{ OPT } \textsc{Tr}(\mathcal{T}, \sigma_n^2(k), \Sigma_2) \Big),$$

thus proving the lemma. □

A.3. Proof of Lemma 3.2

In proving this lemma we make use of the following claim:

CLAIM A.1. *Consider the graph pattern*

$$P \;=\; \Big(\big((t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_k \text{ AND } t^*) \text{ FILTER } R \big) \text{ OPT } \big(t^* \text{ AND } Q \big) \Big)$$

with $t, t_1, \ldots t_k$ triple patterns, R and Q an arbitrary graph pattern, and assume that $\mathrm{vars}(R) \subseteq \bigcup_{1 \le i \le k} \mathrm{vars}(t_i)$. *Then P is equivalent to the graph pattern*

$$P' \;=\; \Big(\big((t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_k \text{ AND } t^* \big) \text{ FILTER } R) \text{ OPT } Q \Big)$$

PROOF. First, it is easy to see that P is well-designed. It was proven in (Pérez et al., 2006) that a pattern of the form $\big((P_1 \text{ OPT } P_2) \text{ FILTER } R \big)$ which is well-designed is equivalent to $\big((P_1 \text{ FILTER } R) \text{ OPT } P_2 \big)$. We can use this property to transform P and P' into $\big(\big((t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_k \text{ AND } t^*) \text{ OPT } (t^* \text{ AND } Q) \big) \text{ FILTER } R \big)$ and $\big(\big(t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_k \text{ AND } t^* \big) \text{ OPT } (Q) \big) \text{ FILTER } R \big)$ respectively.

Thus, the problem is reduced to proving that

$$\big((t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_k \text{ AND } t^*) \text{ OPT } (t^* \text{ AND } Q)\big)$$

is equivalent to

$$\big((t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_k \text{ AND } t^*) \text{ OPT } (Q)\big),$$

which follows easily from the definitions of AND and OPT. □

Thus, we can now proceed proving lemma 3.2.

PROOF. In order to prove the lemma, we show the following property. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be well-designed pattern trees such that $\mathcal{T}_1 \hookrightarrow^* \mathcal{T}_2$, and let $P_1 = \text{TR}(\mathcal{T}_1, \Sigma)$ and $P_2 = \text{TR}(\mathcal{T}_2, \Sigma)$. We claim that $P_1 \equiv P_2$. Since the well-designed property is invariant under $\hookrightarrow^*$, it is enough to show that the property holds whenever $\mathcal{T}_1 \hookrightarrow \mathcal{T}_2$ (that is, $\mathcal{T}_2$ is obtained in a single step from $\mathcal{T}_1$). Thus assume that $\mathcal{T}_1 \hookrightarrow \mathcal{T}_2$. Then $\mathcal{T}_1 = ((V, E, r), (P_n^1)_{n \in V}, (F_n^1)_{n \in V})$ and $\mathcal{T}_2 = ((V, E, r), (P_n^2)_{n \in V}, (F_n^2)_{n \in V})$, and there exists a node u, a triple $t \in P_u^1$, and a child v of u such that $P_v^2 = P_v^1 \cup \{t\}$ and $R_v^2 = R_v^1$, and for every $n \neq v$ it holds that $P_n^2 = P_n^1$ and $R_n^2 = R_n^1$. We next show that $\text{TR}(\mathcal{T}_1, u, \Sigma) \equiv \text{TR}(\mathcal{T}_2, u, \Sigma)$, which by the construction of $\text{TR}(\mathcal{T}_1, \Sigma)$ and $\text{TR}(\mathcal{T}_2, \Sigma)$, implies that $\text{TR}(\mathcal{T}_1, \Sigma) \equiv \text{TR}(\mathcal{T}_2, \Sigma)$.

Thus, assume that u has k children. Then since v is a child of u we know that there exists an index $i \in \{1, \ldots, k\}$ such that $\sigma_u(i) = v$, and then

$$\begin{aligned} \text{TR}(\mathcal{T}_1, u, \Sigma) \;=\; \Big(\cdots \Big(\Big(\text{TR}(P_u^1, F_u^1) \text{ OPT } \text{TR}(\mathcal{T}_1, \sigma_u(1), \Sigma) \Big) \;\cdots\; \Big) \\ \text{OPT } \text{TR}(\mathcal{T}_1, \sigma_u(i), \Sigma) \Big) \;\cdots\; \Big) \text{ OPT } \text{TR}(\mathcal{T}_1, \sigma_u(k), \Sigma) \Big) \end{aligned}$$

and

$$\textsc{Tr}(\mathcal{T}_2, u, \Sigma) \;=\; \Big(\cdots \Big(\Big(\textsc{tr}(P_u^2, F_u^2) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_u(1), \Sigma) \Big) \;\cdots\; \Big) \\ \text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_u(i), \Sigma) \Big) \;\cdots\; \Big) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_u(k), \Sigma) \Big).$$

By rule (A.3), and since both patterns are well-designed, we know that the patterns above are equivalent to

$$\Big(\cdots \Big(\Big(\textsc{tr}(P_u^1, F_u^1) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_1, \sigma_u(i), \Sigma) \Big) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_1, \sigma_u(1), \Sigma) \Big) \;\cdots\; \Big) \\ \text{OPT}\; \textsc{Tr}(\mathcal{T}_1, \sigma_u(k), \Sigma) \Big)$$

and

$$\Big(\cdots \Big(\Big(\textsc{tr}(P_u^2, F_u^2) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_u(i), \Sigma) \Big) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_u(1), \Sigma) \Big) \;\cdots\; \Big) \\ \text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_u(k), \Sigma) \Big),$$

respectively. Moreover, since $\mathcal{T}_1$ and $\mathcal{T}_2$ differ only in the label P_v of node $v = \sigma_u(i)$, we have that for every $j \in \{1, \ldots, k\}$ such that $j \neq i$, it holds that $\textsc{Tr}(\mathcal{T}_1, \sigma_u(j), \Sigma) = \textsc{Tr}(\mathcal{T}_2, \sigma_u(j), \Sigma)$. Thus, in order to prove that $\textsc{Tr}(\mathcal{T}_1, u, \Sigma) \equiv \textsc{Tr}(\mathcal{T}_2, u, \Sigma)$ it is enough to show that (recall that $v = \sigma_u(i)$)

$$\Big(\textsc{tr}(P_u^1, F_u^1) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_1, v, \Sigma) \Big) \;\equiv\; \Big(\textsc{tr}(P_u^2, F_u^2) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_2, v, \Sigma) \Big). \qquad \text{(A.4)}$$

We next show that Property (A.4) holds. Assume that v has ℓ children. Then the right-hand side of (A.4) can be written as

$$\Big(\textsc{tr}(P_u^2, F_u^2) \;\text{OPT}\; \Big(\cdots \Big(\textsc{tr}(P_v^2, F_v^2) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_v(1), \Sigma) \Big) \\ \cdots \quad \text{OPT}\; \textsc{Tr}(\mathcal{T}_2, \sigma_v(\ell), \Sigma) \Big) \Big). \qquad \text{(A.5)}$$

Now, recall that $P_u^2 = P_u^1$ and that $P_v^2 = P_v^1 \cup \{t\}$. Moreover, for all the children of v we have that $\mathcal{T}_1$ and $\mathcal{T}_2$ coincide. Thus we have that (A.5) can be written as

$$\Big(\textsc{tr}(P_u^1, F_u^1) \text{ OPT } \Big(\cdots \Big(\big(t \text{ AND } \textsc{tr}(P_v^1, F_v^1)\big) \text{ OPT } \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big) \cdots \text{ OPT } \textsc{Tr}(\mathcal{T}_1, \sigma_v(\ell), \Sigma)\Big)\Big). \quad \text{(A.6)}$$

