Static Analysis and Optimization of Semantic Web Queries

Andrés Letelier
Dept. of Computer Science
PUC Chile
aileteli@uc.cl

Jorge Pérez
Dept. of Computer Science
Universidad de Chile
jperez@dcc.uchile.cl

Reinhard Pichler Sebastian Skritek
Faculty of Informatics
Technische Universität Wien
[pichler,skritek]@dbai.tuwien.ac.at

ABSTRACT

Static analysis is a fundamental task in query optimization. In this paper we study static analysis and optimization techniques for SPARQL, which is the standard language for querying Semantic Web data. Of particular interest for us is the *optionality* feature in SPARQL. It is crucial in Semantic Web data management, where data sources are inherently incomplete and the user is usually interested in partial answers to queries. This feature is one of the most complicated constructors in SPARQL and also the one that makes this language depart from classical query languages such as relational conjunctive queries. We focus on the class of well-designed SPARQL queries, which has been proposed in the literature as a fragment of the language with good properties regarding query evaluation. We first propose a tree representation for SPARQL queries, called pattern trees, which captures the class of well-designed SPARQL graph patterns and which can be considered as a query execution plan. Among other results, we propose several transformation rules for pattern trees, a simple normal form, and study equivalence and containment. We also study the enumeration and counting problems for this class of queries.

Categories and Subject Descriptors
H.2.3 [**Database Management**]: Query languages

Keywords
SPARQL, RDF, Semantic Web, optimization, query containment

1. INTRODUCTION

The Semantic Web is the initiative of the World Wide Web Consortium (W3C) to make information on the Web readable not only by humans but also by machines. The *Resource Description Framework* (RDF) is the standard data model for the Semantic Web, and since its release as a W3C Recommendation in 1999 [23], the problem of managing RDF data has been in the focus of the Semantic Web community. As a result, the language SPARQL was proposed as a query language for RDF, and became a W3C Recommendation in 2008 [31]. Since the appearance of these standards, the Web has witnessed a constant growth in the amount of RDF data published on-line. Moreover, the advent of huge initiatives like Open Linked Data [6, 7] and Open Government Data [42, 43] that use RDF as a core technology, and the use of RDF in several diverse areas such as bio-informatics, social networks, and data integration, have increased the attention of the research community to study RDF and SPARQL from a database perspective.

Several particular issues of RDF and SPARQL pose new and interesting challenges for the database community [3]. In fact, several research efforts have been pursued towards understanding their fundamental properties and developing specific techniques to efficiently deal with these technologies [25, 1, 30, 40, 36, 2, 32, 27, 24, 34, 3]. Nevertheless, and despite the importance of static query analysis, and in particular of query containment and equivalence for optimization purposes, research on the static analysis of SPARQL queries has received little attention so far (notable exceptions are [35, 9]). The study of static analysis considering the peculiarities of SPARQL and, in particular, query optimization, containment and equivalence, constitute the main focus of this paper.

Let us briefly recall that the data model underlying RDF data is a (directed, arc-labeled) graph. An RDF graph is composed of RDF triples of the form (s, p, o). Regarding SPARQL, its basic constructor is the *triple pattern*, which is essentially an RDF triple that can have variables. The most basic fragment of the language are conjunctions of triple patterns, realized in SPARQL by using the AND operator (see Section 2 for a formal introduction of the language). Thus, if one thinks of RDF graphs as sets of tuples, a triple pattern is essentially a ternary relational atom, and basic SPARQL queries are essentially relational conjunctive queries (CQs). In view of this connection, the rich body of work on static analysis and query optimization on relational CQs including the study of equivalence and containment can be immediately carried over to the conjunctive fragment of SPARQL. Moreover, most of the research focused on statistics, indices, and storage optimization has been concentrated on this fragment [1, 37, 40, 36, 24].

However, when one goes beyond the SPARQL conjunctive fragment, the whole picture changes and the language becomes considerably more complicated. Of particular interest is the *optional matching* feature, which has been the focus of most of the theoretical work regarding this language [2, 27, 34, 3]. The idea behind optional matching, realized in SPARQL by the OPT operator, is to allow information to be added if the information is available in the data source instead of just failing to give an answer whenever some part of the query does not match. This feature is crucial in Semantic Web data management, where data sources are inherently incomplete and have only partial knowledge about the resources that they are modeling. Recent experimental works [15, 28] show that the use of the OPT operator in practice is substantial. For instance, in a query log obtained from the DBPedia SPARQL endpoint [44],

PODS'12, May 21–23, 2012, Scottsdale, Arizona, USA.

after duplicate query elimination, more than 45% of the analyzed queries use the OPT operator [28].

The importance of the OPT operator has also been recognized from a database theory point of view. It has been shown that the combined complexity of SPARQL query evaluation (i.e., checking if some set of variable bindings is a solution) raises from PTIME-membership for the conjunctive fragment to PSPACE-completeness when OPT is considered [27, 34]. In [27], the class of *well-designed* SPARQL *graph patterns* was introduced as a fundamental fragment of OPT queries with good behavior for query evaluation (for a formal definition, see Section 2). In particular, it was shown that the complexity of the evaluation problem for the well-designed fragment is coNP-complete [27].

In this paper we embark on the static analysis of SPARQL queries containing the OPT operator. We focus on the class of *well-designed* SPARQL *graph patterns* mentioned above. As our first contribution we introduce a tree representation of SPARQL queries called SPARQL *pattern trees*. We also introduce a particular class of pattern trees that we call *quasi well-designed* pattern trees (QWDPTs, for short), capturing the class of well-designed SPARQL graph patterns. We further introduce a procedure to evaluate QWDPTs in a top-down way that resembles a top-down evaluation of graph patterns proposed in [25]. Our pattern trees are reminiscent of relational query plans and thus, can be considered as a first step towards an algebra for logical optimization of queries in the SPARQL context, very much like the relational algebra used in relational database systems to construct and manipulate query plans. Notice that previous works on optimization of SPARQL have mainly focused on rewriting queries based on properties of particular operators [27, 34]. We propose transformation rules for QWDPTs that work at the level of the structure of the trees (and thus, the structure of queries). These rules are, for example, capable of eliminating several sources of redundancy in queries, and thus, can be used for query optimization purposes.

Based on our work on the structure of pattern trees, we study the fundamental problems of checking equivalence and containment of SPARQL queries. It is known that full-SPARQL and First-Order Logic have the same expressive power [2]. From this result it is not difficult to prove that equivalence and containment for SPARQL in general are undecidable problems. We show that the equivalence problem for QWDPTs (and therefore, for well-designed SPARQL graph patterns) is NP-complete. The difficult part of the proof is the NP-membership. Recall from the relational world that equivalence and containment are closely related to the search for homomorphisms. The key to our NP-membership result is an appropriate extension of homomorphisms to QWDPTs – leading to the notion of *strong homomorphisms* (for details, see Section 4) and a normal form via the transformation rules for QWDPTs mentioned above.

For the *containment* of queries we consider the *subsumption* relation [3]. As detailed above, solutions for queries containing the OPT operator are essentially incomplete and may possibly bind only a subset of the variables in the query [31, 27]. This naturally leads to the notion of subsumption between solutions: a solution μ_1 subsumes another solution μ_2, if μ_1 extends μ_2 with more variable bindings. More generally, a SPARQL query T_1 subsumes another SPARQL query T_2 if, for every RDF graph G, every solution of T_2 is subsumed by some solution of T_1. It has been argued that subsumption is a meaningful way of comparing the result of SPARQL queries containing the OPT operator [3]. Moreover, subsumption has also been used in the past as a meaningful way of testing containment of queries with incomplete answers over semistructured data [21]. In principle, subsumption can also be used to test equivalence. However, it is not advisable to do so since we prove that subsumption is presumably harder than equivalence by showing the Π_2^P-completeness of subsumption.

As our final contribution, we study the relationship between tractable fragments of CQ answering and tractable fragments of well-designed SPARQL queries containing the OPT operator. For the classical evaluation problem mentioned above, results on tractable fragments of CQs smoothly carry over to well-designed SPARQL graph patterns. The analysis becomes more intricate when we study the enumeration problem (that is, actually computing the set of solutions) and the counting problem (i.e., determining the number of solutions) of well-designed SPARQL graph patterns. Our main result in this respect states that, for a SPARQL query in which all its conjunctive parts (sequences of AND operators) belong to a tractable class of conjunctive queries, the enumeration of solutions is feasible with *polynomial delay*. In contrast, the counting problem remains intractable (more specifically, #P-complete) also in the restricted case where all conjunctions of triple patterns in a SPARQL query are acyclic CQs.

Summary of results and structure of the paper. In Section 2, we formally introduce RDF and SPARQL. A conclusion and an outlook to future work are given in Section 6. The main results of the paper are detailed in the Sections 3 – 5, namely:

- *Algebra of query plans.* In Section 3, we present an algebra of query plans for well-designed SPARQL graph patterns. To this end, we introduce the data structure of quasi well-designed pattern trees (QWDPTs). We show that they allow for a natural evaluation in a top-down fashion of the tree. Moreover, we introduce transformation rules to modify these QWDPTs. In particular, these rules allow us to eliminate redundancies and to define a useful normal form.

- *Containment and equivalence.* In Section 4, we study the fundamental problems of containment (in the form of subsumption) and equivalence of well-designed SPARQL graph patterns. We establish the relationship between subsumption and equivalence and pinpoint their complexity, namely NP-completeness for equivalence, vs. Π_2^P-completeness for subsumption.

- *Enumeration and counting.* In Section 5, we study both, the enumeration and counting problem of SPARQL queries. In particular, we investigate how tractable fragments of CQ evaluation can be carried over to well-designed SPARQL graph patterns. We show that tractable fragments of CQ evaluation indeed give rise to tractable fragments of the SPARQL enumeration problem. In contrast, the counting problem remains intractable even if all sets of triple patterns in the SPARQL query are restricted to acyclic CQs.

We have implemented a prototype tool based on the theoretical results presented in this paper, which is freely available on-line from `http://db.ing.puc.cl/sparql-algebra`. In particular, we have implemented the containment and equivalence tests, and the top down evaluation of SPARQL tree patterns, with some encouraging initial results. We report on them in Section 6.

