Semantics and Optimization of the SPARQL 1.1 Federation Extension

Carlos Buil-Aranda[1], Marcelo Arenas[2], and Oscar Corcho[1]

[1] Ontology Engineering Group, Facultad de Informática, UPM, Spain
[2] Department of Computer Science, PUC Chile

Abstract. The W3C SPARQL working group is defining the new SPARQL 1.1 query language. The current working draft of SPARQL 1.1 focuses mainly on the description of the language. In this paper, we provide a formalization of the syntax and semantics of the SPARQL 1.1 federation extension, an important fragment of the language that has not yet received much attention. Besides, we propose optimization techniques for this fragment, provide an implementation of the fragment including these techniques, and carry out a series of experiments that show that our optimization procedures significantly speed up the query evaluation process.

1 Introduction

The recent years have witnessed a constant growth in the amount of RDF data available, exposed by means of Linked Data-enabled URLs and SPARQL endpoints. Several non-exhaustive, and sometimes out-of-date, lists of SPARQL endpoints or data catalogs are available in different formats (from wiki-based HTML pages to SPARQL endpoints using data catalog description vocabularies). Besides, most of these datasets are interlinked, what allows navigating through them and facilitates building complex queries combining data from heterogeneous datasets.

These SPARQL endpoints accept queries written in SPARQL and adhere to the SPARQL protocol, as defined by the W3C recommendation. However, the current SPARQL recommendation has an important limitation in defining and executing queries that span across distributed datasets, since it only considers the possibility of executing these queries in isolated SPARQL endpoints. Hence users willing to federate queries across a number of SPARQL endpoints have been forced to create ad-hoc extensions of the query language or to include additional information about data sources in the configuration of their SPARQL endpoint servers [14,15]. This has led to the inclusion of query federation extensions in the current SPARQL 1.1 working draft [12] (together with other extensions that are out of the scope of this paper), which are studied in detail in order to generate a new W3C recommendation in the coming months.

The federation extension of SPARQL 1.1 includes two new operators in the query language: SERVICE and BINDINGS. The former allows specifying, inside a SPARQL query, the SPARQL query service in which a portion of the query will be executed. This query service may be known at the time of building the query, and hence the SERVICE operator will already specify the IRI of the SPARQL endpoint where it will be executed; or may be retrieved at query execution time after executing an initial SPARQL query

G. Antoniou et al. (Eds.): ESWC 2011, Part II, LNCS 6644, pp. 1–15, 2011.
© Springer-Verlag Berlin Heidelberg 2011

fragment in one of the aforementioned RDF-enabled data catalogs, so that potential SPARQL endpoints that can answer the rest of the query can be obtained and used. The latter (BINDINGS) allows transferring results that are used to constrain a query, and which will normally come from previous executions of other queries or from constraints specified in user interfaces that then transform these into SPARQL queries.

Till now, most of the work done on federation extensions in the context of the W3C working group has been focused on the description of the language grammar. In this paper we complement this work with the formalization of the syntax and semantics of these federation extensions of SPARQL 1.1, and with the definition of the constraints that have to be considered in their use (which is currently not too restricted) in order to be able to provide pragmatic implementations of query evaluators. As an extreme example of bad performance, we may imagine a query that uses the SERVICE operator with a free variable to specify the SPARQL endpoint where the rest of the query has to be evaluated. We may imagine that a naïve implementation may need to go through all existing SPARQL endpoints on the Web evaluating that query fragment before providing a result, something that can be considered infeasible in practical terms. For our purpose, we define the notions of service-boundedness and service-safeness, which ensure that the SERVICE operator can be safely evaluated.

Besides, we implement the optimizations proposed in [11], using the notion of well-designed patterns, which prove to be effective in the optimization of queries that contain the OPTIONAL operator, the most costly operator in SPARQL [11,17]. This has also important implications in the number of tuples being transferred and joined in federated queries, and hence our implementation benefits from this.

As a result of our work, we have not only formalized these notions, but we have also implemented a system that supports the current SPARQL 1.1 federation extensions and makes use of these optimizations. This system, SPARQL-DQP (which stands for SPARQL Distributed Query Processing), is built on top of the OGSA-DAI and OGSA-DQP infrastructure [3,10], what provides additional robustness to deal with large amounts of data in distributed settings, supporting for example an indirect access mode that is normally used in the development of data-intensive workflows. We have evaluated our system using a small benchmark of real SPARQL 1.1 queries from the bioinformatics domain, and compared it with other similar systems, in some cases adapting the queries to their own ad-hoc SPARQL extensions, so that the benefits of our implementation can be illustrated.

With this work, we aim at advancing to the current state of the art hoping to include it in the next versions of the SPARQL working drafts, and providing SPARQL-DQP as one of the reference implementations of this part of the recommendation. We also hope that the initial benchmark that we have defined can be extended and stabilized in order to provide a good evaluation framework, complementing existing benchmarks.

Organization of the paper. In Section 2, we describe the syntax and semantics of the SPARQL 1.1 federation extension. In Section 3, we introduce the notions of service-safeness, which ensures that the SERVICE operator can be safely evaluated. In Section 4, we present some optimization techniques for the evaluation of the SPARQL 1.1 federation extension. Finally, in Section 5, we present our implementation as well as an experimental evaluation of it.

2 Syntax and Semantics of the SPARQL 1.1 Federation Extension

In this section, we give an algebraic formalization of the SPARQL 1.1 federation extension over simple RDF, that is, RDF without RDFS vocabulary and literal rules. Our starting point is the existing formalization of SPARQL described in [11], to which we add the operators SERVICE and BINDINGS proposed in [12].

We introduce first the necessary notions about RDF (taken mainly from [11]). Assume there are pairwise disjoint infinite sets I, B, and L (IRIs [6], Blank nodes, and Literals, respectively). Then a triple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an *RDF triple*. In this tuple, s is the *subject*, p the *predicate* and o the *object*. An *RDF graph* is a set of RDF triples. Moreover, assume the existence of an infinite set V of variables disjoint from the above sets, and leave UNBOUND to be a reserve word that does not belong to any of the sets mentioned previously.