We now make use of the following property proved in (Pérez et al., 2006). If the graph patterns $R = ((R_1 \text{ AND } R_2) \text{ OPT } R_3)$ and $R' = (R_1 \text{ AND } (R_2 \text{ OPT } R_3))$ are well-designed with R_1, R_2, and R_3 arbitrary patterns, then $R \equiv R'$. We want to apply this equivalence rule to expression (A.6). For this we first argue that the pattern

$$\Big(t \text{ AND } \Big(\textsc{tr}(P_v^1, F_v^1) \text{ OPT } \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big)\Big) \quad \text{(A.7)}$$

is well-designed. To show this, since we know from hypothesis that $\textsc{tr}(P_v^1, F_v^1)$ is safe, we only need to prove that if there is a variable $?X$ that occurs in $\textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)$ and also in t, then this variable should occur in $\textsc{tr}(P_v^1, F_v^1)$. Recall that t is a triple in P_u^1 and that u is the father of v. Moreover, if $?X$ occurs in $\textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)$ then we know that there is a descendant w of v such that $?X$ occurs in P_w^1. Thus we have that $?X$ occurs in P_u^1, and in P_w^1 for a descendant w of v, and then, since u is the father of v and $\mathcal{T}_1$ is a well-designed pattern tree, we obtain that $?X$ occurs in P_v^1. This ensures that (A.7) is well-designed. It is also easy to see that

$$\Big(\big(t \text{ AND } \textsc{tr}(P_v^1, F_v^1)\big) \text{ OPT } \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big)$$

is well-designed, since $\mathcal{T}_2$ is well-designed. Thus we can apply the mentioned rule to obtain the (A.6) is equivalent to

$$\Big(\textsc{tr}(P_u^1, F_u^1) \text{ OPT } \Big(\cdots \Big(\Big[t \text{ AND } \Big((and(P_v^1) \text{ FILTER } R_v^1) \text{ OPT } \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big)\Big] \cdots \text{ OPT } \textsc{Tr}(\mathcal{T}_1, \sigma_v(\ell), \Sigma)\Big)\Big).$$

We can apply exactly the same argument to show that the last pattern is equivalent to

$$\Big(\textsc{tr}(P_u^1, F_u^1)\ \text{OPT}\ \Big(\cdots\Big(\Big[t\ \text{AND}\ \Big(\Big(\textsc{tr}(P_v^1, F_v^1)\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big) \\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(2), \Sigma)\Big)\Big]\ \cdots\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(\ell), \Sigma)\ \Big)\Big).$$

If we keep applying the same transformation we reach the pattern

$$\Big(\textsc{tr}(P_u^1, F_u^1)\ \text{OPT}\ \Big[t\ \text{AND}\ \Big(\cdots\Big(\textsc{tr}(P_v^1, F_v^1)\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big)\ \cdots \\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(\ell), \Sigma)\ \Big)\Big]\Big). \quad \text{(A.8)}$$

Now, notice that $t \in P_u^1$ and thus t is one of the triple patterns that occur in $\textsc{tr}(P_u^1, F_u^1)$. Thus we can apply Claim A.1 to expression (A.8) and obtain

$$\Big(\textsc{tr}(P_u^1, F_u^1)\ \text{OPT}\ \Big(\cdots\Big(\textsc{tr}(P_v^1, F_v^1)\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(1), \Sigma)\Big) \\ \cdots\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, \sigma_v(\ell), \Sigma)\ \Big)\Big),$$

which is exactly

$$\Big(\textsc{tr}(P_u^1, F_u^1)\ \text{OPT}\ \textsc{Tr}(\mathcal{T}_1, v, \Sigma)\Big),$$

and thus completing the proof that (A.4) holds. This proves that $\textsc{Tr}(\mathcal{T}_1, u, \Sigma) \equiv \textsc{Tr}(\mathcal{T}_2, u, \Sigma)$, which in turns implies that $\textsc{Tr}(\mathcal{T}_1, \Sigma) \equiv \textsc{Tr}(\mathcal{T}_2, \Sigma)$.

So far we have shown that if $\mathcal{T}_1$ and $\mathcal{T}_2$ are well-designed pattern trees such that $\mathcal{T}_1 \hookrightarrow^* \mathcal{T}_2$, then $\textsc{Tr}(\mathcal{T}_1, \Sigma) \equiv \textsc{Tr}(\mathcal{T}_2, \Sigma)$. Next we show that this implies that Lemma 3.2 holds.

Let $\mathcal{T}$ be a quasi well-designed pattern tree, and assume that $\mathcal{T}_1$ and $\mathcal{T}_2$ are well-designed pattern trees such that $\mathcal{T} \hookrightarrow^* \mathcal{T}_1$ and $\mathcal{T} \hookrightarrow^* \mathcal{T}_2$. We need to prove that $\textsc{Tr}(\mathcal{T}_1, \Sigma) \equiv \textsc{Tr}(\mathcal{T}_2, \Sigma)$. In order to show this consider the pattern $\mathcal{T}^*$ that is the *maximum* element such that $\mathcal{T} \hookrightarrow^* \mathcal{T}^*$. That is, $\mathcal{T}^*$ is such that for every other tree $\mathcal{T}'$ such that $\mathcal{T} \hookrightarrow^* \mathcal{T}'$ it holds that $\mathcal{T}' \hookrightarrow^* \mathcal{T}^*$. It is straightforward to show that $\mathcal{T}^*$ exists and is unique. Moreover, it

is also easy to see that since $\mathcal{T}$ is quasi well-designed, then $\mathcal{T}^*$ is well-designed. Thus we have that $\mathcal{T}_1 \hookrightarrow^* \mathcal{T}^*$ and since $\mathcal{T}_1$ and $\mathcal{T}^*$ are well-designed from the property that we proved in the previous paragraph, we obtain that $\text{TR}(\mathcal{T}_1, \Sigma) \equiv \text{TR}(\mathcal{T}^*, \Sigma)$. Similarly, we can show that $\text{TR}(\mathcal{T}_2, \Sigma) \equiv \text{TR}(\mathcal{T}^*, \Sigma)$, and thus $\text{TR}(\mathcal{T}_1, \Sigma) \equiv \text{TR}(\mathcal{T}_2, \Sigma)$. This completes the proof of Lemma 3.2. □

A.4. Proof of Theorem 3.2

PROOF. In order to prove this theorem we require the following claim:

Claim 1: Let P_1 and P_2 be sets of triple patterns, and F_1 and F_2 be sets of built-in conditions, such that $\text{vars}(F_1) \subseteq \text{vars}(P_1)$ and $\text{vars}(F_2) \subseteq \text{vars}(P_2)$. Then, for any dataset G,

$$[\![\text{TR}(P_1, F_1)]\!]_G \bowtie [\![\text{TR}(P_2, F_2)]\!]_G = [\![\text{TR}(P_1 \cup P_2, F_1 \cup F_2)]\!]_G.$$

Proof of Claim 1: Given a mapping μ and a set of built-in conditions F, we write $\mu \models F$ to say that, for every built-in condition $R \in F$, $\mu \models R$. If F is empty, then this property trivially holds.

⇒*)* Let $\mu \in [\![\text{TR}(P_1, F_1)]\!]_G \bowtie [\![\text{TR}(P_2, F_2)]\!]_G$. Then $\mu = \mu_1 \cup \mu_2$, with $\mu_1 \sim \mu_2$, $\mu_1 \in [\![P_1]\!]_G$, $\mu_2 \in [\![P_2]\!]_G$, $\mu_1 \models F_1$ and $\mu_2 \models F_2$. Since $\mu_1 \sim \mu_2$, then both mappings coincide in their valuations for all variables in $\text{dom}(\mu_1) \cap \text{dom}(\mu_2)$. Furthermore, since $\text{vars}(F_1) \subseteq \text{dom}(\mu_1)$, then it is also the case that $(\mu_1 \cup \mu_2) \models F_1$. If this was not the case, then there would be a variable $?X \in \text{dom}(\mu_2)$ and a built-in condition $R \in F_1$ with $?X$ mentioned in R, such that $\mu_2(?X)$ maps to some value that makes R false; however, since $\text{vars}(R) \subseteq \text{vars}(\mu_1)$ then $?X \in \text{vars}(\mu_1)$, and since $\mu_1 \sim \mu_2$, then $\mu_1(?X) = \mu_2(?X)$, and thus $\mu_1 \not\models R$, which contradicts our hypothesis. By using the same argument, we have that $(\mu_1 \cup \mu_2) \models F_2$, and then $(\mu_1 \cup \mu_2) \models F_1 \cup F_2$. Therefore, we have that $\mu \models F_1 \cup F_2$, and thus $[\![\text{TR}(P_1 \cup P_2, F_1 \cup F_2)]\!]_G$.

⇐*)* Let $\mu \in [\![\text{TR}(P_1 \cup P_2, F_1 \cup F_2)]\!]_G$. Then $\mu = \mu_1 \cup \mu_2$, with $\mu_1 \sim \mu_2$, $\mu_1 \in [\![P_1]\!]_G$, $\mu_2 \in [\![P_2]\!]_G$ and $(\mu_1 \cup \mu_2) \models F_1 \cup F_2$. This means that, in particular, $(\mu_1 \cup \mu_2) \models F_1$.

Furthermore, since $\mathrm{vars}(F_1) \subseteq \mathrm{dom}(\mu_1)$, then we have that $\mu_1 \models F_1$, and therefore $\mu_1 \in [\![\mathrm{TR}(P_1, F_1)]\!]_G$. By using the same argument, $\mu_2 \models F_2$, and then $\mu_2 \in [\![\mathrm{TR}(P_2, F_2)]\!]_G$. Thus, since $\mu = \mu_1 \cup \mu_2$, we have shown that $\mu \in [\![\mathrm{TR}(P_1, F_1)]\!]_G \bowtie [\![\mathrm{TR}(P_2, F_2)]\!]_G$.