Related work. As we have described, our work is heavily based on the formalization of SPARQL presented in [25, 27] and in particular on the notion of well-designed SPARQL patterns introduced in these papers. In [27] the authors also study the complexity of query evaluation. Schmidt et al. [34] considered several aspects of SPARQL query optimization focused on rewriting queries based on properties of operators [34]. Neither of these works [27, 34] considered the complexity of equivalence and containment, nor the search for tractable fragments for query evaluation, enumeration and counting which are the main problems touched in this paper. The OPT operator in SPARQL resembles a *left-outer join* in SQL. Compared with the huge amount of research on static analysis of

CQs, fragments of SQL containing left-outer join have almost been disregarded with respect to these problems with [22] being one notable exception. Nevertheless, to the best of our knowledge, research on fundamental questions such as the complexity of query equivalence and tractable fragments for query evaluation, has not been carried out to date for queries containing left-outer joins.

Outside the SPARQL context, Kanza et al. [21] studied containment and equivalence for queries over a general semistructured data model. The query language considered in [21] allows for partial answers, nevertheless, as opposed to SPARQL, it does not allow to explicitly state optional parts in a query, and partial answers are generated by considering different semantics for query evaluation. This makes our approach to partial answers, equivalence and containment, orthogonal to [21]. Gutierrez et al. [19] studied similar problems for an abstract RDF query language. The difficulties in [19] arise from considering *blank nodes* and RDFS (features that we do not consider here) but they only consider conjunctive queries without optional parts, thus making their approach also orthogonal to ours. Cohen et al. [11] define a polynomial delay iterator for computing full disjunctions. Full disjunctions are designed to obtain partial answers from relational sources but, in contrast to the OPT operator in SPARQL, full disjunctions are associative and commutative. Thus, the source of difficulties in devising an enumerator for full disjunctions departs from the difficulties that one encounters when enumerating SPARQL queries. Regarding counting partial answers, to the best of our knowledge, this paper is the first one considering the complexity of this problem.

2. BASICS OF RDF AND SPARQL

In this paper we focus on ground RDF graphs, that is, RDF graphs that do not contain blank nodes. Moreover, we do not make an explicit distinction between URIs (uniform resource identifiers) and Literals when defining RDF graphs, and thus we assume that RDF graphs are composed only of URIs. Thus, let $\mathbf{U}$ be an infinite set of URIs. An RDF triple is a tuple in $\mathbf{U} \times \mathbf{U} \times \mathbf{U}$, and an RDF graph (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 URIs actually appearing in G.

SPARQL [31] is the standard query language for RDF. We next formalize its *graph pattern matching facility* 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 with $?X$. Then a SPARQL triple pattern is 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 paper we focus on the SPARQL fragment composed of the operators AND and OPT. Formally, SPARQL graph patterns are recursively defined as follows. (1) a triple pattern is a graph pattern, and (2) if P_1 and P_2 are graph patterns, then $(P_1 \text{ AND } P_2)$ and $(P_1 \text{ OPT } P_2)$ are graph patterns.

For a triple pattern t, we write $\mathrm{vars}(t)$ to denote the set of variables occurring in t, and for a graph pattern P we write $\mathrm{vars}(P)$ for the set of variables that occur in the triples that compose P.

To define the semantics of SPARQL graph patterns, we follow closely the definitions proposed in [27]. A mapping μ is a partial function $\mu : \mathbf{V} \to \mathbf{U}$. 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:

$$M_1 \bowtie M_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in M_1, \mu_2 \in M_2 \text{ and } \mu_1 \sim \mu_2\}$$
$$M_1 \mathbin{⟕} M_2 = (M_1 \bowtie M_2) \cup \{\mu \in M_1 \mid \forall \mu' \in M_2 : \mu \not\sim \mu'\}$$

We now have all the necessary prerequisites to formalize the evaluation of a SPARQL graph pattern over an RDF graph G as a function $[\![\cdot]\!]_G$ that given a pattern returns a set of mappings. Formally, $[\![P]\!]_G$ is defined recursively as follows [27]:

1. If P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \mathrm{dom}(\mu) = \mathrm{vars}(t) \text{ and } \mu(t) \in G\}$.
2. If $P = (P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
3. If $P = (P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \mathbin{⟕} [\![P_2]\!]_G$.

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 for conjunctive queries (CQs), the actual names of the variables matter, since two graph patterns containing different sets of variables can never be equivalent. In [27] 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 [31]. Nevertheless, for the fragment considered in this paper (allowing only for AND and OPT) both semantics coincide [26]. We thus have only formalized the set-semantics of the language.

Example 2.1 (From [27]) *Consider an RDF graph G storing information about professors in a university with the following triples, and the 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)

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

If we evaluate P_1 over G, then intuitively we are retrieving the name of the resources in G and, 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 evaluating P_1 over G we obtain the set of mappings $[\![P_1]\!]_G = \{\mu_1, \mu_2, \mu_3, \mu_4\}$ where

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

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 (and george does not have an email address in G).

Well-designed graph patterns. An important class of SPARQL graph patterns identified in [27], that also plays a central role in this paper, is the class of *well-designed* graph patterns. A pattern P is well-designed if for every subpattern $P' = (P_1 \text{ OPT } P_2)$ 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.

Notice that patterns P_1 and P_2 in Example 2.1 are well-designed. In [27] the authors studied several properties of well-designed patterns. Among others, they showed that the complexity of the evaluation problem is lower for well-designed patterns compared with the general language. Moreover, they suggested that well-designed patterns are suitable for optimization procedures, proposing a set of rewriting rules. In this paper we go an important step further in this direction by proposing an algebra of query plans for well-designed SPARQL graph patterns by introducing an appropriate data structure (so-called pattern trees) and equivalence-preserving transformation rules.

3. PATTERN TREES AND QUERY PLANS

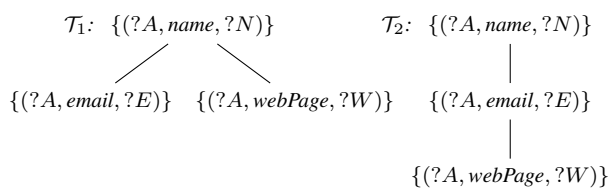
In this section we propose a novel representation of SPARQL graph patterns based on trees, together with an evaluation method for these trees. This tree representation of patterns plays a central role when we study optimization, query equivalence and containment, and also tractable fragments of SPARQL queries.

As usual, we define a *rooted tree* as a tuple $T = (V, E, r)$. We further assume trees to be undirected and unordered. Using this terminology, we can now define our tree representation of SPARQL graph patterns.

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

Given a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n \in V})$ and a node $n \in V$, a *subtree of* $\mathcal{T}$ *rooted at* n is a pattern tree composed of n and a connected subset of its descendants. Moreover, *the complete subtree of* $\mathcal{T}$ *rooted at* n, that we usually denote by $\mathcal{T}_n$, is the pattern tree composed of n and *all* its descendants. We further denote by $\text{vars}(P_n)$ the set of variables that occur in the triples of P_n, and by $\text{vars}(\mathcal{T})$ the set $\bigcup_{n \in V} \text{vars}(P_n)$. For pattern trees, we usually depict the tree structure with the corresponding labels in every node, as in the following example.

Example 3.2 *The following are pattern trees that intuitively correspond to the queries introduced in Example 2.1.*



Next, we give a meaning to pattern trees by transforming pattern trees into SPARQL graph patterns. Towards this goal, we need the following definition of a transformation function $\text{TR}(\cdot, \cdot, \cdot)$. Consider a pattern tree $\mathcal{T} = ((V, E, r), \mathcal{P})$ and a set Σ of functions $\{\sigma_n \mid n \in V\}$ such that for every $n \in V$, function σ_n defines an ordering on the children of n. That is, if n has k children, then σ_n is a function from $\{1, \ldots, k\}$ to the set of children of n, such that $\sigma_n(1)$ is the first child in the order, $\sigma_n(2)$ is the second one, and so on. We need a last definition before presenting the transformation. Given a set $P = \{t_1, \ldots, t_\ell\}$ of triple patterns, we denote by $and(P)$ the graph pattern $(t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_\ell)$. For $\mathcal{T}$ and $n \in V$, we are now ready to define the transformation $\text{TR}(\mathcal{T}, n, \Sigma)$ of $\mathcal{T}_n$, the complete subtree of $\mathcal{T}$ rooted at n, given the order Σ. Assume that n has k children in $\mathcal{T}$, then $\text{TR}(\mathcal{T}, n, \Sigma)$ is defined as the graph pattern expression

$$\big(\cdots \big(\big(and(P_n) \text{ OPT } \text{TR}(\mathcal{T}, \sigma_n(1), \Sigma)\big) \text{ OPT } \text{TR}(\mathcal{T}, \sigma_n(2), \Sigma)\big) \cdots \text{ OPT } \text{TR}(\mathcal{T}, \sigma_n(k), \Sigma)\ \big),$$

and if n has no children, then $\text{TR}(\mathcal{T}, n, \Sigma) = and(P_n)$. Finally, given a pattern tree $\mathcal{T} = ((V, E, r), \mathcal{P})$ and an ordering Σ for $\mathcal{T}$, we define $\text{TR}(\mathcal{T}, \Sigma)$ as $\text{TR}(\mathcal{T}, r, \Sigma)$.

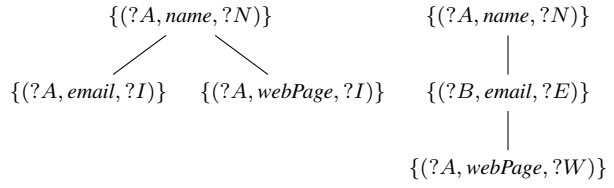
Example 3.3 *Consider the tree $\mathcal{T}_1$ in Example 3.2, and let Σ be the order induced by the picture in the example. Then $\text{TR}(\mathcal{T}_1, \Sigma)$ is pattern P_1 in Example 2.1.*

3.1 Semantics of well-designed pattern trees

We have established a syntactic relationship between pattern trees and SPARQL graph patterns. We now want to establish a semantic relationship between these representations. In particular, 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 introduce a well-designedness condition for pattern trees that will be crucial in defining a semantics for 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 the well-designedness condition for pattern trees.