2.1 Syntax of the Federation Extension

The official syntax of SPARQL [13] considers operators `OPTIONAL`, `UNION`, `FILTER`, `SELECT` and concatenation via a point symbol (`.`), to construct graph pattern expressions. Operators `SERVICE` and `BINDINGS` are introduced in the SPARQL 1.1 federation extension, the former for allowing users to direct a portion of a query to a particular SPARQL endpoint, and the latter for transferring results that are used to constrain a query. The syntax of the language also considers `{ }` to group patterns, and some implicit rules of precedence and association. In order to avoid ambiguities in the parsing, we follow the approach proposed in [11], and we first present the syntax of SPARQL graph patterns in a more traditional algebraic formalism, using operators AND (`.`), UNION (`UNION`), OPT (`OPTIONAL`), FILTER (`FILTER`) and SERVICE (`SERVICE`), then we introduce the syntax of BINDINGS queries, which use the BINDINGS operator (`BINDINGS`), and we conclude by defining the syntax of SELECT queries, which use the SELECT operator (`SELECT`). More precisely, a SPARQL graph pattern expression is defined recursively as follows:

(1) A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a triple pattern).
(2) If P_1 and P_2 are graph patterns, then expressions $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, and $(P_1 \text{ UNION } P_2)$ are graph patterns.
(3) If P is a graph pattern and R is a *SPARQL built-in condition*, then the expression $(P \text{ FILTER } R)$ is a graph pattern.
(4) If P is a graph pattern and $a \in (I \cup V)$, then $(\text{SERVICE } a\ P)$ is a graph pattern.

Moreover, a SPARQL BINDINGS query is defined as follows:

(5) If P is a graph pattern, S is a nonempty list of pairwise distinct variables and $\{A_1, \ldots, A_n\}$ is a nonempty set of lists such that for every $i \in \{1, \ldots, n\}$, it holds that A_i and S have the same length and each element in A_i belongs to $(I \cup L \cup \{\text{UNBOUND}\})$, then $(P \text{ BINDINGS } S\ \{A_1, \ldots, A_n\})$ is a BINDINGS query.

Finally, assuming that P is either a graph pattern or a BINDINGS query, let $\text{var}(P)$ be the set of variables mentioned in P. Then a SPARQL SELECT query is defined as:

(6) If P is either a graph pattern or a BINDINGS query, and W is a set of variables such that $W \subseteq \text{var}(P)$, then $(\text{SELECT } W\ P)$ is a SELECT query.

It is important to notice that the rules (1)–(3) above were introduced in [11], while we formalize in the rules (4)–(6) the federation extension of SPARQL proposed in [12].

In the previous definition, we use the notion of built-in condition for the filter operator. A SPARQL built-in condition is constructed using elements of the set $(I \cup L \cup V)$ and constants, logical connectives ($\neg$, $\wedge$, $\vee$), inequality symbols ($<$, $\leq$, $\geq$, $>$), the equality symbol ($=$), unary predicates like bound, isBlank, and isIRI, plus other features (see [13] for a complete list). Due to the lack of space, we restrict in this paper to the fragment of SPARQL where the built-in condition is a Boolean combination of terms constructed by using $=$ and bound, that is: (1) if $?X, ?Y \in V$ and $c \in (I \cup L)$, then $\text{bound}(?X)$, $?X = c$ and $?X = ?Y$ are built-in conditions, and (2) if R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$ and $(R_1 \wedge R_2)$ are built-in conditions. It should be noticed that the results of the paper can be easily extended to the other built-in predicates in SPARQL.

Let P be either a graph pattern or a BINDINGS query or a SELECT query. In the rest of the paper, we use $\text{var}(P)$ to denote the set of variables occurring in P. Similarly, for a built-in condition R, we use $\text{var}(R)$ to denote the set of variables occurring in R.

2.2 Semantics of the Federation Extension

To define the semantics of SPARQL queries, we need to introduce some extra terminology from [11]. A mapping μ from V to $(I \cup B \cup L)$ is a partial function $\mu : V \to (I \cup B \cup L)$. Abusing notation, for a triple pattern t we denote by $\mu(t)$ the triple obtained by replacing the variables in t according to μ. The domain of μ, denoted by $\text{dom}(\mu)$, is the subset of V where μ is defined. Two mappings μ_1 and μ_2 are compatible when for all $?X \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(?X) = \mu_2(?X)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping.

Let Ω_1 and Ω_2 be sets of mappings. Then the join of, the union of, the difference between and the left outer-join between Ω_1 and Ω_2 are defined as follows [11]:

$$\begin{aligned}
\Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1, \mu_2 \text{ are compatible mappings}\},\\
\Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\},\\
\Omega_1 \smallsetminus \Omega_2 &= \{\mu \in \Omega_1 \mid \text{ for all } \mu' \in \Omega_2, \mu \text{ and } \mu' \text{ are not compatible}\},\\
\Omega_1 ⟕ \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \smallsetminus \Omega_2).
\end{aligned}$$

Next we use the preceding operators to give semantics to graph pattern expressions, BINDINGS queries and SELECT queries. More specifically, we define this semantics as a function $[\![\cdot]\!]_G$, which takes as input any of these types of queries and returns a set of mappings. In this definition, we assume given a partial function ep from the set I of IRIs such that for every $c \in I$, if $\text{ep}(c)$ is defined, then $\text{ep}(c)$ is an RDF graph. Intuitively, function ep is defined for an element $c \in I$ ($c \in \text{dom}(\text{ep})$) if and only if c is the IRI of a SPARQL endpoint, and $\text{ep}(c)$ is the default RDF graph of that endpoint[1].

[1] For simplicity, we only assume a single (default) graph and no named graphs per remote SPARQL endpoint.