This proves the claim.

Let $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ be a quasi well-designed pattern tree. We denote by $\mathcal{T}_n$ the subtree of $\mathcal{T}$ rooted at node n. From now on in this proof, whenever we say that a pattern tree $\mathcal{T}_1$ is a subtree of a pattern tree $\mathcal{T}_2$ we assume that both trees coincide in its root node and that all the labels in $\mathcal{T}_1$ are the same as the labels of $\mathcal{T}_2$ (for the nodes that are composing the subtree $\mathcal{T}_1$).

Now let G be an RDF graph. We next show that for every node $n \in V$ the following property holds. Let M be a set of mappings, and assume that there is a set of triple patterns P_M and a set of built-in conditions F_M, such that $\mathrm{vars}(F_m) \subseteq \mathrm{vars}(P_M)$ and $M = [\![\mathrm{TR}(P_M, F_M)]\!]_G$. Moreover, assume that if $?X$ is a variable that occurs in two different descendants of n but not in P_n, then $?X \in \mathrm{dom}(\mu)$ for every $\mu \in M$ (and thus $?X$ occurs in P_M). Finally, we denote by $\mathcal{T}_n^{P_M, F_M}$ the pattern tree obtained from $\mathcal{T}_n$ by adding all triples in P_M to the triple pattern label of n (the root of $\mathcal{T}_n$), and all the built-in conditions in F_M to the label of built-in conditions of n (that is, the new labels of the root are $P_n \cup P_M$ and $F_n \cup F_M$) . We claim that

$$\mathrm{ext}(M, n, G) \;=\; [\![\mathcal{T}_n^{P_M, F_M}]\!]_G$$

We show this by induction in the tree $\mathcal{T}_n$. If n is a leaf node, then $\mathcal{T}_n$ is composed of a single node labeled P_n and F_n, and then $\mathrm{ext}(M, n, G) = M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G$. On the other hand $[\![\mathcal{T}_n^{P_M, F_M}]\!] = [\![\mathrm{TR}(P_n \cup P_M, F_n \cup F_M)]\!]_G$. By applying Claim 1, then this is equal to $[\![\mathrm{TR}(P_M, F_M)]\!]_G \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G = M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G$, and then the property

holds. Assume now that n has $n_1, \ldots, n_k$ as children. Then in this case we have that

$$\begin{aligned}\mathrm{ext}(M, n, G) \;=\; & \big((M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G) \rtimes \mathrm{ext}(M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G), n_1, G)\big) \\ & \bowtie \cdots \bowtie \big((M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G) \rtimes \mathrm{ext}(M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G), n_k, G)\big).\end{aligned}$$

Assume that $M' = M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G$. It is not difficult to see that for every n_i, if $?X$ occurs in two different descendants u and v of n_i but not in P_{n_i}, then $?X \in \mathrm{dom}(\mu)$ for every $\mu \in M'$. This is because u and v are also descendants of n, and thus variable $?X$ is either in P_n or in $\mathrm{dom}(\mu)$ for every $\mu \in M$. Moreover, we have that $M' = [\![\mathrm{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G$. Thus we can apply the induction hypothesis, and then we have that $\mathrm{ext}(M \bowtie [\![\mathrm{TR}(P_n, F_n)]\!]_G, n_i, G) = [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$, and thus, we can write $\mathrm{ext}(M, n, G)$ as

$$\begin{aligned}\mathrm{ext}(M, n, G) \;=\; & \big([\![\mathrm{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_1}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big) \bowtie \\ & \cdots \bowtie \big([\![P_M \cup P_n]\!]_G \rtimes [\![\mathcal{T}_{n_k}^{P_M \cup P_n}]\!]_G\big).\end{aligned}$$

Thus, in order to prove what we need, it is enough to show that

$$\begin{aligned}[\![\mathcal{T}_n^{P_M, F_M}]\!]_G \;=\; & \big([\![\mathrm{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_1}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big) \bowtie \\ & \cdots \bowtie \big([\![\mathrm{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_k}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big). \quad \text{(A.9)}\end{aligned}$$

Before proving this we observe that, although $\mathcal{T}_n$ can be a pattern which is not quasi well-designed, the properties of M ensures that $\mathcal{T}_n^{P_M}$ is quasi well-designed. Similarly $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ is quasi well-designed for every $i \in \{1, \ldots, k\}$. We now prove (A.9). Thus assume that $\mu \in [\![\mathcal{T}_n^{P_M, F_M}]\!]_G$. Then since $\mathcal{T}_n^{P_M, F_M}$ is quasi well-designed, by Lemma 3.3 we know that there exists a subtree $\mathcal{T}'$ of $\mathcal{T}_n^{P_M, F_M}$ such that $\mu \in [\![\mathcal{T}']\!]_G$, and μ is maximal, i.e. there is no other subtree $\mathcal{T}''$ such that μ is strictly subsumed by a mapping in $[\![\mathcal{T}'']\!]_G$. Consider a maximal such subtree $\mathcal{T}'$ for μ. First notice that $\mathcal{T}'$ is composed of the root of $\mathcal{T}_n^{P_M, F_M}$ plus (possibly empty) subtrees of the $\mathcal{T}_{n_i}$'s as children of the root of $\mathcal{T}_n^{P_M, F_M}$.

Thus, assume that $\mathcal{T}'$ is composed of the root of $\mathcal{T}_n^{P_M,F_M}$ plus trees $\mathcal{T}'_1$, $\mathcal{T}'_2$,...,$\mathcal{T}'_k$ as children, where every $\mathcal{T}'_i$ is either empty (in which case nothing is added as a child to the root of $\mathcal{T}_n^{P_M,F_M}$), or $\mathcal{T}'_i$ is a subtree of $\mathcal{T}_{n_i}$. Since $\mu \in [\![\mathcal{T}']\!]_G$ and the root of $\mathcal{T}'$ contains $P_n \cup P_M$ and $F_n \cup F_M$ as labels, we know that there exists a mapping μ' such that $\mu' \sqsubseteq \mu$ and $\mu_n \in [\![\text{TR}(P_n \cup P_M, F_n \cup F_M)]\!]_G$. Notice that this mapping μ' is unique (and has as domain, exactly the variables mentioned in $P_n \cup P_M$). We next prove some properties of the trees $\mathcal{T}'_i$ depending on whether they are empty or not.

- Given that $\mu \in [\![\mathcal{T}']\!]_G$, we have that for every $i \in \{1,\ldots,k\}$, if $\mathcal{T}'_i$ is not empty then there exists a mapping μ'_i such that $\mu'_i \sqsubseteq \mu$ and $\mu'_i \in [\![\mathcal{T}'_i]\!]_G$. For every nonempty $\mathcal{T}'_i$ consider the pattern $(\mathcal{T}'_i)^{P_M \cup P_n, F_M \cup F_n}$ constructed similarly as $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$. Then by the construction of $\mathcal{T}'$, we know that there exists a mapping $\mu' \cup \mu'_i \in [\![(\mathcal{T}'_i)^{P_M \cup P_n, F_M \cup F_n}]\!]$, with μ' the portion of μ such that $\mu' \in [\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G$. Notice that $\mu' \cup \mu'_i \sqsubseteq \mu$. We claim that $\mu' \cup \mu'_i \in [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$. On the contrary, assume that $\mu' \cup \mu'_i \notin [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$. Since $\mu' \cup \mu'_i \in [\![(\mathcal{T}'_i)^{P_M \cup P_n}, F_M \cup F_n]\!]$, $(\mathcal{T}'_i)^{P_M \cup P_n, F_M \cup F_n}$ is a subtree of $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$, and $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ is quasi well-designed, by Lemma 3.3 we know that there exists a subtree $\mathcal{T}''_i$ of $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ and a mapping ν_i such that $\mu' \cup \mu'_i \sqsubset \nu_i$ and $\nu_i \in [\![\mathcal{T}''_i]\!]$. Since $\mu' \cup \mu'_i \sqsubset \nu_i$ we know that there exists a variable $?Y \in \text{dom}(\nu_i)$ such that $?Y \notin \text{dom}(\mu' \cup \mu'_i)$. Moreover for every such variable $?Y$ that is in $\text{dom}(\nu_i)$ but not in $\text{dom}(\mu' \cup \mu'_i)$, we have that $?Y$ does not occur in any other branch of $\mathcal{T}_n^{P_M,F_M}$ since $\mathcal{T}_n^{P_M,F_M}$ is quasi well-designed. In particular, $?Y$ does not occur in any $\mathcal{T}_{n_j}$ for $j \neq i$, and then we have that $?Y \notin \text{dom}(\mu)$. Moreover, since $\mu' \cup \mu'_i \sqsubseteq \mu$ and all the variables that are in $\text{dom}(\nu_i)$ but not in $\text{dom}(\mu' \cup \mu'_i)$ are not in $\text{dom}(\mu)$, we have that μ and ν_i are compatibles, and then $\mu \sqsubset \mu \cup \nu_i$. Furthermore, since $\nu_i \in [\![\mathcal{T}''_i]\!]$, we have that there exists a subtree $\mathcal{T}''$ of $\mathcal{T}_n^{P_M,F_M}$ such that $\mu \sqsubset \mu \cup \nu_i \in [\![\mathcal{T}'']\!]$. This is a contradiction with the maximality of μ.