Definition 3.4 *A pattern tree $\mathcal{T} = ((V, E, r), \mathcal{P})$ is* well-designed *if for every variable $?X$ occurring in $\mathcal{T}$, the set $\{n \in V \mid ?X \in \text{vars}(P_n)\}$ induces a connected subgraph of T.*

Example 3.5 *The pattern trees in Example 3.2 are well-designed, while the following pattern trees are not:*



Variable $?I$ in the tree on the left, and variable $?A$ in the tree on the right, induce disconnected subgraphs.

As expected, this well-designedness condition over trees is tightly connected to the well-designedness condition for graph patterns. In particular, the following holds.

Proposition 3.6 *Let $\mathcal{T}$ be a well-designed pattern tree, and Σ an arbitrary set of ordering functions for $\mathcal{T}$. Then $\text{TR}(\mathcal{T}, \Sigma)$ is a well-designed graph pattern.*

Before defining the semantics, we introduce a relaxation of well-designedness that plays a fundamental role in our study.

Definition 3.7 *A pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n\in V})$ is a* quasi well-designed pattern tree *(*QWDPT *for short) if for every pair of nodes $u, v \in V$ and each variable $?X \in \text{vars}(P_u) \cap \text{vars}(P_v)$ there exists a node n that is a common ancestor of u and v in $\mathcal{T}$, such that $?X \in \text{vars}(P_n)$.*

The pattern tree on the right in Example 3.5 is a QWDPT, while the pattern on the left is not (notice that the common ancestor of u and v in Definition 3.7 may be u or v, as in Example 3.5). Another notion that we need to introduce is that of *duplicating triples to children*. Formally, we say that a pattern tree $\mathcal{T}' = ((V', E', r'), (P'_n)_{n\in V'})$ was derived from a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n\in V})$ 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 $\mathcal{T}_m = \mathcal{T}'$. It is easy to observe that every QWDPT can be converted into a well-designed pattern tree by duplicating triples along branches. Formally, for every QWDPT $\mathcal{T}$, there exists a well-designed pattern tree $\mathcal{T}'$ such that $\mathcal{T} \hookrightarrow^* \mathcal{T}'$. It is also easy to observe that the (quasi) well-designed property is invariant under $\hookrightarrow^*$. We now have all the necessary ingredients to define a semantics of pattern trees. We first define the set of SPARQL graph patterns defined by a QWDPT.

Definition 3.8 *Let $\mathcal{T}$ be a QWDPT. The set of SPARQL graph patterns defined by $\mathcal{T}$ is*

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

To define the result of evaluating a QWDPT $\mathcal{T}$ over an RDF graph G, we first show that all queries in $\text{SEM}(\mathcal{T})$ 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 $\text{SEM}(\mathcal{T})$.

Lemma 3.9 *Let $\mathcal{T}$ be a well-designed pattern tree, let Σ_1, Σ_2 be two arbitrary orderings for $\mathcal{T}$, and let $P_1 = \text{TR}(\mathcal{T}, \Sigma_1)$ and $P_2 = \text{TR}(\mathcal{T}, \Sigma_2)$ be the graph patterns obtained by transforming $\mathcal{T}$ with Σ_1 and Σ_2, respectively. Then $P_1 \equiv P_2$.*

Lemma 3.10 *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 $P_1 = \text{TR}(\mathcal{T}_1, \Sigma)$ and $P_2 = \text{TR}(\mathcal{T}_2, \Sigma)$, then $P_1 \equiv P_2$.*

Putting these two lemmas together, we get the following result.

Theorem 3.11 *Let $\mathcal{T}$ be a QWDPT. Then all graph patterns in $\text{SEM}(\mathcal{T})$ are equivalent, i.e., for any two graph patterns $P_1, P_2 \in \text{SEM}(\mathcal{T})$, it holds that $P_1 \equiv P_2$.*

Definition 3.12 *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 $P \in \text{SEM}(\mathcal{T})$.*

By Theorem 3.11, the semantics of a QWDPT according to Definition 3.12 is well defined. This means that, for QWDPT $\mathcal{T}$, we may choose any representative from $\text{SEM}(\mathcal{T})$ for evaluation. 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 3.12 plus Proposition 3.6 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.

Proposition 3.13 *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.*

3.2 Evaluating pattern trees

In this section we introduce a procedural semantics for QWDPTs that takes advantage of our tree representation. In [25] the authors proposed a top-down evaluation method for SPARQL graph patterns and they showed that given a graph G, it is equivalent to the evaluation given by $[\![\cdot]\!]_G$. Our proposal is similar to the approach in [25], but it is based on an alternative characterization of the evaluation of well-designed graph patterns proposed in [27]. We reformulate here this characterization for the case of pattern trees. It will later play an important role when we study transformations of pattern trees as well as containment and equivalence testing. We first introduce the necessary terminology.

We say that a mapping μ_1 is *subsumed by* μ_2, denoted by $\mu_1 \sqsubseteq \mu_2$, if $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \text{dom}(\mu_1)$ and for every $?X \in \text{dom}(\mu_1)$ it holds that $\mu_1(?X) = \mu_2(?X)$ (implying that $\mu_1 \sim \mu_2$). We write $\mu_1 \sqsubset \mu_2$ whenever $\mu_1 \sqsubseteq \mu_2$ and $\mu_1 \neq \mu_2$. Further, recall that given a set $P = \{t_1, \ldots, t_\ell\}$ of triple patterns, we denote by $and(P)$ the graph pattern $(t_1 \text{ AND } t_2 \text{ AND } \cdots \text{ AND } t_\ell)$. Now given a pattern tree $\mathcal{T} = ((V, E, r), (P_n)_{n\in V})$, we use $and(\mathcal{T})$ to denote the SPARQL pattern constructed by taking the conjunction (AND) of all the triples that occur in $\mathcal{T}$. That is, if $V = \{n_1, \ldots, n_\ell\}$, then

$$and(\mathcal{T}) = \big(and(P_{n_1}) \text{ AND } \cdots \text{ AND } and(P_{n_\ell})\big).$$

We next characterize the evaluation of a QWDPT. It follows directly from the results in [27] for well-designed graph patterns, and the relationship with QWDPTs shown in the previous section.

Lemma 3.14 *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*

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

Lemma 3.14 essentially states that the mappings in the evaluation of a QWDPT over some graph G are exactly those that map all triples in some subtree $\mathcal{T}'$ of $\mathcal{T}$ (hence $and(\mathcal{T}')$) into G, and that cannot be further extended by considering another subtree $\mathcal{T}''$ of $\mathcal{T}$. This characterization inspires the following procedural semantics that is obtained by evaluating the pattern tree by a top-down traversal. For simplicity, given a label P_n of node n and a graph G, we denote by $[\![P_n]\!]_G$ the set $[\![and(P_n)]\!]_G$.

Definition 3.15 *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 $\text{ext}(M, n, G)$ as follows. If n is a leaf, then*

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

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 $M_i = (M \bowtie [\![P_n]\!]_G) \rtimes \mathrm{ext}(M \bowtie [\![P_n]\!]_G, n_i, G)$. 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.

The following theorem shows that the top-down evaluation defined above coincides with the semantics of pattern trees introduced in the previous section.

Theorem 3.16 *Let $\mathcal{T}$ be a QWDPT and G an RDF graph. Then $[\![\mathcal{T}]\!]_G = [\![\mathcal{T}]\!]^{td}_G$.*

Recall that in Definition 3.12 we defined the semantics of QWDPTs by their extensions to well-designed SPARQL patterns. Theorem 3.16 now allows us to define the semantics of QWDPTs directly via their tree representation. This nicely supports the idea of using QWDPTs as query execution plans for well-designed SPARQL query patterns: they provide a syntactical representation of a query together with an operational semantics working on this representation. In these terms, the relaxation from well-designed pattern trees to QWDPTs provides additional potential for optimization and redundancy elimination for those query plans.

3.3 Transformation of QWDPTs

One advantage of QWDPTs is that they allow us to define several equivalence-preserving transformations on the structure of the pattern trees. Previous works [27, 34] on transformation rules for SPARQL patterns have been based on the properties of the SPARQL operators. In contrast, the transformations that we introduce in this section are based on the tree structure of QWDPTs (i.e. the *operator structure*) and the structure of the sets of triple patterns composing the pattern tree. These structural transformations provide further evidence that pattern trees are a suitable query-plan representation for SPARQL in the spirit of classical relational-algebra query plans.

Before presenting our rules, we need to introduce some additional notation. Let $\mathcal{T} = ((V, E, r), \mathcal{P})$ 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 3.17 *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* $\mathrm{vars}(P_r)$.

We are now ready to state a set of transformation rules for QWDPTs. In the formulation of the rules we assume that whenever we remove a node n from a pattern tree, then all edges incident to n are removed as well. We further assume a fixed QWDPT $\mathcal{T} = ((V, E, r), (P_n)_{n \in V})$ to be the pattern tree before the application, and we consider $\mathcal{T}' = ((V', E', r'), (P'_n)_{n \in V'})$ as the resulting QWDPT after applying the rule. If P'_n is not defined explicitly for some $n \in V'$, we always consider $P'_n = P_n$ by "default".

Rule R1 (*deletion of redundant triples*): Let $n \in V$. If there exists a triple $t \in P_n$ s.t. $t \in P_{n'}$ 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.

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 $P'_{n_i} = P_{n_i} \cup P_n$ 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.

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 $h\colon P_n \to P_{\mathrm{branch}(\hat{n})}$ with $h(?X) = ?X$ for all variables $?X \in \mathrm{vars}(P_n) \cap \mathrm{vars}(P_{\mathrm{branch}(\hat{n})})$, then *merge* n into $\hat{n}$, i.e. let $P'_{\hat{n}} = P_{\hat{n}} \cup P_n$, $V' = V \setminus \{n\}$ (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 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 $h\colon P_n \to P_{n'} \cup P_{\mathrm{branch}(\hat{n})}$ with $h(?X) = ?X$ for all variables $?X \in \mathrm{vars}(P_n) \cap \mathrm{vars}(P_{\mathrm{branch}(\hat{n})})$, then 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.