Moreover, in this definition $\mu_\emptyset$ represents the mapping with empty domain (which is compatible with any other mapping).

The evaluation of a graph pattern P over an RDF graph G, denoted by $[\![P]\!]_G$, is defined recursively as follows (due to the lack of space, we refer the reader to the extended version of the paper for the definition of the semantics of the FILTER operator):

(1) If P is a triple pattern t, then $[\![P]\!]_G = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G\}$.
(2) If P is $(P_1 \text{ AND } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_G$.
(3) If P is $(P_1 \text{ OPT } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G ⟕ [\![P_2]\!]_G$.
(4) If P is $(P_1 \text{ UNION } P_2)$, then $[\![P]\!]_G = [\![P_1]\!]_G \cup [\![P_2]\!]_G$.
(5) If P is $(\text{SERVICE } c\ P_1)$ with $c \in I$, then

$$[\![P]\!]_G = \begin{cases} [\![P_1]\!]_{\text{ep}(c)} & \text{if } c \in \text{dom(ep)} \\ \{\mu_\emptyset\} & \text{otherwise} \end{cases}$$

(6) If P is $(\text{SERVICE } ?X\ P_1)$ with $?X \in V$, then $[\![P]\!]_G$ is equal to:

$$\bigcup_{c \in I} \Big\{ \mu \mid \text{there exists } \mu' \in [\![(\text{SERVICE } c\ P_1)]\!]_G \text{ s.t. } \text{dom}(\mu) = (\text{dom}(\mu') \cup \{?X\}),$$
$$\mu(?X) = c \text{ and } \mu(?Y) = \mu'(?Y) \text{ for every } ?Y \in \text{dom}(\mu') \Big\}$$

Moreover, the semantics of BINDINGS queries is defined as follows. Given a list $S = [?X_1, \ldots, ?X_\ell]$ of pairwise distinct variables, where $\ell \geq 1$, and a list $A = [a_1, \ldots, a_\ell]$ of values from $(I \cup L \cup \{\text{UNBOUND}\})$, let $\mu_{S,A}$ be a mapping with domain $\{?X_i \mid i \in \{1, \ldots, \ell\} \text{ and } a_i \in (I \cup L)\}$ and such that $\mu_{S,A}(?X_i) = a_i$ for every $?X_i \in \text{dom}(\mu_{S,A})$. Then

(7) If $P = (P_1 \text{ BINDINGS } S\ \{A_1, \ldots, A_n\})$ is a BINDINGS query:

$$[\![P]\!]_G = [\![P_1]\!]_G \bowtie \{\mu_{S,A_1}, \ldots, \mu_{S,A_n}\}.$$

Finally, the semantics of SELECT queries is defined as follows. Given a mapping $\mu : V \to (I \cup B \cup L)$ and a set of variables $W \subseteq V$, the restriction of μ to W, denoted by $\mu_{|W}$, is a mapping such that $\text{dom}(\mu_{|W}) = (\text{dom}(\mu) \cap W)$ and $\mu_{|W}(?X) = \mu(?X)$ for every $?X \in (\text{dom}(\mu) \cap W)$. Then

(8) If $P = (\text{SELECT } W\ P_1)$ is a SELECT query: $[\![P]\!]_G = \{\mu_{|W} \mid \mu \in [\![P_1]\!]_G\}$.

It is important to notice that the rules (1)–(4) above were introduced in [11], while we propose in the rules (5)–(8) a semantics for the operators SERVICE and BINDINGS introduced in [12]. Intuitively, if $c \in I$ is the IRI of a SPARQL endpoint, then the idea behind the definition of $(\text{SERVICE } c\ P_1)$ is to evaluate query P_1 in the SPARQL endpoint specified by c. On the other hand, if $c \in I$ is not the IRI of a SPARQL endpoint, then $(\text{SERVICE } c\ P_1)$ leaves unbounded all the variables in P_1, as this query cannot be evaluated in this case. This idea is formalized by making $\mu_\emptyset$ the only mapping in the evaluation of $(\text{SERVICE } c\ P_1)$ if $c \notin \text{dom(ep)}$. In the same way, $(\text{SERVICE } ?X\ P_1)$

is defined by considering all the possible IRIs for the variable $?X$, that is, all the values $c \in I$. In fact, $(\text{SERVICE } ?X \ P_1)$ is defined as the union of the evaluation of the graph patterns $(\text{SERVICE } \ c \ P_1)$ for the values $c \in I$, but also storing in $?X$ the IRIs from where the values of the variables in P_1 are coming from. Finally, the idea behind the definition of $(P_1 \ \text{ BINDINGS } \ S \ \{A_1, \ldots, A_n\})$ is to constrain the values of the variables in S to the values specified in $A_1, \ldots, A_n$.

Example 1. Assume that G is an RDF graph that uses triples of the form $(a, \text{service_address}, b)$ to indicate that a SPARQL endpoint with name a is located at the IRI b. Moreover, let P be the following SPARQL query:

$$\Big[\text{SELECT } \{?X, ?N, ?E\} \\ \Big(\Big((?X, \text{service_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y \ (?N, \text{email}, ?E))\Big) \\ \text{BINDINGS } [?N] \ \{[\,\text{John}\,], [\,\text{Peter}\,]\}\Big)\Big]$$

Query P is used to compute the list of names and email addresses that can be retrieved from the SPARQL endpoints stored in an RDF graph. In fact, if $\mu \in [\![P]\!]_G$, then $\mu(?X)$ is the name of a SPARQL endpoint stored in G, $\mu(?N)$ is the name of a person stored in that SPARQL endpoint and $\mu(?E)$ is the email address of that person. Moreover, the operator BINDINGS in this query is used to filter the values of the variable $?N$. Specifically, if $\mu \in [\![P]\!]_G$, then $\mu(?N)$ is either John or Peter. □

The goal of the rules (5)–(8) is to define in an unambiguous way what the result of evaluating an expression containing the operators SERVICE and BINDINGS should be. As such, these rules should not be considered as an implementation of the language. In fact, a direct implementation of the rule (6), that defines the semantics of a pattern of the form $(\text{SERVICE } ?X \ P_1)$, would involve evaluating a particular query in every possible SPARQL endpoint, which is obviously infeasible in practice. In the next section, we face this issue and, in particular, we introduce a syntactic condition on SPARQL queries that ensures that a pattern of the form $(\text{SERVICE } ?X \ P_1)$ can be evaluated by only considering a finite set of SPARQL endpoints, whose IRIs are actually taken from the RDF graph where the query is being evaluated.