- Now assume that $\mathcal{T}_i'$ is empty. We show next that there is no mapping ν_i compatible with μ such that $\nu_i \in [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$. On the contrary, assume that there is a mapping ν_i compatible with μ such that $\nu_i \in [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$. Given that $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ is quasi well-designed from Lemma 3.3 we obtain that there exists a subtree $\mathcal{T}_i'$ of $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ such that $\nu_i \in [\![\mathcal{T}_i']\!]_G$. Recall that $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ is constructed from $\mathcal{T}_{n_i}$ by adding $P_M \cup P_n$ and $F_M \cup F_n$ to the labels of the root. Thus, we have that $\nu_i = \nu_i' \cup \nu_i''$ such that $\nu_i' \in [\![P_M \cup P_n, F_M \cup F_n]\!]_G$ and $\nu_i'' \in [\![\mathcal{T}_i'']\!]$ where $\mathcal{T}_i''$ is the tree obtained from $\mathcal{T}_i'$ deleting $P_M \cup P_n$ and $F_M \cup F_n$ from its root. Then $\mathcal{T}_i''$ is a subtree of $\mathcal{T}_{n_i}$. Consider now the tree $\mathcal{T}''$ obtained from $\mathcal{T}'$ by adding $\mathcal{T}_i''$ as a child to the root of $\mathcal{T}'$. Then we have that $\mathcal{T}''$ is a subtree of $\mathcal{T}_n^{P_M}$ and that $\mu \cup \nu_i \in [\![\mathcal{T}'']\!]_G$. Thus, if $\mu \cup \nu_i \neq \mu$ we obtain a contradiction with the maximality of μ, and if $\mu \cup \nu_i = \mu$ we obtain a contradiction with the maximality of $\mathcal{T}'$. Thus, we have shown that there is no mapping ν_i compatible with μ such that $\nu_i \in [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$.

Let μ' the portion of μ such that $\mu' \in [\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G$. Summarizing we have shown that:

- for every $\mathcal{T}_i'$ which is not empty, there exists a portion of μ, say μ_i' such that $\mu' \cup \mu_i' \in [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$, and thus $\mu' \cup \mu_i' \in \big([\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big)$, and
- for every $\mathcal{T}_i'$ which is empty, we have that μ is not compatible with any mapping in $[\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$, implying that μ' is not compatible with any mapping in $[\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$, and thus $\mu' \in [\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_k}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$.

From this we obtain that μ can be written as $\mu = \mu_1 \cup \mu_2 \cup \cdots \cup \mu_k$ such that $\mu_i \in [\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_k}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$, which implies that μ is in $\big([\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_1}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big) \bowtie \cdots \bowtie \big([\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_k}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big)$.

For the opposite direction, if we assume that μ is in $\big([\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_1}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big) \bowtie \cdots \bowtie \big([\![\text{TR}(P_M \cup P_n, , F_M \cup F_n)]\!]_G \rtimes [\![\mathcal{T}_{n_k}^{P_M \cup P_n, F_M \cup F_n}]\!]_G\big)$, then

$\mu = \mu_1 \cup \cdots \mu_k$ with $\mu_i \in [\![\text{TR}(P_M \cup P_n, F_M \cup F_n)]\!]_G \bowtie [\![\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}]\!]_G$. By using an argument similar to the one used in the previous case, and using the fact that every $\mathcal{T}_{n_i}^{P_M \cup P_n, F_M \cup F_n}$ is quasi well-designed and Lemma 3.3, it is not difficult to conclude that μ is in $[\![\mathcal{T}_n^{P_M}]\!]_G$.

To conclude the proof of the Theorem, just observe that for the pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V}, (F_n)_{n \in V})$ the set $[\![\mathcal{T}]\!]_G^{\text{td}}$ is defined as $\text{ext}(\{\mu_\emptyset\}, r, G)$, which by the property shown before (and since $\mathcal{T}$ is quasi well-designed) is equal to $[\![\mathcal{T}_r^\emptyset]\!]_G = [\![\mathcal{T}]\!]_G$. This completes the proof of the theorem. □

A.5. Proof of Theorem 3.3

PROOF. Before stating the proofs, we make a short note on the notation. In the following, we always use $\mathcal{T} = ((V, E, r), \mathcal{P}, \mathcal{F})$ with $\mathcal{P} = (P_n)_{n \in V}$ and $\mathcal{F} = (F_n)_n n \in V$ to denote the QWDPT before the rule application, and $\mathcal{T}' = ((V', E', r), \mathcal{P}', \mathcal{F}')$ with $\mathcal{P}' = (P'_n)_{n \in V'}$ to denote the result of the rule application. We will prove the correctness of each rule separately.

(R1) Assume that triple t was deleted from P_n for some node $n \in V$, because t occurs also in $P_{\hat{n}}$ of some ancestor $\hat{n}$ of n. Note that V' and V can differ at most by n, and this only in the case that $P_n = \{t\}$. First of all it is easy to see that $\mathcal{T}'$ is quasi well-designed if $\mathcal{T}$ is. If n was a common ancestor of two nodes sharing some variable in $\text{vars}(t)$, then also $\hat{n}$ is a common ancestor of those two nodes. Additionally, the deletion of t from n cannot break the safeness of the node because of the definition of rule R1.

Hence it only remains to show that $\mathcal{T} \equiv \mathcal{T}'$. Towards this goal, we will show that for every subtree $\mathcal{T}_1$ of $\mathcal{T}$, there exists a subtree $\mathcal{T}'_1$ of $\mathcal{T}'$ s.t. $\text{redux}(\mathcal{T}_1) \equiv \text{redux}(\mathcal{T}'_1)$, and vice versa. From this and Lemma 3.3, the desired result follows easily.

⇒) Let $\mathcal{T}_1 = ((V_1, E_1, r), \mathcal{P}_1, \mathcal{F}_1)$ be a subtree of $\mathcal{T}$. Now if $n \in V_1$ but $n \notin V'$, let $V'_1 = V_1 \setminus \{n\}$, otherwise let $V'_1 = V_1$. Then define $T'_1 = T'[V'_1]$, where $T' = (V', E', r)$, and finally $\mathcal{T}'_1 = (T'_1, (P'_n)_{n \in V'_1})$. We distinguish two cases.

- If $n \notin V_1$, it follows immediately that $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$, as actually $\mathcal{T}_1 = \mathcal{T}_1'$, and therefore $\mathrm{redux}(\mathcal{T}_1) \equiv \mathrm{redux}(\mathcal{T}_1')$.
- If $n \in V_1$, note that independent of $n \in V'$ or $n \notin V'$, the only triple pattern by which $\mathcal{T}_1$ and $\mathcal{T}_1'$ may differ is t. However, since $n \in V_1$, also $\hat{n} \in V_1$, and therefore by definition also $\hat{n} \in V_1'$. Since $t \in P_{\hat{n}}$ it follows that t is still contained in $\mathrm{redux}(\mathcal{T}_1')$, hence $\mathrm{redux}(\mathcal{T}_1') \equiv \mathrm{redux}(\mathcal{T}_1)$ holds.

This concludes the case.