The following result shows the correctness of the rules.

Theorem 3.18 *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, to $\mathcal{T}$. Then $\mathcal{T}'$ is a QWDPT such that $\mathcal{T} \equiv \mathcal{T}'$.*

We say that a QWDPT $\mathcal{T}$ is *reduced* w.r.t. to some rule R, if R cannot be applied to $\mathcal{T}$. While checking if some QWDPT $\mathcal{T}$ is reduced w.r.t. R3 or R4 is an expensive task (it requires to decide the existence of some homomorphisms), it is rather easy to determine if $\mathcal{T}$ is reduced w.r.t. R1 or R2. Moreover, already if $\mathcal{T}$ is reduced only w.r.t. R1 and R2, it possesses some useful properties that make it easier to work with – and reason about – $\mathcal{T}$. We thus introduce a first normal form for QWDPTs based on these two rules.

Definition 3.19 *We say that a QWDPT $\mathcal{T}$ is in* non-redundant normal form (NR normal form) *if $\mathcal{T}$ is reduced w.r.t. rules R1 and R2.*

Next, we discuss some properties of the NR normal form.

Proposition 3.20 *Let $\mathcal{T}$ be a QWDPT. Then the following hold:*

1. *Iteratively applying rules R1 and R2 (in arbitrary order) to $\mathcal{T}$ leads to a unique pattern tree $\mathcal{T}^*$ in NR normal form.*
2. *If $\mathcal{T}$ is in NR normal form then it remains in NR normal form when applying rules R3 or R4 to $\mathcal{T}$.*

The crucial property of the NR normal form is the following. Let $\mathcal{T} = ((V, E, r), \mathcal{P})$ be a QWDPT in NR normal form. Then for every $n \in V$ s.t. $n \neq r$, it holds that $\mathrm{newvars}(n) \neq \emptyset$. This simple property, which follows directly from the definition of rule R2, allows us to define an alternative characterization of the solutions of QWDPTs in terms of maximal subtrees. In the characterization we use the following notation. Given a mapping μ and a set of mappings M, we say that M subsumes μ, denoted by $\mu \sqsubseteq M$ if there exists a mapping $\nu \in M$ such that $\mu \sqsubseteq \nu$.

Lemma 3.21 *Let $\mathcal{T}$ be a QWDPT in NR normal form with root r, and G an RDF graph. Then $\mu \in [\![\mathcal{T}]\!]_G$ if and only if there exists a subtree $\mathcal{T}'$ of $\mathcal{T}$ rooted at r such that*

1. $\mathrm{dom}(\mu) = \mathrm{vars}(\mathcal{T}')$, *and*
2. $\mathcal{T}'$ *is the maximal subtree of $\mathcal{T}$ such that $\mu \sqsubseteq [\![and(\mathcal{T}')]\!]_G$.*

Notice that as opposed to Lemma 3.14 that characterizes the mappings in the evaluation of a QWDPT as the maximal (w.r.t. $\sqsubseteq$) mappings satisfying some property, Lemma 3.21 takes advantage of the NR normal form to characterize mappings in terms of the structure of a QWDPT, in particular, in terms of maximal subtrees.

The NR normal form provides a "cheap" elimination of some redundancies. As such it will be an integral part of the equivalence test for QWDPTs in Section 4.2. But also rule R3, which can reduce some more complex sources of redundancy in the structure of trees, will play an important role in the equivalence test. We thus introduce another normal form.

Definition 3.22 *Let $\mathcal{T}$ be a QWDPT. We say that $\mathcal{T}$ is in* R3 normal form *if $\mathcal{T}$ is reduced w.r.t. rules R1, R2, and R3.*

One intuition of the R3 normal form is that given some QWDPT $((V, E, r), \mathcal{P})$ in this normal form, for every $n \in V$ with parent $\hat{n}$, there exists at least one RDF graph G and mapping μ with $\mu(P_{\mathrm{branch}(\hat{n})}) \subseteq G$ that cannot be extended to a mapping μ' s.t. $\mu'(P_n) \subseteq G$. I.e. from the fact that some variable assignment maps $P_{\mathrm{branch}(\hat{n})}$ into G, we cannot derive any statement about P_n. This intuitively implies that every node in the tree carries some information which is non-redundant with respect to its ancestors. We next discuss some basic properties of the R3 normal form.

Proposition 3.23 *Let $\mathcal{T}$ be a QWDPT. Then the following hold:*

1. *Iteratively applying R1, R2, and R3 to $\mathcal{T}$ eventually leads to a (not necessarily unique) pattern $\mathcal{T}^*$ that is in R3 normal form. Moreover, if $\mathcal{T}$ is in NR-normal form, then iteratively applying R3 leads to a unique pattern $\mathcal{T}^*$ in R3 normal form.*
2. *The number of rule applications of R1, R2, and R3 needed to arrive at a pattern in R3 normal form is linear in the size of $\mathcal{T}$.*

We have proposed a tree representation of SPARQL queries and a set of rules that can be used to restructure these trees. We can consider these trees as query plans for SPARQL queries. The results presented in this section therefore describe a starting point for the study of an algebra of query plans, which forms the basis of query optimization for this language. QWDPTs together with rules R1 – R3 will also be crucial for studying classical static analysis problems for SPARQL in the next section. Rule R4 has been mainly presented so as to give a flavor of what further transformation rules in this algebra could look like. It may be beneficial in particular in an environment where parallel processing is supported.

4. CONTAINMENT AND EQUIVALENCE

In this section we study the fundamental problems of containment and equivalence of well-designed SPARQL queries. Similarly to query languages on relational databases, these problems are crucial for query optimization. For containment we consider the *subsumption relation* ($\sqsubseteq$) introduced in Section 3.2 rather than the classical subset relation ($\subseteq$). Clearly, for CQs, the two notions coincide. However, in the presence of partial query answers, subsumption is the more natural notion of containment [21, 3], and has also been considered in recent work to compare the evaluation of two patterns containing OPT operators [27, 3]. This is illustrated in the following example (taken from [3]).

Example 4.1 (From [3]) *Consider two SPARQL graph patterns $P_1 = (?X, n, ?Y)$ and $P_2 = (?X, n, ?Y)$ OPT $(?X, e, ?Z)$, and an RDF graph $G = \{(a, n, b), (a, e, c)\}$. Then $[\![P_1]\!]_G = \{\mu = \{?X \to a, ?Y \to b\}\}$, while $[\![P_2]\!]_G = \{\mu' = \{?X \to a, ?Y \to b, ?Z \to c\}\}$. Hence $P_1 \not\sqsubseteq P_2$. This is, however, unintuitive, since the answer to P_2 contains strictly more information than that to P_1, and it is easy to see that for no graph G, pattern P_2 returns fewer bindings than P_1.*

For CQs without existentially quantified variables, both equivalence and containment are tractable. In the presence of existential quantifiers, they are classical NP-complete problems [8]. In this paper, we study equivalence and containment for well-designed SPARQL queries, or, equivalently, for our representation by quasi well-designed pattern trees (QWDPTs). In contrast to CQs, the complexities of containment (in the form of subsumption) and equivalence diverge in this case. Indeed, we prove that subsumption between QWDPTs is Π_2^P-complete while the equivalence problem is NP-complete. The NP-membership will be the most difficult part to prove. The key to this NP-membership result is the R3-normal form introduced in the previous section and an appropriate extension of homomorphisms, which we shall refer to as "strong homomorphisms".

4.1 Complexity of subsumption

We extend the definition of subsumption of mappings introduced in Section 3.2, to subsumption of sets of mappings. Given sets of mappings M_1 and M_2 we say that M_1 is subsumed by M_2, denoted by $M_1 \sqsubseteq M_2$, if for every $\mu_1 \in M_1$ there exists a $\mu_2 \in M_2$ such that $\mu_1 \sqsubseteq \mu_2$. Further, for two QWDPTs $\mathcal{T}_1$ and $\mathcal{T}_2$, we say that $\mathcal{T}_1$ is subsumed by $\mathcal{T}_2$, denoted by $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$, if $[\![\mathcal{T}_1]\!]_G \sqsubseteq [\![\mathcal{T}_2]\!]_G$ holds for every graph G. We are now ready to provide a necessary and sufficient condition to test whether $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$.

Lemma 4.2 *Consider QWDPTs $\mathcal{T}_1$ and $\mathcal{T}_2$ with roots r_1 and r_2, respectively. Then $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$ if and only if for every subtree $\mathcal{T}_1'$ of $\mathcal{T}_1$ rooted at r_1, there exists a subtree $\mathcal{T}_2'$ of $\mathcal{T}_2$ rooted at r_2 s.t.:*

1. $\mathrm{vars}(\mathcal{T}_1') \subseteq \mathrm{vars}(\mathcal{T}_2')$, *and*
2. *there exists a homomorphism from the triples in $\mathcal{T}_2'$ to the triples in $\mathcal{T}_1'$ that is the identity over* $\mathrm{vars}(\mathcal{T}_1')$.

Lemma 4.2 yields a straightforward Π_2^P procedure to check whether $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$ holds: check for every subtree $\mathcal{T}_1'$ of $\mathcal{T}_1$ that there exists a subtree $\mathcal{T}_2'$ of $\mathcal{T}_2$ and a homomorphism satisfying properties (1) and (2). Below, we also show the matching lower bound.