3 On Evaluating the SERVICE Operator

As we pointed out in the previous section, the evaluation of a pattern of the form $(\text{SERVICE } ?X \ P)$ is infeasible unless the variable $?X$ is bound to a finite set of IRIs. This notion of *boundedness* is one of the most significant and unclear concepts in the SPARQL federation extension. In fact, the current version of the specification [12] only specifies that a variable $?X$ in a pattern of the form $(\text{SERVICE } ?X \ P)$ *must be bound*, but without providing a formal definition of what that means. Here we provide a formalization of this concept, studying the complexity issues associated with it.

3.1 The Notion of Boundedness

In Example 1, we present a SPARQL query containing a pattern (SERVICE $?Y$ $(?N, \text{email}, ?E)$). Given that variable $?Y$ is used to store the address of a remote SPARQL endpoint to be queried, it is important to assign a value to $?Y$ prior to the evaluation of the SERVICE pattern. In the case of the query in Example 1, this needs of a simple strategy: given an RDF graph G, first compute $[\![(?X, \text{service_address}, ?Y)]\!]_G$, and then for every μ in this set, compute $[\![(\text{SERVICE}\ a\ (?N, \text{email}, ?E))]\!]_G$ with $a = \mu(?Y)$. More generally, SPARQL pattern $(\text{SERVICE}\ ?Y\ (?N, \text{email}, ?E))$ can be evaluated in this case as only a finite set of values from the domain of G need to be considered as the possible values of $?Y$. This idea naturally gives rise to the following notion of boundedness for the variables of a SPARQL query. In the definition of this notion, $\text{dom}(G)$ refers to the domain of G, that is, the set of elements from $(I \cup B \cup L)$ that are mentioned in G, and $\text{dom}(P)$ refers to the set of elements from $(I \cup L)$ that are mentioned in P.

Definition 1 (Boundedness). *Let P be a SPARQL query and $?X \in \text{var}(P)$. Then $?X$ is bound in P if one of the following conditions holds:*

- *P is either a graph pattern or a BINDINGS query, and for every RDF graph G and mapping $\mu \in [\![P]\!]_G$, it holds that $?X \in \text{dom}(\mu)$ and $\mu(?X) \in (\text{dom}(G) \cup \text{dom}(P))$.*
- *P is a SELECT query $(\text{SELECT}\ W\ P_1)$ and $?X$ is bound in P_1.*

The BINDINGS operator can make a variable $?X$ in a query P to be bound by assigning to it a fixed set of values. Given that these values are not necessarily mentioned in the RDF graph G where P is being evaluated, the previous definition first imposes the condition that $?X \in \text{dom}(\mu)$, and then not only considers the case $\mu(?X) \in \text{dom}(G)$ but also the case $\mu(?X) \in \text{dom}(P)$. As an example of the above definition, we note that variable $?Y$ is bound in the graph pattern

$$P_1 = ((?X, \text{service_address}, ?Y)\ \text{AND}\ (\text{SERVICE}\ ?Y\ (?N, \text{email}, ?E))),$$

as for every RDF graph G and mapping $\mu \in [\![P_1]\!]_G$, we know that $?Y \in \text{dom}(\mu)$ and $\mu(?Y) \in \text{dom}(G)$. Moreover, we also have that variable $?Y$ is bound in $(\text{SELECT}\ \{?X, ?N, ?E\}\ P_1)$ as $?Y$ is bound in graph pattern P_1.

A natural way to ensure that a SPARQL query P can be evaluated in practice is by imposing the restriction that for every sub-pattern $(\text{SERVICE}\ ?X\ P_1)$ of P, it holds that $?X$ is bound in P. However, in the following theorem we show that such a condition is undecidable and, thus, a SPARQL query engine would not be able to check it in order to ensure that a query can be evaluated.

Theorem 1. *The problem of verifying, given a SPARQL query P and a variable $?X \in \text{var}(P)$, whether $?X$ is bound in P is undecidable.*

The fact that the notion of boundedness is undecidable prevents one from using it as a restriction over the variables in SPARQL queries. To overcome this limitation, we introduce here a syntactic condition that ensures that a variable is bound in a pattern and that can be efficiently verified.

Definition 2 (Strong boundedness). *Let P be a SPARQL query. Then the set of strongly bound variables in P, denoted by* $\mathrm{SB}(P)$*, is recursively defined as follows:*

- *if $P = t$, where t is a triple pattern, then* $\mathrm{SB}(P) = \mathrm{var}(t)$*;*
- *if $P = (P_1 \text{ AND } P_2)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$*;*
- *if $P = (P_1 \text{ UNION } P_2)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cap \mathrm{SB}(P_2)$*;*
- *if $P = (P_1 \text{ OPT } P_2)$ or $P = (P_1 \text{ FILTER } R)$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1)$*;*
- *if $P = (\text{SERVICE } c\ P_1)$, with $c \in I$, or $P = (\text{SERVICE } ?X\ P_1)$, with $?X \in V$, then* $\mathrm{SB}(P) = \emptyset$*;*
- *if $P = (P_1 \text{ BINDINGS } S\ \{A_1, \ldots, A_n\})$, then* $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \{?X \mid$ *$?X$ is in S and for every $i \in \{1, \ldots, n\}$, it holds that* $?X \in \mathrm{dom}(\mu_{S,A_i})\}$.
- *if $P = (\text{SELECT } W\ P_1)$, then* $\mathrm{SB}(P) = (W \cap \mathrm{SB}(P_1))$.