$\Leftarrow$) Let $\mathcal{T}_1' = ((V_1', E_1', r), \mathcal{P}_1', \mathcal{F}_1')$ be a subtree of $\mathcal{T}'$. We distinguish two cases (whether $n \in V'$ or not), and for each of these cases we consider again two possibilities:

- $n \in V'$: Then we distinguish the case that $n \in V_1'$ from the case $n \notin V_1'$. If $n \notin V_1'$, then obviously $\mathcal{T}_1'$ is a subtree of $\mathcal{T}$ as well, and we just define $\mathcal{T}_1 = \mathcal{T}_1'$, and the required $\mathrm{redux}(\mathcal{T}_1) \equiv \mathrm{redux}(\mathcal{T}_1')$ follows trivially.
  In the case that $n \in V_1'$, then define $\mathcal{T}_1$ as $\mathcal{T}_1 = ((V_1', E_1', r), (P_n)_{n \in V_1'}, (F_n)_{n \in V_1'})$. Note that $P_{n'} = P_{n'}'$ for all $n' \in V = V'$ except for n, and $F_{n'} = F_{n'}'$ for all $n' \in V = V'$. And for n, $P_n = P_n' \cup \{t\}$. Hence $\mathrm{redux}(\mathcal{T}_1') = \mathrm{redux}(\mathcal{T}_1)$ follows immediately, since t is already contained in $P_{\hat{n}}$, and $\hat{n} \in V_1'$ whenever $n \in V_1'$. This concludes the case.
- $n \notin V'$: In this case we have to distinguish if some child n' of n (in $\mathcal{T}$) is in V_1', or not. In the latter case, i.e. if no such $n' \in V_1'$, then $\mathcal{T}_1'$ is also a subtree of $\mathcal{T}$, and we are done. Now assume that some child n' is in V_2'. Then we define the subtree $\mathcal{T}_1 = (T_1, (P_n)_{n \in V(T_1)}, (F_n)_{n \in F(T_1)})$ where $T_1 = T[V_1' \cup \{n\}]$ (with $T = (V, E, r)$). Since $\hat{n} \in V_1'$ whenever $n' \in V_1'$, we know that t already occurs in $\mathrm{redux}(\mathcal{T}_1')$. Hence we get that $\mathrm{redux}(\mathcal{T}_1) \equiv \mathrm{redux}(\mathcal{T}_1')$, which concludes the proof.

We thus have shown that for every subtree $\mathcal{T}_1$ of $\mathcal{T}$ there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mathrm{redux}(\mathcal{T}_1) \equiv \mathrm{redux}(\mathcal{T}_1')$, and that also for every subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mathrm{redux}(\mathcal{T}_1) \equiv \mathrm{redux}(\mathcal{T}_1')$.

To see that this implies $\mathcal{T} \equiv \mathcal{T}'$, consider an arbitrary RDF graph G and let $\mu \in [\![\mathcal{T}]\!]_G$. By Lemma 3.3, there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathcal{T}_1]\!]_G$, and no mapping ν and subtree $\mathcal{T}_2$ of $\mathcal{T}$ s.t. $\mu \sqsubset \nu$ and $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$. We have just shown above that in this case there also exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ with $\mathrm{redux}(\mathcal{T}_1') \equiv \mathrm{redux}(\mathcal{T}_1)$. Hence $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$. Further, if there would exist some ν' and subtree $\mathcal{T}_2'$ of $\mathcal{T}$ s.t. $\mu \sqsubset \nu$ and $\nu \in [\![\mathrm{redux}(\mathcal{T}_2')]\!]_G$, this would lead to a contradiction of μ being a solution to $\mathcal{T}$: By the above result, there would also exist a corresponding subtree $\mathcal{T}_2$ of $\mathcal{T}$ with $\mathrm{redux}(\mathcal{T}_2') \equiv \mathrm{redux}(\mathcal{T}_2)$. Hence by Lemma 3.3, this implies that $\mu \notin [\![\mathcal{T}]\!]_G$.

The case for $\mu \in [\![\mathcal{T}']\!]_G$ is shown analogously.

(R2) Assume that node $n \in V$ was merged into all its children $n_1, \ldots, n_k$ because of $\mathrm{newvars}(n) = \emptyset$. Then $V' = V \setminus \{n\}$. First of all it is easy to see that $\mathcal{T}'$ is quasi well-designed if $\mathcal{T}$ is: If n was a common ancestor of two nodes sharing some variable $?X \in \mathrm{vars}(P_n)$, then there must exist some ancestor $\hat{n}$ of n s.t. $?X \in \mathrm{vars}(P_{\hat{n}})$, since $\mathrm{newvars}(n) = \emptyset$. Hence $\hat{n}$ still is a common ancestor of those two nodes. Furthermore, for each child n_i of n we have that $\mathrm{vars}(F'_{n_i}) = \mathrm{vars}(F_{n_i}) \cup \mathrm{vars}(F_n)$, and $\mathrm{vars}(P'_{n_i}) = \mathrm{vars}(P_{n_i}) \cup \mathrm{vars}(P_n)$. Since $\mathrm{vars}(F_{n_i}) \subseteq \mathrm{vars}(P_{n_i})$, it follows that $\mathrm{vars}(F'_{n_i}) \subseteq \mathrm{vars}(P'_{n_i})$.

It therefore remains to show that $\mathcal{T}_1 \equiv \mathcal{T}_2$. Towards this goal, we make use of the following claims:

- *Claim 1:* Let G be an arbitrary RDF graph. If μ is a variable binding s.t. there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ with $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$, then there also exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$.
- *Claim 2:* Let G be an arbitrary RDF graph. If μ is a variable binding s.t. there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ with $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$, then there also exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$.

From these claims, the desired result follows immediately: Given some $\mu \in [\![\mathcal{T}]\!]_G$ for some arbitrary RDF graph G, we know by Lemma 3.3 that there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$, and that there does not exist a mapping ν and subtree $\mathcal{T}_2$ of $\mathcal{T}$ s.t.

$\mu \sqsubset \nu$ and $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$. We further know because of Claim 1 that there also exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$, s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$. On the other hand, if there would exist a mapping ν with $\mu \sqsubset \nu$ and a subtree $\mathcal{T}_2'$ of $\mathcal{T}'$ s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2')]\!]_G$, then by Claim 2 we know that there would also exist a corresponding subtree $\mathcal{T}_2$ of $\mathcal{T}$ s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$. This however would be a contradiction to the assumption that $\mu \in [\![\mathcal{T}]\!]_G$. Hence properties (1) and (2) of Lemma 3.3 also hold for μ w.r.t. $\mathcal{T}'$, which proves that $\mu \in [\![\mathcal{T}']\!]_G$.

The case that for every $\mu \in [\![\mathcal{T}']\!]_G$ also $\mu \in [\![\mathcal{T}]\!]_G$ holds is shown by using the symmetric arguments. Hence it only remains to prove the two claims:

Proof of Claim 1: Let μ be the mapping mentioned in the claim and $\mathcal{T}_1$ be a subtree of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$. Let n be the node merged into its children by R2. Then consider $\mathcal{T}_1'$ as subtree of $\mathcal{T}'$ defined as follows:

- Assume $n \notin V_1$. Then $\mathcal{T}_1' = \mathcal{T}_1$. Obviously $\mathcal{T}_1'$ is a valid subtree of $\mathcal{T}'$, and also $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$. Hence the desired result follows immediately.
- Assume $n \in V_1$ but no child n' of n is contained in V_1. Then, denoting with T' the tree $T' = (V', E', r)$ and with V_1' the set $V_1' = V_1 \setminus \{n\}$, let $\mathcal{T}_1' = (T'[V_1'], (P_n')_{n \in V_1'})$. Then $\bigcup_{n \in V_1'} P_n' \subseteq \bigcup_{n \in V_1} P_n$, while $\bigcup_{n \in V_1'} \mathrm{vars}(P_n') = \bigcup_{n \in V_1} \mathrm{vars}(P_n)$. Hence $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$ holds.
- Assume $n \in V_1$ and at least one child n' of n is also contained in V_1. We define $\mathcal{T}_1'$ as in the previous case, i.e. denote with T' the tree $T' = (V', E', r)$ and let $V_1' = V_1 \setminus \{n\}$. Then define $\mathcal{T}_1'$ as $\mathcal{T}_1' = (T'[V_1'], (P_n')_{n \in V_1'})$. Again it holds that $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$, since all triples $t \in P_n$ are contained in $P_{n'}'$ for every child n' of n. Hence $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$ follows from the assumption that at least one such child is contained in $\mathcal{T}_1'$.

So we finally have shown that in all three cases there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ with $\mathrm{redux}(\mathcal{T}_1') = \mathrm{redux}(\mathcal{T}_1)$. From this, the desired result follows again immediately.