Theorem 4.3 *The subsumption problem of QWDPTs (and, therefore, of well-designed SPARQL graph patterns) is Π_2^P-complete.*

PROOF IDEA. The membership was argued above. The hardness is shown by reduction from the well-known Π_2^P-hard problem 3-QSAT$_{\forall,2}$. Given an arbitrary instance of this problem by a quantified Boolean formula $\Psi = \forall \vec{x} \exists \vec{y}\, \phi(\vec{x}, \vec{y})$ where ϕ is in 3-CNF, we define two QWDPTs $\mathcal{T}_1$ and $\mathcal{T}_2$, s.t. $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$ iff Ψ is valid. The QWDPT $\mathcal{T}_2$ consists of the root only and contains a variable $?X_i$ and $?Y_j$ for every variable x_i in $\vec{x}$ resp. y_j in $\vec{y}$. Moreover, $\mathcal{T}_2$ contains triple patterns which "encode" the clauses in ϕ. The QWDPT $\mathcal{T}_1$ consists of the root plus one child node n_i for every variable x_i in $\vec{x}$. Recall from Lemma 3.14 the correspondence between mappings μ_1 in $[\![\mathcal{T}_1]\!]_G$ (for a some RDF graph G) and subtrees of $\mathcal{T}_1$. Now every $\mu_1 \in [\![\mathcal{T}_1]\!]_G$ corresponds to some subtree $\mathcal{T}_1'$ of $\mathcal{T}_1$ containing a particular subset of the child nodes n_i of the root of $\mathcal{T}_1$. Moreover, every such μ_1 requires a particular binding of the variables $?X_i$ in $\mathcal{T}_2$ and thus defines a particular truth assignment I on $\vec{x}$ (with $I(x_i) = \mathit{true}$ iff $\mu_1(P_{n_i}) \subseteq G$). Then μ_1 can be extended

to a mapping μ_2 in $[\![\mathcal{T}_1]\!]_G$ if all triple patterns in $\mathcal{T}_2$ can be sent to G by an appropriate instantiation of the variables $?Y_j$. Again the variable bindings of the variables $?Y_j$ are in 1-to-1 correspondence with truth value assignments to the variables y_j in $\vec{y}$. Moreover, sending the triple patterns in $\mathcal{T}_2$ (which encode the clauses of ϕ) into G comes down to satisfying the clauses in ϕ. □

Our next result establishes the close connection between subsumption and equivalence of well-designed SPARQL queries.

Lemma 4.4 *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two QWDPTs. Then $\mathcal{T}_1 \equiv \mathcal{T}_2$ if and only if $\mathcal{T}_1 \sqsubseteq \mathcal{T}_2$ and $\mathcal{T}_2 \sqsubseteq \mathcal{T}_1$.*

From Theorem 4.3 and Lemma 4.4 we obtain that equivalence of well-designed SPARQL queries can be tested in Π_2^P. However, in the next section we provide a better upper-bound, namely NP.

4.2 Complexity of equivalence

We now prove that testing the equivalence of two QWDPTs (and, thus of two well-designed SPARQL queries) is NP-complete. The difficult part is the NP-membership. A key concept for this proof is the notion of a *strong homomorphism* between two branches of one or two pattern trees. Based on this concept, we introduce the notion of *strongly homomorphically equivalent* branches.

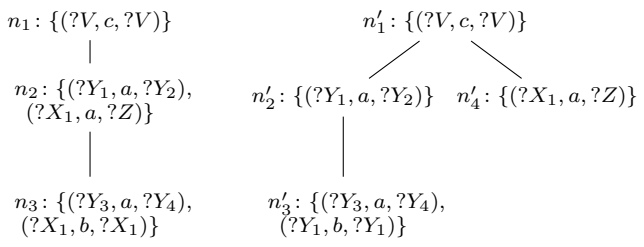
Definition 4.5 (strong homomorphism) *Consider two QWDPTs $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$. Moreover, let $n_1 \in V_1$, $n_2 \in V_2$, and let $\mathrm{branch}(n_1, \mathcal{T}_1)$ be the sequence of nodes $r_1 = n^1, \ldots, n^k = n_1$.*

We say that there exists a strong homomorphism *$\mathcal{H}\colon \mathrm{branch}(n_1, \mathcal{T}_1) \to \mathrm{branch}(n_2, \mathcal{T}_2)$ if $\mathcal{H}$ is a set $\mathcal{H} = \{h_i \mid 1 \leq i \leq k\}$ of homomorphisms $h_i\colon P_{n^i} \to P_{\mathrm{branch}(n_2,\mathcal{T}_2)} \cup P_{\mathrm{branch}(n^{i-1},\mathcal{T}_1)}$ s.t. $h_i(?X) = ?X$ for all $?X \in \mathrm{vars}(P_{n^i}) \cap \mathrm{vars}(P_{\mathrm{branch}(n_2,\mathcal{T}_2)} \cup P_{\mathrm{branch}(n^{i-1},\mathcal{T}_1)})$ (where for $i = 1$ let $P_{\mathrm{branch}(n^{i-1},\mathcal{T}_1)} = \emptyset$).*

We further say that $\mathrm{branch}(n_1, \mathcal{T}_1)$ and $\mathrm{branch}(n_2, \mathcal{T}_2)$ are strongly homomorphically equivalent *if there exist strong homomorphisms $\mathcal{H}_1\colon \mathrm{branch}(n_1, \mathcal{T}_1) \to \mathrm{branch}(n_2, \mathcal{T}_2)$ and $\mathcal{H}_2\colon \mathrm{branch}(n_2, \mathcal{T}_2) \to \mathrm{branch}(n_1, \mathcal{T}_1)$.*

The basic intuition of a strong homomorphism $\mathcal{H}\colon \mathrm{branch}(n_1) \to \mathrm{branch}(n_2)$ is that every variable assignment μ that maps $P_{\mathrm{branch}(n_2)}$ into some RDF graph G can be extended to a variable assignment μ' that also maps $P_{\mathrm{branch}(n_1)}$ into G. Note that a simple homomorphism $h\colon P_{\mathrm{branch}(n_1)} \to P_{\mathrm{branch}(n_2)}$ is not enough to guarantee this, as can be seen in the following example.

Example 4.6 *Consider the following QWDPTs $\mathcal{T}_1$ and $\mathcal{T}_2$:*



Consider the branches $\mathrm{branch}(n_3, \mathcal{T}_1)$ and $\mathrm{branch}(n_3', \mathcal{T}_2)$. It is easy to see that there exists a homomorphism $h\colon P_{\mathrm{branch}(n_3,\mathcal{T}_1)} \to P_{\mathrm{branch}(n_3',\mathcal{T}_2)}$, that is the identity on all shared variables $?Y_1$, $?Y_2$, $?Y_3$, $?Y_4$, $?V$, and with $h(?X_1) = ?Y_1$ and $h(?Z) = ?Y_2$. However, there does not exist a strong homomorphism $\mathcal{H}\colon P_{\mathrm{branch}(n_3,\mathcal{T}_1)} \to P_{\mathrm{branch}(n_3',\mathcal{T}_2)}$ because a homomorphism $h_3\colon P_{n_3} \to P_{\mathrm{branch}(n_3',\mathcal{T}_2)} \cup P_{\mathrm{branch}(n_2,\mathcal{T}_1)}$ with the desired properties according to Definition 4.5 is missing.

Now consider the RDF graph $G = \{(v, c, v), (y_1, a, y_2), (x_1, a, z), (y_3, a, y_4), (y_1, b, y_1)\}$, and some variable assignment τ that is defined only on variables in $\mathrm{branch}(n_3', \mathcal{T}_2)$. If τ maps $\mathrm{branch}(n_3', \mathcal{T}_2)$ into G, then because of the homomorphism h there exists and extension τ' of τ that also maps $\mathrm{branch}(n_3, \mathcal{T}_1)$ into G.

However, if we consider variable mappings that assign a value to variables not occurring in $\mathrm{branch}(n_3', \mathcal{T}_2)$ (which will be necessary in order to test equivalence), then simple homomorphisms are not enough. This is especially true if the additional variables occur somewhere in the branch of n_3. Consider $\mu\colon \{?V \to v, ?Z \to z\} \cup \{?Y_i \to y_i \mid 1 \leq i \leq 4\} \cup \{?X_1 \to x_1\}$, for which $\mu(P_{\mathrm{branch}(n_3',\mathcal{T}_2)}) \subseteq G$ holds. But $\mu(P_{\mathrm{branch}(n_3,\mathcal{T}_1)}) \nsubseteq G$: because of $\mu(?X_1) = x_1$, the triple $(?X_1, b, ?X_1)$ is mapped to (x_1, b, x_1) by μ, but $(x_1, b, x_1) \notin G$. Hence, despite the existence of h and $\mu(\mathrm{branch}(n_3', \mathcal{T}_2)) \subseteq G$, μ is not a solution to $\mathcal{T}_1$.

Note that if we replace the pattern $(?X_1, b, ?X_1)$ in n_3 by $(?Y_1, b, ?Y_1)$, then there exists a strong homomorphism $\mathcal{H}\colon P_{\mathrm{branch}(n_3,\mathcal{T}_1)} \to P_{\mathrm{branch}(n_3',\mathcal{T}_2)}$. Actually, $\mathcal{T}_1$ and $\mathcal{T}_2$ can then be shown to be equivalent.

Consider nodes n_1 and n_2 as in Definition 4.5. Intuitively, the reason why a simple homomorphism is not enough is that there may be RDF graphs G and variable assignments μ that not only map all triple patterns in $\mathrm{branch}(n_2)$ into G, but also the triple patterns contained in some "prefix" of $\mathrm{branch}(n_1)$. Hence to extend such variable assignments to the complete branch $\mathrm{branch}(n_1)$, the existing variable assignments on the variables in this "prefix" of $\mathrm{branch}(n_1)$ must not be altered. This idea, which will be crucial for our NP equivalence test, is formalized in the following lemma.