The previous definition recursively collects from a SPARQL query P a set of variables that are guaranteed to be bound in P. For example, if P is a triple pattern t, then $\mathrm{SB}(P) = \mathrm{var}(t)$ as one knows that for every variable $?X \in \mathrm{var}(t)$ and for every RDF graph G, if $\mu \in [\![t]\!]_G$, then $?X \in \mathrm{dom}(\mu)$ and $\mu(?X) \in \mathrm{dom}(G)$. In the same way, if $P = (P_1 \text{ AND } P_2)$, then $\mathrm{SB}(P) = \mathrm{SB}(P_1) \cup \mathrm{SB}(P_2)$ as one knows that if $?X$ is bound in P_1 or in P_2, then $?X$ is bound in P. As a final example, notice that if $P = (P_1 \text{ BINDINGS } S\ \{A_1, \ldots, A_n\})$ and $?X$ is a variable mentioned in S such that $?X \in \mathrm{dom}(\mu_{S,A_i})$ for every $i \in \{1, \ldots, n\}$, then $?X \in \mathrm{SB}(P)$. In this case, one knows that $?X$ is bound in P since $[\![P]\!]_G = [\![P_1]\!]_G \bowtie \{\mu_{S,A_1}, \ldots, \mu_{S,A_n}\}$ and $?X$ is in the domain of each one of the mappings μ_{S,A_i}, which implies that $\mu(?X) \in \mathrm{dom}(P)$ for every $\mu \in [\![P]\!]_G$. In the following proposition, we formally show that our intuition about $\mathrm{SB}(P)$ is correct, in the sense that every variable in this set is bound in P.

Proposition 1. *For every SPARQL query P and variable $?X \in \mathrm{var}(P)$, if $?X \in$* $\mathrm{SB}(P)$*, then $?X$ is bound in P.*

Given a SPARQL query P and a variable $?X \in \mathrm{var}(P)$, it can be efficiently verified whether $?X$ is strongly bound in P. Thus, a natural and efficiently verifiable way to ensure that a SPARQL query P can be evaluated in practice is by imposing the restriction that for every sub-pattern $(\text{SERVICE } ?X\ P_1)$ of P, it holds that $?X$ is strongly bound in P. However, this notion still needs to be modified in order to be useful in practice, as shown by the following examples.

Example 2. Assume first that P_1 is the following graph pattern:

$$P_1 = ((?X, \text{service_description}, ?Z) \text{ UNION } ((?X, \text{service_address}, ?Y) \text{ AND } (\text{SERVICE } ?Y\ (?N, \text{email}, ?E)))).$$

That is, either $?X$ and $?Z$ store the name of a SPARQL endpoint and a description of its functionalities, or $?X$ and $?Y$ store the name of a SPARQL endpoint and the IRI where it is located (together with a list of names and email addresses retrieved from that location). Variable $?Y$ is neither bound nor strongly bound in P_1. However, there is a simple strategy that ensures that P_1 can be evaluated over an RDF graph G: first compute $[\![(?X, \text{service_description}, ?Z)]\!]_G$, then compute $[\![(?X, \text{service_address}, ?Y)]\!]_G$, and finally for every μ in the set

$[\![(?X, \text{service_address}, ?Y)]\!]_G$, compute $[\![(\text{SERVICE}\ a\ (?N, \text{email}, ?E))]\!]_G$ with $a = \mu(?Y)$. In fact, the reason why P_1 can be evaluated in this case is that $?Y$ is bound (and strongly bound) in the sub-pattern $((?X, \text{service_address}, ?Y)\ \text{AND}\ (\text{SERVICE}\ ?Y\ (?N, \text{email}, ?E)))$ of P_1.

As a second example, assume that G is an RDF graph that uses triples of the form $(a_1, \text{related_with}, a_2)$ to indicate that the SPARQL endpoints located at the IRIs a_1 and a_2 store related data. Moreover, assume that P_2 is the following graph pattern:

$$P_2 = ((?U_1, \text{related_with}, ?U_2)\ \text{AND}$$
$$(\text{SERVICE}\ ?U_1\ ((?N, \text{email}, ?E)\ \text{OPT}\ (\text{SERVICE}\ ?U_2\ (?N, \text{phone}, ?F))))).$$

When this query is evaluated over the RDF graph G, it returns for every tuple $(a_1, \text{related_with}, a_2)$ in G, the list of names and email addresses that that can be retrieved from the SPARQL endpoint located at a_1, together with the phone number for each person in this list for which this data can be retrieved from the SPARQL endpoint located at a_2 (recall that graph pattern $(\text{SERVICE}\ ?U_2\ (?N, \text{phone}, ?F))$ is nested inside the first SERVICE operator in P_2). To evaluate this query over an RDF graph, first it is necessary to determine the possible values for variable $?U_1$, and then to submit the query $((?N, \text{email}, ?E)\ \text{OPT}\ (\text{SERVICE}\ ?U_2\ (?N, \text{phone}, ?F)))$ to each one of the endpoints located at the IRIs stored in $?U_1$. In this case, variable $?U_2$ is bound (and also strongly bound) in P_2. However, this variable is not bound in the graph pattern $((?N, \text{email}, ?E)\ \text{OPT}\ (\text{SERVICE}\ ?U_2\ (?N, \text{phone}, ?F)))$, which has to be evaluated in some of the SPARQL endpoints stored in the RDF graph where P_2 is being evaluated, something that is infeasible in practice. Notice that the difficulties in evaluating P_2 are caused by the nesting of SERVICE operators (more precisely, by the fact that P_2 has a sub-pattern of the form $(\text{SERVICE}\ ?X_1\ Q_1)$, where Q_1 has in turn a sub-pattern of the form $(\text{SERVICE}\ ?X_2\ Q_2)$ such that $?X_2$ is bound in P_2 but not in Q_1). □

In the following section, we use the concept of strongly boundedness to define a notion that ensures that a SPARQL query containing the SERVICE operator can be evaluated in practice, and which takes into consideration the ideas presented in Example 2.