Proof of Claim 2: Let μ be the mapping mentioned in the claim and $\mathcal{T}_1'$ be a subtree of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$. Further let $n \in V$ be the node merged with its children by

R2, and denote with $n_1, \ldots, n_k$ the children of n in V. Then consider $\mathcal{T}_1$ as subtree of $\mathcal{T}$ defined as follows:

- If no $n_i \in V_1'$ (for $i \in \{1, \ldots, k\}$), then define $\mathcal{T}_1$ as $\mathcal{T}_1 = \mathcal{T}_1'$. Obviously this is a valid subtree of $\mathcal{T}$, and $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$ trivially holds. Hence the desired result follows immediately.
- If $n_i \in V_1'$ for some $i \in \{1, \ldots, k\}$, then let $T = (V, E, r)$ and $V_1 = V_1' \cup \{n\}$. Define $\mathcal{T}_1$ as $\mathcal{T}_1 = (T[V_1], (P_n)_{n \in V_1})$. Again, $\mathrm{redux}(\mathcal{T}_1') = \mathrm{redux}(\mathcal{T}_1)$: Note that $P_n \subseteq P'_{n_i'}$ holds for every $i \in \{1, \ldots, k\}$. In fact, $P'_{n_i} = P_{n_i} \cup P_n$, while for all nodes $\bar{n} \in V$ with $\bar{n} \neq n_i$ it holds that $P_{\bar{n}} = P'_{\bar{n}}$. Hence $\mathrm{redux}(\mathcal{T}_1') = \mathrm{redux}(\mathcal{T}_1)$ holds again, which proves the case.

We have shown that in both cases there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ with $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$, from which the desired result follows immediately. Hence this concludes the proof.

(R3) Assume that node $n \in V$ was merged into its parent $\hat{n}$ due to some homomorphism $h\colon P_n \to P_{\mathrm{branch}(\hat{n},\mathcal{T})}$, with $h(?X) = ?X$ for all $?X \in \mathrm{vars}(P_n) \cap \mathrm{vars}(P_{\mathrm{branch}(\hat{n},\mathcal{T})})$. Note that $V' = V \setminus \{n\}$, $P_{n'} = P'_{n'}$ for all $n' \in V \setminus \{n, \hat{n}\}$, and $P'_{\hat{n}} = P_n \cup P_{\hat{n}}$. Furthermore, note that since $\mathrm{vars}(F_n) \cup \mathrm{vars}(\mathrm{branch}(\hat{n}, \mathcal{T})) = \emptyset$, then every variable $?X$ appearing in F_n must not appear in any descendant n' of $\hat{n}$ which is not n or a descendant of n, since $?X$ would not appear in any common ancestor of n and n'. Therefore, the addition of F_n to $F_{\hat{n}}$ does not change the evaluation of $\mathcal{T}$. Thus, we can proceed with the proof as if $F_n = \emptyset$.

First of all it is easy to see that $\mathcal{T}'$ is quasi well-designed if $\mathcal{T}$ is. If n was the common ancestor for two nodes sharing some variable, then $\hat{n}$ is now an ancestor of both of these nodes, and $P_n \subseteq P'_{\hat{n}}$. Hence it only remains to show that $\mathcal{T} \equiv \mathcal{T}'$. This proof is based on the crucial property underlying R3:

- *Claim 1:* Let $\mathcal{T}$ be as defined above, and G an arbitrary RDF graph. Let μ be a mapping such that there exists a subtree $\mathcal{T}_1 = ((V_1, E_1, r), (P_n)_{n \in V_1}, (F_n)_{n \in V_1})$

of $\mathcal{T}$ with $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$. Then the following holds: If $\hat{n} \in V_1$, then there exists a subtree $\mathcal{T}_2 = ((V_2, E_2, r), (P_n)_{n \in V_2}, (F_n)_{n \in V_2})$ of $\mathcal{T}$ with $\hat{n}, n \in V_2$ s.t. either

- $\mu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$ (i.e. $\mu(P_n) \subseteq G$ already holds) or
- there exists an assignment ν with $\mu \sqsubseteq \nu$ s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$.

Using this claim, we show two claims similar to those already used to prove the correctness of R2:

- *Claim 2:* Let G be an arbitrary RDF graph. If $\mu \in [\![\mathcal{T}]\!]_G$ then there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$.
- *Claim 3:* Let G be an arbitrary RDF graph. If $\mu \in [\![\mathcal{T}']\!]_G$, then there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$.

Combining Claim 2 and Claim 3 allows us to conclude the following: Let $\mu \in [\![\mathcal{T}]\!]_G$ for an arbitrary RDF graph G. Then we know from Lemma 3.3 that there exists some subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$, and that there does not exist a mapping ν with $\mu \sqsubset \nu$ s.t. we can find a subtree $\mathcal{T}_2$ of $\mathcal{T}$ s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$. Now from Claim 1 we know that there also exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$. Hence if μ is no solution to $\mathcal{T}'$, then (by Lemma 3.3) this can only be because there exists some ν with $\mu \sqsubset \nu$ s.t. for some subtree $\mathcal{T}_2'$ of $\mathcal{T}'$ it holds that $\nu \in [\![\mathrm{redux}(\mathcal{T}_2')]\!]_G$. However, then by Claim 3 there must also exist a corresponding subtree $\mathcal{T}_2$ of $\mathcal{T}$ s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$. This contradicts the assumption that μ is a solution to $\mathcal{T}$, and therefore proves the case, i.e. that μ is also a solution to $\mathcal{T}'$.

The proof in the other direction (i.e. showing that every solution μ to $\mathcal{T}'$ is also a solution to $\mathcal{T}$) works by the symmetric arguments. Hence to finish the proof it only remains to show the correctness of the Claims.

Proof of Claim 1: Consider $\mathcal{T}$, $\mathcal{T}_1$, G, and μ as defined in the Claim. Further assume that $\hat{n} \in V_1$. We distinguish two cases:

- $\mathrm{newvars}(n) = \emptyset$. We show that then $\mu(P_n) \subseteq G$ already holds, i.e. $\mathcal{T}_2 = ((V_2, E_2, r), (P_n)_{n \in V_2}, (F_n)_{n \in V_2})$, where $V_2 = V_1 \cup \{n\}$ and $E_2 = E_1 \cup \{(\hat{n}, n)\}$. To see that this indeed holds, recall that $h(?X) = ?X$ for all $?X \in \mathrm{vars}(P_n) \cap \mathrm{vars}(P_{\mathrm{branch}(\hat{n},\mathcal{T})})$. Since $\mathrm{newvars}(n) = \emptyset$ we have $\mathrm{vars}(P_n) \cap \mathrm{vars}(P_{\mathrm{branch}(\hat{n},\mathcal{T})}) = \mathrm{vars}(P_n)$, hence for all $t \in P_n$ it is the case that $h(t) = t$. Since further $h(t) \in P_{\mathrm{branch}(\hat{n},\mathcal{T})}$ by definition of h and for all $s \in P_{\mathrm{branch}(\hat{n},\mathcal{T})}$ we have $\mu(s) \in G$ by assumption, it follows immediately that $\mu(h(t)) = \mu(t) \in G$ for all $t \in P_n$. This finishes the case.
- $\mathrm{newvars}(n) \neq \emptyset$. Obviously, if $n \in V_1$ the claim is already satisfied, hence the only interesting case is $n \notin V_1$. Since $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$, if $n \notin V_1$ then μ cannot be defined on $\mathrm{newvars}(n)$. Hence to prove the claim we show that there exists an extension ν of μ to all variables $?X \in \mathrm{newvars}(n)$ s.t. $\nu(P_{n'}) \subseteq G$ for all $n' \in V_1 \cup \{n\}$. I.e. $\mathcal{T}_2 = ((V_2, E_2, r), (P_n)_{n \in V_2})$ with $V_2 = V_1 \cup \{n\}$ and $E_2 = E_1 \cup \{(\hat{n}, n)\}$ is a QWDPT s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2)]\!]_G$ as promised in the claim.

  Towards this goal, define ν as $\nu(?X) = \mu(h(?X))$ for all variables $?X \in \mathrm{vars}(P_n)$, and $\nu(?X) = \mu(?X)$ for all variables $?X \in \mathrm{dom}(\mu) \setminus \mathrm{vars}(P_n)$. Then ν is obviously well defined. Further, ν is a proper extension of μ: For all $?X \in \mathrm{vars}(P_n) \cap \mathrm{dom}(\mu)$ it must be the case that also $?X \in \mathrm{vars}(P_n) \cap \mathrm{vars}(P_{\mathrm{branch}(\hat{n},\mathcal{T})})$ because $\mathcal{T}$ is quasi well-designed. Hence $h(?X) = ?X$ for all those variables $?X$, and therefore the two definitions of ν agree on those variables.

  It only remains to show that $\nu(P_n) \subseteq G$ holds. First note that $h(t) \in P_{\mathrm{branch}(\hat{n},\mathcal{T})}$ holds for all $t \in P_n$. Next, for all $s \in P_{\mathrm{branch}(\hat{n},\mathcal{T})}$ we have that $\mu(s) \in G$. As a consequence, $\mu(h(t)) \in G$ holds for all $t \in P_n$, and therefore $\nu(t) = \mu(h(t)) \subseteq G$ for all $t \in P_n$. This proves the claim.