Lemma 4.7 *Consider two QWDPTs $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$, and let $n_1 \in V_1$ and $n_2 \in V_2$ with $\mathrm{branch}(n_1, \mathcal{T}_1) = n^1, \ldots, n^k$. Then the following statements are equivalent:*

1. *There exists a strong homomorphism $\mathcal{H}\colon \mathrm{branch}(n_1, \mathcal{T}_1) \to \mathrm{branch}(n_2, \mathcal{T}_2)$.*
2. *For every $i \in \{1, \ldots, k\}$, for every RDF graph G and every mapping $\mu\colon \mathrm{vars}(P_{\mathrm{branch}(n_2,\mathcal{T}_2)} \cup P_{\mathrm{branch}(n^{i-1},\mathcal{T}_1)}) \to \mathrm{dom}(G)$ the following holds:*
   If $\mu(P_{\mathrm{branch}(n_2,\mathcal{T}_2)} \cup P_{\mathrm{branch}(n^{i-1},\mathcal{T}_1)}) \subseteq G$, then there exists a mapping $\mu'\colon \mathrm{vars}(P_{\mathrm{branch}(n_1,\mathcal{T}_1)}) \to \mathrm{dom}(G)$ s.t. $\mu'(P_{\mathrm{branch}(n_1,\mathcal{T}_1)}) \subseteq G$ and $\mu(?X) = \mu'(?X)$ for all $?X \in \mathrm{dom}(\mu) \cap \mathrm{dom}(\mu')$ (where for $i = 1$, let $P_{\mathrm{branch}(n^{i-1})} = \emptyset$).

If n_1 and n_2 are from the same pattern tree, we can show a slightly stronger result below, namely: for every solution that maps the triple patterns at the branch to n_2 into an RDF graph G, the same solution also maps the triple patterns at the branch to n_1 into G, provided that a strong homomorphism $\mathrm{branch}(n_1) \to \mathrm{branch}(n_2)$ exists.

Proposition 4.8 *Consider a QWDPT $\mathcal{T} = ((V, E, r), \mathcal{P})$ and nodes $n_1, n_2 \in V$. Then the following statements are equivalent:*

1. *There exists a strong homomorphism $\mathcal{H}\colon \mathrm{branch}(n_1, \mathcal{T}) \to \mathrm{branch}(n_2, \mathcal{T})$.*
2. *For every RDF graph G and $\mu \in [\![\mathcal{T}]\!]_G$ it holds that $\mu(P_{\mathrm{branch}(n_1)}) \subseteq G$ whenever $\mu(P_{\mathrm{branch}(n_2)}) \subseteq G$.*

For convenience, we introduce some more notation. Let $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$ be two QWDPTs. For $n_1 \in V_1$, we denote with $\mathrm{cor}(n_1, V_2)$ the set of *corresponding nodes in V_2*, i.e., $\mathrm{cor}(n_1, V_2) = \{n_2 \mid n_2 \in V_2, \mathrm{newvars}(n_1) \cap \mathrm{newvars}(n_2) \neq \emptyset\}$. If clear from the context, we may drop the V_2 and just write $\mathrm{cor}(n_1)$ to increase readability.

Before showing the NP-membership of testing the equivalence between two QWDPTs (and, hence, of two well-designed SPARQL queries), we still need some further results. The following lemma gives 3 necessary conditions for the equivalence of two QWDPTs in R3-normal form: the two QWDPTs must have identical roots, the same set of variables and the same set of triples. Note that the R3-normal form is crucial for the first and the last condition.

Lemma 4.9 *Consider two QWDPTs $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$ in R3-normal form. If $\mathcal{T}_1 \equiv \mathcal{T}_2$ then the following properties hold:*

1. $P_{r_1} = P_{r_2}$
2. $\mathrm{vars}(\mathcal{T}_1) = \mathrm{vars}(\mathcal{T}_2)$
3. $\bigcup_{n \in V_1} P_n = \bigcup_{n \in V_2} P_n$

Lemma 4.10 *Consider two QWDPTs $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$ in R3-normal form. Let $n_1, n_3 \in V_1$ and $n_2 \in V_2$.*

If there exist strong homomorphisms $\mathcal{H}\colon \mathrm{branch}(n_3, \mathcal{T}_1) \to \mathrm{branch}(n_2, \mathcal{T}_2)$ and $\mathcal{H}'\colon \mathrm{branch}(n_2, \mathcal{T}_2) \to \mathrm{branch}(n_1, \mathcal{T}_1)$, then n_1 is not an ancestor of n_3.

We are now ready to formulate necessary *and* sufficient conditions for the equivalence of two QWDPTs in R3-normal form.

Theorem 4.11 *Let $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$ be two QWDPTs in R3-normal form. Then $\mathcal{T}_1 \equiv \mathcal{T}_2$ iff (1) $\bigcup_{n \in V_1} P_n = \bigcup_{n \in V_2} P_n$, (2) $P_{r_1} = P_{r_2}$, and (3) for all pairs (n_1, n_2) of nodes $n_1 \in V_1$ and $n_2 \in V_2$ with $\mathrm{newvars}(n_1) \cap \mathrm{newvars}(n_2) \neq \emptyset$ it holds that $\mathrm{branch}(n_1, \mathcal{T}_1)$ and $\mathrm{branch}(n_2, \mathcal{T}_2)$ are strongly homomorphically equivalent.*

We want to point out that the requirement that both queries contain the same set of atoms is necessary, and does not follow from the strong homomorphical equivalence of all nodes that share "new" variables, as can be seen in the following example.

Example 4.12 *Consider the following two QWDPTs.*

Obviously these two QWDPTs are not equivalent, as they do not even contain the same set of variables. However it can be easily checked that all required strong homomorphisms exist.

Theorem 4.11 immediately yields an equivalence test for well-designed SPARQL queries $\mathcal{T}_1$ and $\mathcal{T}_2$: just consider the SPARQL queries as QWDPTs, transform them into R3-normal form, and check if the conditions from Theorem 4.11 are fulfilled. However, there is a serious problem with such an algorithm. Indeed, we cannot afford to transform $\mathcal{T}_1$ and $\mathcal{T}_2$ into R3-normal form in an NP-algorithm, since this includes a coNP-test for checking that no further application of rule R3 is possible. The following theorem provides a way to overcome this problem. Intuitively it states that we do not need to transform $\mathcal{T}_1$ and $\mathcal{T}_2$ into R3-normal form. Instead, it suffices to apply rule R3 "often enough" so that the conditions from Theorem 4.11 are fulfilled. Theorem 4.13 guarantees that these conditions still hold for the R3 normal form, which we never explicitly need to compute.

Theorem 4.13 *Consider two QWDPTs $\mathcal{T}_1 = ((V_1, E_1, r_1), \mathcal{P}_1)$ and $\mathcal{T}_2 = ((V_2, E_2, r_2), \mathcal{P}_2)$ in NR normal form such that (1) $\bigcup_{n \in V_1} P_n = \bigcup_{n \in V_2} P_n$, (2) $P_{r_1} = P_{r_2}$, and (3) for all pairs (n_1, n_2) of nodes $n_1 \in V_1$ and $n_2 \in V_2$ with $\mathrm{newvars}(n_1) \cap \mathrm{newvars}(n_2) \neq \emptyset$ it holds that $\mathrm{branch}(n_1, \mathcal{T}_1)$ and $\mathrm{branch}(n_2, \mathcal{T}_2)$ are strongly homomorphically equivalent. Further, let $\mathcal{T}_1^* = ((V_1^*, E_1^*, r_1^*), \mathcal{P}_1^*)$ and $\mathcal{T}_2^* = ((V_2^*, E_2^*, r_2^*), \mathcal{P}_2^*)$ be R3 normal forms of $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively.*

Then the following conditions still hold for $\mathcal{T}_1^*$ and $\mathcal{T}_2^*$: (1) $\bigcup_{n \in V_1^*} P_n = \bigcup_{n \in V_2^*} P_n$, (2) $P_{r_1^*} = P_{r_2^*}$, and (3) for all pairs (n_1^*, n_2^*) of nodes $n_1^* \in V_1^*$ and $n_2^* \in V_2^*$ with $\mathrm{newvars}(n_1^*) \cap \mathrm{newvars}(n_2^*) \neq \emptyset$, it holds that $\mathrm{branch}(n_1^*, \mathcal{T}_1^*)$ and $\mathrm{branch}(n_2^*, \mathcal{T}_2^*)$ are strongly homomorphically equivalent.

Now we have all ingredients to prove the main result of this section.

Theorem 4.14 *The equivalence problem of QWDPTs (and, therefore, of well-designed SPARQL graph patterns) is NP-complete.*

PROOF. The NP-hardness is shown by a straightforward reduction from 3-Colorability. The NP-membership can be seen by the following algorithm, which takes two QWDPTs $\mathcal{T}_1$ and $\mathcal{T}_2$ as input.

1. Transform $\mathcal{T}_1$ and $\mathcal{T}_2$ into NR normal forms $\mathcal{T}_1^*$ and $\mathcal{T}_2^*$, resp.
2. Guess two application sequences of rule R3 (i.e., nodes where to apply rule R3 and the corresponding homomorphisms).
3. Check that the homomorphisms from step 2 fulfill the conditions of rules R3. Denote the QWDPTs resulting from these R3 applications with $\mathcal{T}_1'$ and $\mathcal{T}_2'$.
4. For any two nodes n_1 in $\mathcal{T}_1'$ and n_2 in $\mathcal{T}_2'$ with $\mathrm{newvars}(n_1) \cap \mathrm{newvars}(n_2) \neq \emptyset$, guess strong homomorphisms (i.e., collections of homomorphisms according to Definition 4.5) in both directions between $\mathrm{branch}(n_1, \mathcal{T}_1')$ and $\mathrm{branch}(n_2, \mathcal{T}_2')$.
5. Check that $\mathcal{T}_1'$ and $\mathcal{T}_2'$ fulfill conditions (1) – (3) of Theorem 4.13.