3.2 The Notion of Service-Safeness: Considering Sub-patterns and Nested Service Operators

The goal of this section is to provide a condition that ensures that a SPARQL query containing the SERVICE operator can be safely evaluated . To this end, we first need to introduce some terminology. Given a SPARQL query P, define $\mathcal{T}(P)$ as the *parse* tree of P. In this tree, every node corresponds to a sub-pattern of P. An example of a parse tree of a pattern Q is shown in Figure 1. In this figure, $u_1, u_2, u_3, u_4, u_5, u_6$ are the identifiers of the nodes of the tree, which are labeled with the sub-patterns of Q. It is important to notice that in this tree we do not make any distinction between the different operators in SPARQL, we just store the structure of the sub-patterns of a SPARQL query.

Tree $\mathcal{T}(P)$ is used to define the notion of service-boundedness, which extends the concept of boundedness, introduced in the previous section, to consider variables that

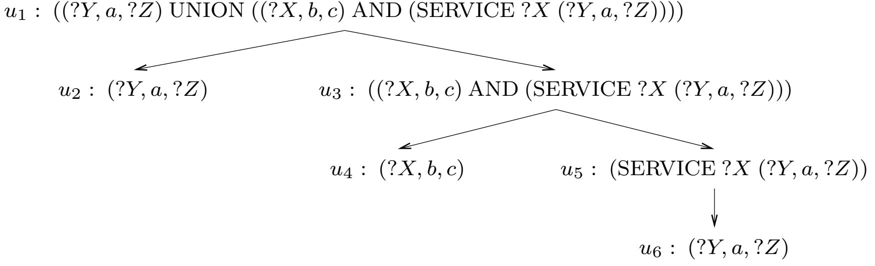

Fig. 1. Parse tree $\mathcal{T}(Q)$ for the graph pattern $Q = ((?Y, a, ?Z)$ UNION $((?X, b, c)$ AND $(\text{SERVICE } ?X\ (?Y, a, ?Z))))$

are bound inside sub-patterns and nested SERVICE operators. It should be noticed that these two features were identified in the previous section as important for the definition of a notion of boundedness (see Example 2).

Definition 3 (Service-boundedness). *A SPARQL query P is service-bound if for every node u of $\mathcal{T}(P)$ with label* $(\text{SERVICE } ?X\ P_1)$*, it holds that: (1) there exists a node v of $\mathcal{T}(P)$ with label P_2 such that v is an ancestor of u in $\mathcal{T}(P)$ and $?X$ is bound in P_2; (2) P_1 is service-bound.*

For example, query Q in Figure 1 is service-bound. In fact, condition (1) of Definition 3 is satisfied as u_5 is the only node in $\mathcal{T}(Q)$ having as label a SERVICE graph pattern, in this case $(\text{SERVICE } ?X\ (?Y, a, ?Z))$, and for the node u_3, it holds that: u_3 is an ancestor of u_5 in $\mathcal{T}(P)$, the label of u_3 is $P = ((?X, b, c)$ AND $(\text{SERVICE } ?X\ (?Y, a, ?Z)))$ and $?X$ is bound in P. Moreover, condition (2) of Definition 3 is satisfied as the sub-pattern $(?Y, a, ?Z)$ of the label of u_5 is also service-bound.

The notion of service-boundedness captures our intuition about the condition that a SPARQL query containing the SERVICE operator should satisfy. Unfortunately, the following theorem shows that such a condition is undecidable and, thus, a query engine would not be able to check it in order to ensure that a query can be evaluated.

Theorem 2. *The problem of verifying, given a SPARQL query P, whether P is service-bound is undecidable.*

As for the case of the notion of boundedness, the fact that the notion of service-boundedness is undecidable prevents one from using it as a restriction over the variables used in SERVICE calls. To overcome this limitation, we replace the restriction that the variables used in SERVICE calls are bound by the decidable restriction that they are strongly bound. In this way, we obtain a syntactic condition over SPARQL patterns that ensures that they are service-bound, and which can be efficiently verified.

Definition 4 (Service-safeness). *A SPARQL query P is service-safe if for every node u of $\mathcal{T}(P)$ with label* $(\text{SERVICE } ?X\ P_1)$*, it holds that: (1) there exists a node v of $\mathcal{T}(P)$ with label P_2 such that v is an ancestor of u in $\mathcal{T}(P)$ and $?X \in \text{SB}(P_2)$; (2) P_1 is service-safe.*

Proposition 2. *If a SPARQL query P is service-safe, then P is service-bound.*

The notion of service-safeness is used in our system to verify that a SPARQL pattern can be evaluated in practice. We conclude this section by pointing out that it can be efficiently verified whether a SPARQL query P is service-safe, by using a bottom-up approach over the parse tree $\mathcal{T}(P)$ of P.

4 Optimizing the Evaluation of the OPTIONAL Operator in SPARQL Federated Queries

If a SPARQL query Q including the SERVICE operator has to be evaluated in a SPARQL endpoint A, then some of the sub-queries of Q may have to be evaluated in some external SPARQL endpoints. Thus, the problem of optimizing the evaluation of Q in A, and, in particular, the problem of reordering Q in A to optimize this evaluation, becomes particularly relevant in this scenario, as in some cases one cannot rely on the optimizers of the external SPARQL endpoints. Motivating by this, we present in this section some optimization techniques that extend the techniques presented in [11] to the case of SPARQL queries using the SERVICE operator, and which can be applied to a considerable number of SPARQL federated queries.