Proof of Claim 2: Let $\mu \in [\![\mathcal{T}]\!]_G$. Then we know from Lemma 3.3 that there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$. Let $\mathcal{T}_1 = ((V_1, E_1, r), \mathcal{P}_1)$. By combining property

(2) of Lemma 3.3 with Claim 1, we also know that we can choose $\mathcal{T}_1$ in such a way that either $\hat{n} \in V_1$ and $n \in V_1$ or $\hat{n} \notin V_1$ and $n \notin V_1$. Hence we distinguish two cases:

- $\hat{n} \notin V_1$. Then just define $\mathcal{T}_1'$ as $\mathcal{T}_1' = \mathcal{T}_1$. Obviously $\mathcal{T}_1'$ is a valid subtree of $\mathcal{T}'$. Then trivially $\mathrm{redux}(\mathcal{T}_1') = \mathrm{redux}(\mathcal{T}_1)$, and the desired result follows.
- $\hat{n}, n \in V_1$. Then define $V_1' = V_1 \setminus \{n\}$, and let T' be $T' = (V', E', r)$. We define $\mathcal{T}_1'$ as $(T'[V_1'], (P_n')_{n \in V_1'})$. Hence, V_1 and V_1' differ only by n. However, $P_n \subseteq P_{\hat{n}}'$, and because $P_{n'} = P_{n'}'$ for all $n' \in V_1$ with $n' \neq n$ and $n' \neq \hat{n}$, it follows immediately that $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$. From this, the desired result follows immediately.

Proof of Claim 3: Let $\mu \in [\![\mathcal{T}']\!]_G$. Then by Lemma 3.3 there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mu \in [\![\mu(\mathcal{T}_1')]\!]_G$. Let $\mathcal{T}_1' = ((V_1', E_1', r), \mathcal{P}_1')$. Then we distinguish two cases:

- $\hat{n} \notin V_1'$. Then obviously $\mathcal{T}_1$ defined as $\mathcal{T}_1 = \mathcal{T}_1'$ is also a subtree of $\mathcal{T}$, and $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$.
- $\hat{n} \in V_1'$. Then let $V_1 = V_1' \cup \{n\}$ and denote with T the tree $T = (V, E, r)$. Now define $\mathcal{T}_1 = (T[V_1], (P_n)_{n \in V_1})$. Note that $P_{\hat{n}}' = P_{\hat{n}} \cup P_n$, and $P_{n'}' = P_{n'}$ for all $n' \in V$ with $n' \neq n$ and $n' \neq \hat{n}$. Hence, it again follows that $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$, which proves the case, and therefore finishes the proof.

(R4) For $\hat{n}, n, \bar{n} \in V$ let $\hat{n}$ be the parent of n, and n the parent of $\bar{n}$, s.t $\bar{n}$ was transformed into a child of $\hat{n}$ by one application of R4. Let $h\colon P_n \to P_{\mathrm{branch}(\hat{n}, \mathcal{T})} \cup P_{\bar{n}}$ be the homomorphism that allows to apply R4. Finally, note that $V' = V$, $\mathcal{F}' = \mathcal{F}$ and $\mathcal{P}' = \mathcal{P}$. First of all $\mathcal{T}'$ is quasi well-designed because of the definition of R4 that only allows its application if the result is quasi well-designed. Hence it only remains to show that $\mathcal{T} \equiv \mathcal{T}'$. This proof is based on the following crucial property underlying R4:

- *Claim 1:* Let $\mathcal{T}'$ be as defined above. Then for every RDF graph G and every mapping μ s.t. there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ with $\mu \in [\![\mathrm{redux}(\mathcal{T}')]\!]_G$, the following holds: If $\bar{n} \in V(\mathcal{T}_1')$ and $n \notin V(\mathcal{T}_1')$, then there exists a mapping ν

with $\mu \sqsubseteq \nu$ and a subtree $\mathcal{T}_2' = ((V_2', E_2', r), (P_n')_{n \in V_2'})$ of $\mathcal{T}'$ s.t. $n \in V_2'$, $\mathcal{T}_1'$ is a subtree of $\mathcal{T}_2'$, and $\nu \in [\![\mathrm{redux}(\mathcal{T}_2')]\!]_G$.

Note that the above claim includes the case that $\nu = \mu$, e.g. if $\mathrm{vars}(P_n) \backslash \mathrm{vars}(P_{\mathrm{branch}(\bar{n},\mathcal{T}')}) = \emptyset$. Also, the above claim implies (together with Lemma 3.3) that there cannot exist a solution $\mu \in [\![\mathcal{T}']\!]_G$ s.t. $\mu(P_{\mathrm{branch}(\bar{n},\mathcal{T}')}) \subseteq G$, but $\mu(P_{\mathrm{branch}(n,\mathcal{T}')}) \nsubseteq G$. Using this property, we can show the same claims as in the proof for the correctness of R3:

- *Claim 2:* Let G be an arbitrary RDF graph. If $\mu \in [\![\mathcal{T}]\!]_G$ then there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$.
- *Claim 3:* Let G be an arbitrary RDF graph. If $\mu \in [\![\mathcal{T}']\!]_G$, then there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$.

Again, these results imply that $\mathcal{T} \equiv \mathcal{T}'$. For an arbitrary RDF graph G, consider $\mu \in [\![\mathcal{T}]\!]_G$. Then by Lemma 3.3 there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$. Hence by Claim 2 there exists a corresponding subtree $\mathcal{T}_1'$ of $\mathcal{T}'$, s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$. Further, if there would exist some mapping ν with $\mu \sqsubset \nu$ and a subtree $\mathcal{T}_2'$ of $\mathcal{T}'$ s.t. $\nu \in [\![\mathrm{redux}(\mathcal{T}_2')]\!]_G$, then by Claim 3 there would also exist a corresponding subtree $\mathcal{T}_2$ of $\mathcal{T}$. However, this gives a contradiction to the assumption that μ is a solution to $\mathcal{T}$. Hence such a ν cannot exist, and therefore (by Lemma 3.3), μ is also a solution to $\mathcal{T}'$. The case for $\mu \in [\![\mathcal{T}']\!]_G$ is shown by the symmetric arguments. Hence it only remains to prove the correctness of the three Claims.

Proof of Claim 1: Let $\mathcal{T}'$ be as defined in the claim, let G be an arbitrary RDF graph, and $\mathcal{T}_1' = ((V_1', E_1', r), (P_n')_{n \in V_1'})$ a subtree of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$ holds. Now assume that $\bar{n} \in V_1'$ and $n \notin V_1'$. Then we define ν as follows: For all $?X \in \mathrm{vars}(P_n)$, let $\nu(?X) = \mu(h(?X))$, and for all $?X \in \mathrm{dom}(\mu)$ let $\nu(?X) = \mu(?X)$. First of all, note that ν is well-defined, since for $?X \in \mathrm{dom}(\mu) \cap \mathrm{vars}(P_n)$, due to $\mathcal{T}'$ being quasi well-designed and the definition of R4, it holds that $h(?X) = ?X$, hence $\nu(?X) = \mu(h(?X)) = \mu(?X)$. Further, it obviously holds for all $n' \in V_1'$ that $\nu(P_{n'}) \subseteq G$. To see that this also holds for P_n, just note that $h(t) \in P_{\mathrm{branch}(\bar{n},\mathcal{T})}$ for all $t \in P_n$. Thus $\mu(h(t)) \in G$ holds. Hence for

$\mathcal{T}_2' = ((V_1' \cup \{n\}, E_1' \cup \{(\hat{n}, n)\}, r), (P'_{n'})_{n' \in V_1' \cup \{n\}})$ it holds that $\nu \in [\![\mathrm{redux}(\mathcal{T}_2')]\!]_G$, which proves the claim.

Proof of Claim 2: Let $\mu \in [\![\mathcal{T}]\!]_G$. Then by Lemma 3.3 there exists a subtree $\mathcal{T}_1$ of $\mathcal{T}$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$. Let $\mathcal{T}_1 = ((V_1, E_1, r), (P_n)_{n \in V_1})$. Denote with T' the tree $T' = (V', E', r)$. Then it is easy to check that $T_1' = T'[V_1]$ is indeed a connected subtree of T'. Hence we get $\mathcal{T}_1' = (T_1', (P_n')_{n \in V_1})$, and trivially $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$, which proves the claim.