Step 1 is feasible in polynomial time since rules R1 and R2 are "cheap". The certificate guessed in steps 2 and 4 is polynomially bounded. Finally, the computation and checks in step 3 and the checks in step 5 fit into polynomial time. □

5. ENUMERATION AND COUNTING

Conjunctive query (CQ) evaluation[1] is a classical NP-complete problem [8]. A lot of effort has thus been invested into the search for tractable fragments of CQs [41, 10, 13, 17, 18, 16]. This search for tractable fragments of CQs has also been extended to the enumeration problem (i.e., given a CQ Q and a database D, output all tuples in the result of Q over D) and the counting problem of CQs (i.e., given a CQ Q and a database D, compute the number of tuples in the result of Q over D) [13, 14, 4, 29]. Sets of triple patterns are essentially CQs over a relational schema with a single ternary predicate. We now want to extend the study of tractable fragments of CQ evaluation to tractable fragments of evaluating well-designed SPARQL graph patterns. For the decision problem (i.e., given an RDF graph G, a well-designed SPARQL graph pattern P and a variable binding μ, check if μ is a solution), tractable fragments of CQ evaluation immediately carry over to tractable fragments of SPARQL evaluation. For the enumeration problem (i.e., given an

[1]There are several strongly related problems like asking if a given tuple is contained in the result of a given CQ over a given database, or asking if a given Boolean CQ evaluates to true over a given database, or query containment, etc. All these problems have straightforward reductions between each other. By slight abuse of notation we thus simply speak of "CQ evaluation" to refer to any of these problems.

RDF graph G and a well-designed SPARQL graph pattern P, compute all solutions μ) and the counting problem (i.e., given an RDF graph G and a well-designed SPARQL graph pattern P, compute the number of solutions μ), a much more detailed analysis is required. This is the main topic of this section.

But let us first look at the decision problem of evaluating well-designed SPARQL graph patterns. In [27], this problem was shown to be coNP-complete. For our representation of SPARQL graph patterns as QWDPT, a coNP test can work as follows. Let $\mathcal{T} = ((V, E, r), \mathcal{P})$ be a QWDPT and assume that it is in NR normal form (which can be computed in polynomial time). By using the characterization of the evaluation of QWDPTs provided in Lemma 3.21, in order to check if μ is a solution of $\mathcal{T}$ over G, the coNP-algorithm can first find a subtree $\mathcal{T}'$ of $\mathcal{T}$ rooted at r s.t. $\mathrm{dom}(\mu) = \mathrm{vars}(\mathcal{T}')$. Notice that if this subtree exists, then it is unique (since $\mathcal{T}$ is in NR normal form), and thus, this step can be done in polynomial time. Then the algorithm checks that $\mathcal{T}'$ is a maximal subtree such that $\mu \sqsubseteq [\![and(\mathcal{T}')]\!]_G$. The latter test requires coNP-power since we have to check that μ cannot be extended to match any of the sets of triple patterns at nodes "below" the leaf nodes of $\mathcal{T}'$. Note that this simple coNP-algorithm heavily relies on the NR normal form from Section 3.3 (the coNP-algorithm provided in [27] is considerably more complex).

Clearly, if all sets of triple patterns are from tractable fragments of CQ evaluation, the problem of checking if μ is a solution of $\mathcal{T}$ over G becomes tractable:

Corollary 5.1 *Suppose that we only consider QWDPTs (and thus well-designed SPARQL graph patterns), where for each node t the set P_t of triple patterns is from tractable fragments of CQ evaluation. Then the decision problem of such QWDPTs is also tractable.*

Note that tractability is required for each set P_t individually, hence for different nodes t and t', the sets P_t and $P_{t'}$ may belong to different tractable fragments.

5.1 Enumeration of well-designed SPARQL

Recall that an appropriate notion of *tractable* enumeration has to take the size of the output into account. Indeed, even for a single set of triple patterns, the set of solutions can be exponentially big. Hence, our goal is to identify conditions under which the enumeration of the solutions is feasible with *polynomial delay*, i.e., the time to either compute the next solution or to detect that no further solution exists must be polynomially bounded in the input size. For instance, acyclic CQs and CQs of bounded treewidth or hypertreewidth [41, 10, 13, 17] have this property.

Polynomial delay algorithms are usually implemented in the form of *iterators*. I.e., they are implemented in terms of functions **next**() and **hasNext**(), where **next**() returns the next solution, while **hasNext**() returns if there exists yet another solution. For polynomial delay algorithms, both functions run in polynomial time.

Following the presentation in [11], in order to increase readability we do not define the functions **next**() and **hasNext**() explicitly. Instead, the enumeration algorithm is described as an ordinary algorithm, and we consider iterators as constructs that take an enumeration algorithm as argument and provide the **next**() and **hasNext**() functions. I.e., consider an iterator I := **new** Iterator($E(x)$) for an enumeration algorithm E with input x. In response to I.**next**() being called, the iterator executes $E(x)$ until it encounters **output**(A) for the first time. Then the execution of E is interrupted, and A is returned as the result of I.**next**(). At the next call of I.**next**(), the execution of E is continued at the position where it was last interrupted, i.e. right after the last **output**(.) command executed (and the last state of E is restored). Once E terminates (instead of being interrupted), no further answer exists. The function **hasNext**() can be either implemented by continuing the execution of E and checking if another result is generated or not, or (like in our case) it is implemented by checking the current state of E.

```
Enumerate(t,μ)
 1: cqit := new Iterator(EnumerateCQ(P_t, μ));
 2: while( cqit.hasNext() ){
    // let t_1, ..., t_k be the children of t
 3:   maxi := 0;
 4:   μ_curr := cqit.next();
 5:   for( i = 1 to k ){
 6:     it_i := new Iterator(Enumerate(t_i, μ_curr));
 7:     flag_i := it_i.hasNext();
 8:     if(flag_i){
 9:       μ_i := it_i.next();
10:       maxi := i;
11:     }
12:   }
13:   if( ⋀_{i=1}^{k} ¬flag_i ){
14:     output(μ_curr);
15:     continue;
16:   }
17:   repeat{
18:     output( μ_curr ∪ ⋃_{1≤i≤k∧flag_i=true} μ_i );
19:     continueflag := false;
20:     for( i = maxi downto 1 ){
21:       if(it_i.hasNext()){
22:         μ_i := it_i.next();
23:         continueflag := true;
24:         for( j = i+1 to maxi ){
25:           if( flag_j ) {
26:             it_j := new Iterator(Enumerate(t_j, μ_curr));
27:             μ_j := it_j.next();
28:           }
29:         }
30:         i := 0; // leave the for-loop
31:       }
32:     }
33:   } until( ¬continueflag)
34: }
```

Figure 1: Iterator for SPARQL tree patterns.

In the following, assume a QWDPT $\mathcal{T} = ((V, E, r), \mathcal{P})$ to be evaluated over some RDF graph G. The algorithm in Figure 1 assumes the existence of some enumeration algorithm EnumerateCQ(P_t,μ) that, given a set P_t of triple patterns and a partial variable assignment μ returns all extensions of μ to P_t over G. We consider EnumerateCQ(P_t,μ) as a black box. The idea of our (recursive) enumeration algorithm Enumerate(t,μ) is as follows. For $t \in V$ and a partial assignment μ, the algorithm first checks if μ can be extended to P_t (lines 1–2). For each such extension μ_{curr}, it checks recursively for each child node t_i if there exists an extension of μ_{curr} to P_{t_i} (lines 5–7). Next, for each t_i that has such an extension the first solution is stored, together with the biggest index i s.t. t_i provides a solution (lines 8–10). If μ_{curr} cannot be extended to any child node, then the algorithm just returns μ_{curr} as one extension of μ to the complete subtree rooted at t (line 14; recall that the execution of the **output**(.) statement ends the execution of the call to **next**(), and the control flow is returned to the caller), and then considers the next extension of μ on t (line

15 jumps to the next iteration of the while loop in line 2). If on the other hand μ_{curr} can be extended to some children of t (lines 17 – 33), the algorithm enumerates all these extensions as follows. (For the sake of simplicity, in the following we only consider the ℓ children to which μ_{curr} can be extended.) In lines 17–33, all possible solutions are created that can be built from combining the extensions of μ_{curr} to $t_1, \ldots, t_\ell$. Note that the first possible extension for each t_i was saved in μ_i in line 9. After returning this solution (line 18), the solutions are enumerated by iterating over the solutions for $t_1, \ldots, t_\ell$ as follows. First, the child node t_i with the highest index i is identified that has yet another solution (lines 20–21). This extension is saved in μ_i (line 22), and for all children t_j with $i < j \leq \ell$ the iterators are reset to the first extension of μ_{curr}, which is stored in μ_j (lines 24–27). If such an index i exists, the new solution is returned in the next iteration of the repeat loop (line 18), otherwise all extensions of μ_{curr} have already been returned, and the algorithm terminates.

Theorem 5.2 *The problem of enumerating all solutions of a QWDPT (and hence, of a well-designed SPARQL graph pattern) can be reduced in polynomial time (by a Turing reduction) to the problem of enumerating all solutions of CQs.*

PROOF IDEA. The iterator described in Figure 1 reduces the problem of enumerating all solutions of a QWDPT to the problem of enumerating all solutions of CQs. Moreover, neglecting the cost of the calls to the iterator for CQs, the algorithm in Figure 1 clearly works in polynomial time. □

We may thus conclude that any tractability results for CQs immediately carry over to well-designed SPARQL graph patterns.

Theorem 5.3 *Suppose that we only consider QWDPTs where the sets of triple patterns at each node are from tractable fragments of CQ evaluation. Then the enumeration problem of such QWDPTs can be solved with polynomial-time delay.*

An inspection of our iterator for QWDPTs reveals that Theorem 5.3 could be further strengthened: for the tractability of the enumeration problem, it is sufficient that the sets of triple patterns are from tractable fragments of CQ evaluation *after considering all "old variables" at each node as constants*. In general, such an elimination of variables from a CQ may yield a significantly bigger tractable class.

5.2 Counting of well-designed SPARQL

Before studying the counting problem of well-designed SPARQL graph patterns, we recall some basic notions and results from counting complexity. The most intensively studied counting complexity class is $\#\mathsf{P}$. It contains those problems which consist in counting the number of accepting computation paths of a non-deterministic polynomial-time Turing machine. In other words, $\#\mathsf{P}$ captures the counting problems corresponding to decision problems in NP. Alternatively, *counting problems* can be presented using a *witness* function R which for every input x returns a set $R(x)$ of *witnesses* for x. Every witness function gives rise to the *counting problem* $\# \cdot R$ defined as follows: given a string $x \in \Sigma^*$, find the cardinality $|R(x)|$ of the *witness* set $R(x)$. According to [20], if $\mathcal{C}$ is a complexity class of decision problems, we define $\# \cdot \mathcal{C}$ as the class of all counting problems whose witness function R satisfies the following conditions.