4.1 Optimization via Well-Designed Patterns

In [11,17], the authors study the complexity of evaluating the fragment of SPARQL consisting of the operators AND, UNION, OPT and FILTER. One of the conclusions of these papers is that the main source of complexity in SPARQL comes from the use of the OPT operator. In light of these results, it was introduced in [11] a fragment of SPARQL that forbids a special form of interaction between variables appearing in optional parts, which rarely occurs in practice. The patterns in this fragment, which are called well-designed patterns [11], can be evaluated more efficiently and are suitable for reordering and optimization. In this section, we extend the definition of the notion of being well-designed to the case of SPARQL patterns using the SERVICE operator, and prove that the reordering rules proposed in [11], for optimizing the evaluation of well-designed patterns, also hold in this extension. The use of these rules allows to reduce the number of tuples being transferred and joined in federated queries, and hence our implementation benefits from this as shown in Section 5.

Let P be a graph pattern constructed by using the operators AND, OPT, FILTER and SERVICE, and assume that P satisfies the safety condition that for every sub-pattern $(P_1 \text{ FILTER } R)$ of P, it holds that $\text{var}(R) \subseteq \text{var}(P_1)$. Then, by following [11], we say that P is well-designed if for every sub-pattern $P' = (P_1 \text{ OPT } P_2)$ of P and for every variable $?X$ occurring in P: If $?X$ occurs both inside P_2 and outside P', then it also occurs in P_1. All the graph patterns given in the previous sections are well-designed. On the other hand, the following pattern P is not well-designed:

$$((?X, \text{nickname}, ?Y) \text{ AND } (\text{SERVICE } c\ ((?X, \text{email}, ?U) \text{ OPT } (?Y, \text{email}, ?V)))),$$

as for the sub-pattern $P' = (P_1 \text{ OPT } P_2)$ of P with $P_1 = (?X, \text{email}, ?U)$ and $P_2 = (?Y, \text{email}, ?V))$, we have that $?Y$ occurs in P_2 and outside P' in the triple

pattern $(?X, \text{nickname}, ?Y)$, but it does not occur in P_1. Given an RDF graph G, graph pattern P retrieves from G a list of people with their nicknames, and retrieves from the SPARQL endpoint located at the IRI c the email addresses of these people and, optionally, the email addresses associated to their nicknames. What is unnatural about this graph pattern is the fact that $(?Y, \text{email}, ?V)$ is giving optional information for $(?X, \text{nickname}, ?Y)$, but in P appears as giving optional information for $(?X, \text{name}, ?U)$. In fact, it could happen that some of the results retrieved by using the triple pattern $(?X, \text{nickname}, ?Y)$ are not included in the final answer of P, as the value of variable $?Y$ in these intermediate results could be incompatible with the values for this variable retrieved by using the triple pattern $(?Y, \text{email}, ?V)$.

In the following proposition, we show that well-designed patterns including the SERVICE operator are suitable for reordering and, thus, for optimization.

Proposition 3. *Let P be a well-designed pattern and P' a pattern obtained from P by using one of the following reordering rules:*

$$\begin{aligned}
((P_1 \text{ OPT } P_2) \text{ FILTER } R) &\longrightarrow ((P_1 \text{ FILTER } R) \text{ OPT } P_2),\\
(P_1 \text{ AND } (P_2 \text{ OPT } P_3)) &\longrightarrow ((P_1 \text{ AND } P_2) \text{ OPT } P_3),\\
((P_1 \text{ OPT } P_2) \text{ AND } P_3) &\longrightarrow ((P_1 \text{ AND } P_3) \text{ OPT } P_2).
\end{aligned}$$

Then P' is a well-designed pattern equivalent to P.

The proof of this proposition is a simple extension of the proof of Proposition 4.10 in [11]. In the following section, we show that the use of these rules can have a considerable impact in the cost of evaluating graph patterns.

5 Implementation of SPARQL-DQP and Well-Designed Patterns Optimization

In this section, we describe how we implemented and evaluated the optimization techniques presented in the previous section. In particular, we demonstrate that they effectively decrease the processing time of SPARQL 1.1 federated queries.

5.1 Implementation: SPARQL-DQP

We have implemented the rewriting rules described in Section 4.1 in SPARQL-DQP [5], together with a bottom up algorithm for checking the condition of being well-designed. SPARQL-DQP is a query evaluation system built on top of OGSA-DAI [3] and OGSA-DQP [10]. OGSA-DAI is a generic service-based data access, integration, transformation and delivery framework that allows executing data-centric workflows involving heterogeneous data resources. OGSA-DAI is integrated in Apache Tomcat and within the Globus Toolkit, and is used in OMII-UK, the UK e-Science platform. OGSA-DQP is the Distributed Query Processing extension of OGSA-DAI, which access distributed OGSA-DAI data resources and provides parallelization mechanisms. SPARQL-DQP [5] extends this framework with new SPARQL parsers, logical query plan builders, operators and optimizers for distributed query processing. The main reason for selecting this framework is that it provides built-in infrastructure to support DQP

and enables handling large datasets and tuple streams, which may result from the execution of queries in different query services and data sources. The low level technical details of our implementation can be found in [5].

5.2 Evaluation

In our evaluation, we compare the results and performance of our system with other similar systems that provide some support for SPARQL query federation. Currently, the engines supporting the official SPARQL 1.1 federation extension are: DARQ [14], Networked Graphs [15] and ARQ, which is available via an online web service (`http://www.sparql.org/`) as well as a library for Jena (`http://jena.sourceforge.net/`). Other system that supports distributed RDF querying is presented in [18]. We do not consider this system here as it uses the query language SeRQL instead of SPARQL.

The objective of our evaluation is to show first that we can handle SPARQL queries that comply with the federated extension, and second that the optimization techniques proposed in Section 4.1 actually reduce the time needed to process queries. We have checked for existing SPARQL benchmarks like the Berlin SPARQL Benchmark [4], SP^2Bench [16] and the benchmark proposed in [7]. Unfortunately for our purposes, the first two are not designed for a distributed environment, while the third one is based on a federated scenario but is not as comprehensive as the Berlin SPARQL Benchmark and SP^2Bench. Thus, we decided to base our evaluation on some queries from the life sciences domain, similar to those in [7] but using a base query and increasing its complexity like in [4]. These queries are real queries used by Bio2RDF experts.