Proof of Claim 3: Let $\mu \in [\![\mathcal{T}']\!]_G$. Then by Lemma 3.3 there exists a subtree $\mathcal{T}_1'$ of $\mathcal{T}'$ s.t. $\mu \in [\![\mathrm{redux}(\mathcal{T}_1')]\!]_G$. Let $\mathcal{T}_1' = ((V_1', E_1', r), (P_n')_{n \in V_1})$. We already discussed above that it follows immediately from Claim 1 and Lemma 3.3 that whenever $\bar{n} \in V_1'$, then also $n \in V_1'$, as otherwise μ cannot be a valid solution to $\mathcal{T}'$, since Claim 1 implies an immediate contradiction to Lemma 3.3. As a result, denoting with T the tree $T = (V, E, r)$, the induced graph $T[V_1']$ is always a connected subtree of T. Using this, we can define $\mathcal{T}_1$ as $\mathcal{T}_1 = (T[V_1'], (P_n)_{n \in V_1'})$. Hence $\mathrm{redux}(\mathcal{T}_1) = \mathrm{redux}(\mathcal{T}_1')$ trivially holds, from which also $\mu \in [\![\mathrm{redux}(\mathcal{T}_1)]\!]_G$ follows trivially. This proves the claim.

(R5) Before starting the proof, consider the following claim:

Claim 1: Let F be a set of built-in conditions, n a pattern tree node, μ a mapping such that $\mu \models \mathit{conj}(F)$ and $M = \mathrm{maxprop}(F, n)$. Then $\mu \models \mathrm{maxprop}(F, n)$.

Proof of Claim 1: Remember that for a built-in condition R, $\mathrm{maxprop}(R, n)$ is constructed from an equivalent clause R' in conjunctive normal form. Assume that $\mu \models F$ but $\mu \not\models \mathrm{maxprop}(F, n)$. Therefore, there exists some built-in condition $R \in F$ for which its equivalent built-in condition R' contains a clause c such that $\mu \not\models c$. Then $\mu \not\models R'$, and since R' is equivalent to R, $\mu \not\models R$. But since $R \in F$, we have that $\mu \not\models \mathit{conj}(F)$. Therefore, we have arrived at a contradiction, thus proving the claim.

For $n, \bar{n} \in V$ let n be the parent of $\bar{n}$, and assume that $M = \mathrm{maxprop}(F_n, \bar{n})$ was copied onto $\bar{n}$ by applying rule R5.

First, assume that M is not empty, since otherwise $\mathcal{T}' = \mathcal{T}$ and the proof would be trivial. Note that $\mathcal{T}$ and $\mathcal{T}'$ differ at most by $\mathcal{F}_n$, and that the resulting subtree is quasi well-designed, since the rule application ensures that node $\bar{n}'$ is safe.

Let $\mathcal{T}_n$ be the subtree of $\mathcal{T}$ rooted at n, and $\mathcal{T}'_{n'}$ be the subtree of $\mathcal{T}'$ rooted at n'. Since the semantics of QWDPTs is well defined (by Theorem 3.2), we only need to show that $\mathcal{T}_n \equiv \mathcal{T}'_{n'}$.

Consider orderings Σ and Σ' for $\mathcal{T}_n$ and $\mathcal{T}'_{n'}$, which make $\sigma_n(1) = \bar{n}$ and $\sigma_{n'}(1) = \bar{n}'$, and coincide in the ordering of every other node. Therefore, if n has k children, we have

$$\begin{aligned}\textsc{Tr}(\mathcal{T}_n, n, \Sigma) \;=\; \Big(\cdots \Big(\Big(\textsc{tr}(P_n, F_n) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_n, \bar{n}, \Sigma) \Big) \;\cdots\; \Big) \\ \text{OPT}\; \textsc{Tr}(\mathcal{T}_n, \sigma_n(i), \Sigma) \Big) \;\cdots\; \Big) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_n, \sigma_n(k), \Sigma) \Big)\end{aligned}$$

and

$$\begin{aligned}\textsc{Tr}(\mathcal{T}'_{n'}, n', \Sigma') \;=\; \Big(\cdots \Big(\Big(\textsc{tr}(P_{n'}, F_{n'}) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}'_{n'}, \bar{n}', \Sigma') \Big) \;\cdots\; \Big) \\ \text{OPT}\; \textsc{Tr}(\mathcal{T}'_{n'}, \sigma_{n'}(i), \Sigma') \Big) \;\cdots\; \Big) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}'_{n'}, \sigma_{n'}(k), \Sigma') \Big).\end{aligned}$$

Now, let

$$\begin{aligned}P_1 &= \big(\textsc{tr}(P_n, F_n) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_n, \bar{n}, \Sigma)\big) \\ P_2 &= \big(\textsc{tr}(P_{n'}, F_{n'}) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}'_{n'}, \bar{n}', \Sigma')\big).\end{aligned}$$

Since $\mathcal{T}_n$ and $\mathcal{T}'_{n'}$ only differ on F_n and F'_n, we can reduce the problem to showing that $P_1 \equiv P_2$. Notice that since $P_n = P_{n'}$, $F_n = F_{n'}$, F_n is not empty and Σ coincides with Σ' for every node, then

$$\begin{aligned}P_1 &= \big((P_n \;\text{FILTER}\; \mathit{conj}(F_n)) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}_n, \bar{n}, \Sigma)\big) \\ P_2 &= \big((P_n \;\text{FILTER}\; \mathit{conj}(F_n)) \;\text{OPT}\; \textsc{Tr}(\mathcal{T}'_{n'}, \bar{n}', \Sigma)\big).\end{aligned}$$

Therefore, consider a graph G. We will show that a mapping μ is in $[\![P_1]\!]_G$ if and only if μ is in $[\![P_2]\!]_G$.

⇒) Since $\mu \in [\![P_1]\!]_G$, then either $\mu \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G \bowtie [\![\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma)]\!]_G$, or $\mu \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G$ with no compatible mapping in $[\![\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma)]\!]_G$.

In the first case we have that $\mu = \mu_1 \cup \mu_2$, with $\mu_1 \in [\![(P_n \text{ FILTER } F_n)]\!]_G$ and $\mu_2 \in [\![\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma)]\!]_G$. Since $\mu \models conj(F_n)$, we know from Claim 1 that $\mu \models \text{maxprop}(F_n, \bar{n})$. In particular, since $\text{maxprop}(F_n, \bar{n})$ mentions only the variables in $\text{vars}(\bar{n})$, we know that $\mu_2 \models \text{maxprop}(F_n, \bar{n})$. Then, $\mu_2 \in [\![(\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma) \text{ FILTER } conj(\text{maxprop}(F_n, n))]\!]_G$, and therefore,

$$\mu \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G \bowtie [\![(\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma) \text{ FILTER } conj(\text{maxprop}(F_n, n))]\!]_G.$$

From here it is easy to show that $(\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma) \text{ FILTER } conj(\text{maxprop}(F_n, n))$ is equivalent to $\text{TR}(\mathcal{T}'_{n'}, \bar{n}', \Sigma)$ by applying rewrite rule 7 from Proposition 4.10 shown in Pérez et al. (2009), thus proving that $\mu \in P_2$.

In the second case, we have that $\mu \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G$, and there is no compatible mapping $\mu_2 \in [\![(\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma)]\!]_G$. Since $\bar{n}'$ only has more built-in conditions than $\bar{n}$, by the semantics of the FILTER operator we have that there is no mapping $\mu'_2 \in [\![(\text{TR}(\mathcal{T}'_{n'}, \bar{n}', \Sigma)]\!]_G$ which is compatible with μ, and therefore, $\mu \in P_2$.

⇐) Since $\mu \in [\![P_2]\!]_G$, then either $\mu \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G \bowtie [\![\text{TR}(\mathcal{T}'_{n'}, \bar{n}', \Sigma)]\!]_G$, or $\mu \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G$ with no compatible mapping in $[\![\text{TR}(\mathcal{T}'_{n'}, \bar{n}', \Sigma)]\!]_G$.

In the first case we have that $\mu = \mu_1 \cup \mu_2$, with $\mu_1 \in [\![(P_n \text{ FILTER } conj(F_n)]\!]_G$ and $\mu_2 \in [\![\text{TR}(\mathcal{T}'_{n'}, \bar{n}', \Sigma)]\!]_G$. Since

$$\text{TR}(\mathcal{T}'_{n'}, \bar{n}', \Sigma) \equiv (\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma) \text{ FILTER } conj(\text{maxprop}(F_n, n))$$

we have that μ_2 is also in $\text{TR}(\mathcal{T}_n, \bar{n}, \Sigma)$, and therefore $\mu \in P_1$.

The second case is exactly the same as the one shown before in the opposite direction, and therefore we are done.

Thus, we have proven that $P_1 \equiv P_2$, and from here we can retrace back to arrive at $\mathcal{T} \equiv \mathcal{T}'$. $\square$