1. There is a polynomial $p(n)$ such that for every $x \in \Sigma^*$ and every $y \in R(x)$ we have $|y| \leq p(|x|)$;
2. The problem "given x and y, is $y \in R(x)$?" is in $\mathcal{C}$.

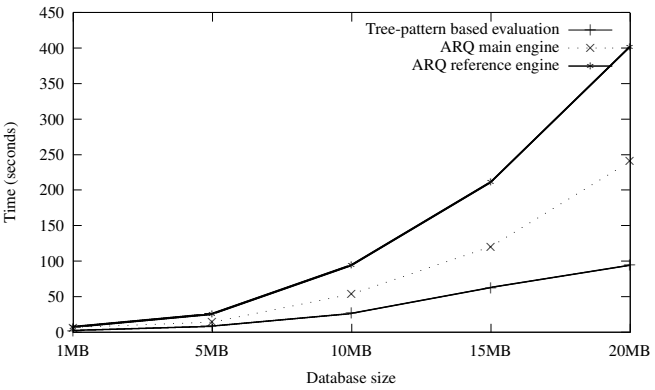


Figure 2: Performance of the QWDPT-based evaluation compared with non-optimized and optimized ARQ, for a modified version of Query 7 of SP^{2}Bench benchmark [33].

It is easy to verify that $\#\mathsf{P} = \# \cdot \mathsf{P}$. Moreover, the following inclusions hold [20]: $\#\mathsf{P} \subseteq \# \cdot \mathsf{NP} \subseteq \# \cdot \mathsf{coNP}$.

Counting the number of answer tuples of a CQ is $\#\mathsf{P}$-complete for CQs with free variables only and $\# \cdot \mathsf{NP}$-complete for CQs which may contain existential quantifiers [5]. Recently, it has been shown that the counting problem of CQs where not all variables are free, remains $\#\mathsf{P}$-complete even if we restrict ourselves to acyclic CQs. In contrast, the counting problem of CQs without existential quantifiers becomes tractable for CQs of bounded hypertree-width [29]. In this section, we extend the study of the counting problem from CQs to well-designed SPARQL patterns showing that, as opposed to the enumeration problem, tractability does not carry over. More specifically, we prove the following theorems.

Theorem 5.4 *The problem of counting all solutions of a QWDPT (and hence, of a well-designed SPARQL graph pattern) is $\#\cdot\mathsf{coNP}$-complete.*

PROOF IDEA. We only discuss the $\#\cdot\mathsf{coNP}$-*membership*. Consider the following witness function R: it takes as argument a pair $(\mathcal{T}, G)$ consisting of a QWDPT $\mathcal{T}$ and a graph G. Then R maps each pair $(\mathcal{T}, G)$ to the set of solutions of $\mathcal{T}$ over G. The problem of counting all solutions of a QWDPT $\mathcal{T}$ over a graph G corresponds to the counting problem $\# \cdot R$. The $\# \cdot \mathsf{coNP}$-membership follows from the fact that it can be tested in coNP if a given mapping μ is a solution of $\mathcal{T}$ over G [27]. □

Theorem 5.5 *The problem of counting all solutions of a QWDPT (and hence, of a well-designed SPARQL graph pattern) is in $\#\mathsf{P}$ if the sets of triple patterns at each node are from tractable fragments of CQ evaluation. The problem remains $\#\mathsf{P}$-complete even if all sets of triple patterns correspond to acyclic CQs.*

PROOF IDEA. The $\#\mathsf{P}$-*hardness* is shown by reduction from the problem of counting the number of perfect matchings in a bipartite graph, which is a classical $\#\mathsf{P}$-complete problem [39].

For the $\#\mathsf{P}$-*membership*, we consider the same witness function R as in the proof of Theorem 5.4. If all sets of triple patterns are from tractable fragments of CQ evaluation, then we can test in PTIME if a given mapping μ is a solution of $\mathcal{T}$ over G. □

6. CONCLUDING REMARKS

Static analysis is a fundamental task in query optimization. In this paper we have studied this problem for SPARQL queries. One

of our main contributions is the introduction of an abstract representation of well-designed queries as trees that resemble relational query plans. This representation allowed us to provide transformation rules and normal forms that proved to be useful when studying equivalence, containment, and tractable query enumeration. An interesting line for future work is the inclusion of more SPARQL operators to our study, in particular, projection and filtering.

On the practical side, we are currently developing a prototype based on the results presented in this paper, by modifying the non-optimized version of the ARQ SPARQL engine [45]. Our preliminary results are encouraging. In particular, the top-down evaluation of QWDPTs performs very well whenever a node in some branch of the tree provides a small set of results. We created examples of queries along with data which represent this case, by using the SP^{2}Bench benchmark [33], in particular modifying *Query 7* of the benchmark (that uses various levels of OPT nesting). In this scenario, our prototype even outperforms ARQ's optimized engine (Figure 2). An extension of these experiments to a systematic comparison of our approach with previous ones is left for future work. We have also implemented some transformation rules, the normal forms, and equivalence test. An on-line interface for our prototype is available from `http://db.ing.puc.cl/sparql-algebra`.

Acknowledgements

This work was funded in part by Marie Curie action IRSES under Grant No. 24761 (Net2), and by the Vienna Science and Technology Fund (WWTF) through project ICT08-032. Jorge Pérez was supported by Fondecyt grant 11110404 and by VID grant U-Inicia 11/04 Universidad de Chile.

7. REFERENCES

[1] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable semantic web data management using vertical partitioning. In *VLDB*, pages 411–422. ACM, 2007.

[2] R. Angles and C. Gutierrez. The expressive power of SPARQL. In *ISWC*, pages 114–129. Springer, 2008.

[3] M. Arenas and J. Pérez. Querying Semantic Web data with SPARQL. In *PODS*, pages 305–316. ACM, 2011.

[4] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222. Springer, 2007.

[5] M. Bauland, P. Chapdelaine, N. Creignou, M. Hermann, and H. Vollmer. An algebraic approach to the complexity of generalized conjunctive queries. In *SAT 2004 - Revised Selected Papers*, pages 30–45. Springer, 2005.

[6] T. Berners-Lee. Linked data – design issues. http://www.w3.org/DesignIssues/LinkedData.html, 2006.

[7] C. Bizer, T. Heath, and T. Berners-Lee. Linked data - the story so far. *Int. J. Semantic Web Inf. Syst.*, 5(3):1–22, 2009.

[8] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. ACM, 1977.

[9] M. Chekol, J. Euzenat, P. Genevès, and N. Layaïda. PSPARQL query containment. In *DBPL*, 2011.

[10] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000.

[11] S. Cohen and I. Fadida and Y. Kanza and B. Kimelfeld and Y. Sagiv. Full Disjunctions: Polynomial-Delay Iterators in Action. In *VLDB*, pages 739–750. ACM, 2006.

[12] A. Durand, M. Hermann, and P. G. Kolaitis. Subtractive reductions and complete problems for counting complexity classes. *Theor. Comput. Sci.*, 340(3):496–513, 2005.

[13] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[14] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004.

[15] M. A. Gallego, J. D. Fernández, M. A. Martínez-Prieto, and P. de la Fuente. An empirical study of real-world SPARQL queries. In *USEWOD*, 2011.

[16] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[17] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[18] G. Greco and F. Scarcello. The power of tree projections: local consistency, greedy algorithms, and larger islands of tractability. In *PODS 2010*, pages 327–338. ACM, 2010.

[19] C. Gutierrez, C. A. Hurtado, A. O. Mendelzon, and J. Pérez. Foundations of semantic web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[20] L. A. Hemaspaandra and H. Vollmer. The satanic notations: Counting classes beyond #P and other definitional adventures. *SIGACT News, Complexity Theory Column 8*, 26(1):2–13, 1995.

[21] Y. Kanza and W. Nutt and Y. Sagiv. Querying Incomplete Information in Semistructured Data. *J. Comput. Syst. Sci.*, 64(3):655–693, 2002.

[22] P.-Å. Larson and J. Zhou. View matching for outer-join views. In *VLDB*, pages 445–456. ACM, 2005.

[23] O. Lassila and R. Swick. Resource description framework (RDF) model and syntax. *W3C Recommnedation*, January 1999.

[24] T. Neumann and G. Weikum. The rdf-3x engine for scalable management of rdf data. *VLDB J.*, 19(1):91–113, 2010.

[25] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. In *ISWC*, pages 30–43. Springer, 2006.

[26] J. Pérez, M. Arenas, and C. Gutierrez. Semantics of SPARQL. Technical Report, Universidad de Chile TR/DCC-2006-17, October 2006.

[27] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[28] F. Picalausa and S. Vansummeren. What are real SPARQL queries like? In *SWIM*, pages 7:1–7:6. ACM, 2011.

[29] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. In *AMW*. CEUR-WS.org, 2011.

[30] A. Polleres. From SPARQL to rules (and back). In *WWW*, pages 787–796. ACM, 2007.

[31] E. Prud'Hommeaux and A. Seaborne. SPARQL query language for RDF. *W3C Recommnedation*, January 2008.

[32] M. Schmidt, T. Hornung, N. Küchlin, G. Lausen, and C. Pinkel. An experimental comparison of RDF data management approaches in a SPARQL benchmark scenario. In *ISWC*, pages 82–97. Springer, 2008.

[33] M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. SP^{2}Bench: A SPARQL performance benchmark. In *ICDE*, pages 222–233. IEEE, 2009.

[34] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, pages 4–33. ACM, 2010.

[35] G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of RDF/S query patterns. In *ISWC*, pages 607–623. Springer, 2005.

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

[37] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *WWW*, pages 595–604. ACM, 2008.

[38] J. D. Ullman. Information integration using logical views. In *ICDT*, pages 19–40. Springer, 1997.

[39] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.

[40] C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[41] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94. IEEE, 1981.

[42] http://data.gov.uk.

[43] http://www.data.gov.

[44] http://DBpedia.org/sparql.

[45] ARQ. http://sourceforge.net/projects/jena/files/ARQ/.