Datasets description. The Bio2RDF datasets contains 2,3 billion triples organized around 40 datasets with sometimes overlapping information. The Bio2RDF datasets that we have used in our benchmark are: Entrez Gene (13 million triples, stored in the local endpoint sparql-pubmed), Pubmed (797 million triples), HHPID (244,021 triples) and MeSH (689,542 triples, stored in the local endpoint sparql-mesh). One of the practical problems that these benchmarks have is that public SPARQL endpoints normally restrict the amount of results that they provide. To overcome this limitation we installed Entrez Gene and MeSH in servers without these restrictions. We also divided them in files of 300,000 triples, creating endpoints for each one of them.

Queries used in the evaluation. We used 7 queries in our evaluation. The query structure follows the following path: using the Pubmed references obtained from the Entrez gene dataset, we access the Pubmed endpoint (queries Q1 and Q2). In these queries, we retrieve information about genes and their references in the Pubmed dataset. From Pubmed we access the information in the National Library of Medicine's controlled vocabulary thesaurus (queries Q3 and Q4), stored at MeSH endpoint, so we have more complete information about such genes. Finally, to increase the data retrieved by our queries we also access the HHPID endpoint (queries Q5, Q6 and Q7), which is the knowledge base for the HIV-1 protein. The queries, in increasing order of complexity, can be found at `http://www.oeg-upm.net/files/sparql-dqp/`. Next we show query Q4 to give the reader an idea of the type of queries that we are considering:

```
SELECT ?pubmed ?gene1 ?mesh ?descriptor ?meshReference
WHERE
{
  {SERVICE <http://127.0.0.1:2020/sparql-pubmed> {
     ?gene1 <http://bio2rdf.org/geneid_resource:pubmed_xref> ?pubmed .}}.
  {SERVICE <http://pubmed.bio2rdf.org/sparql> {
     ?pubmed <http://bio2rdf.org/pubmed_resource:meshref> ?mesh .
     ?mesh   <http://bio2rdf.org/pubmed_resource:descriptor> ?descriptor .}}.
   OPTIONAL { SERVICE <http://127.0.0.1:2021/sparql-mesh> {
     ?meshReference <http://www.w3.org/2002/07/owl#sameAs> ?descriptor .}}.
}
```

Results. Our evaluation was done in an Amazon EC2 instance. The instance has 2 cores and 7.5 GB of memory run by Ubuntu 10.04. The data used in this evaluation, together with the generated query plans and the original queries in Java formatting, can be found at `http://www.oeg-upm.net/files/sparql-dqp/`. The results of our evaluation are shown in the following table:

Query	Not optimized SPARQL-DQP	**Optimized SPARQL-DQP**	DARQ	NetworkedGraphs	ARQ
Q1	79,000ms.	**79,000ms.**	10+ min.	10+ min.	440,296ms.
Q2	64,179ms.	**64,179ms.**	10+ min.	10+ min.	10+ min.
Q3	134,324ms.	**134,324ms.**	10+ min.	10+ min.	10+ min.
Q4	152,559ms.	**136,482ms.**	10+ min.	10+ min.	10+ min.
Q5	146,575ms.	**146,575ms.**	10+ min.	10+ min.	10+ min.
Q6	322,792ms.	**79,178ms.**	10+ min.	10+ min.	10+ min.
Q7	350,554ms.	**83,153ms.**	10+ min.	10+ min.	10+ min.

A first clear advantage of our implementation is the ability to use asynchronous calls facilitated by the use of indirect access mode, what means that we do not get time out in any of the queries. This time out happens when accessing an online distributed query processing like in the case of ARQ (`www.sparql.org/query`). It is important to note that the ability to handle this type of queries is essential for many types of data-intensive applications, such as those based on Bio2RDF. Data transfer also plays a key role in query response times. For example, in some queries the local query engine received 150,000 results from Entrez gene, 10,000 results from Pubmed, 23,841 results from MeSH and 10,000 results from HHPID. The implemented optimizations are less noticeable when the amount of transferred data is fewer.

It is possible to observe three different sets of results from this preliminary evaluation. The first set (Q1–Q3 and Q5) are those that are not optimized because the reordering rules in Section 4.1 are not applicable. The second query group (Q4) represents the class of queries that can be optimized using our approach, but where the difference is not too relevant, because the less amount of transferred data. The last group of queries (Q6–Q7) shows a clear optimization when using the well-designed patterns rewriting rules. For example, in query 6 the amount of transferred data varies from a join of $150,000 \times 10,000$ tuples to a join of $10,000 \times 23,841$ tuples (using Entrez, Pubmed and MeSH endpoints), which highly reduces the global processing time of the query. Regarding the comparison with other systems, they do not properly handle these amounts of data. We represent as 10+ min. those queries that need more than 10 minutes to be answered.

In summary, we have shown that our implementation provides better results than other similar systems. Besides, we have also shown that our implementation, which benefits from an indirect access mode, can be more appropriate to deal with large datasets.

Acknowledgments. We thank the anonymous referees, the OGSA-DAI team (specially Ally Hume), Marc-Alexandre Nolin, Jorge Pérez and Axel Polleres for their help with this work. This research was supported by ADMIRE project FP7 ICT-215024 and FONDECYT grant 1090565.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley, Reading (1995)
2. Angles, R., Gutierrez, C.: The Expressive Power of SPARQL. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 114–129. Springer, Heidelberg (2008)
3. Antonioletti, M., et al.: OGSA-DAI 3.0 - The Whats and the Whys. UK e-Science All Hands Meeting, pp. 158–165 (2007)
4. Bizer, C., Schultz, A.: The Berlin SPARQL Benchmark. Int. J. Semantic Web Inf. Syst. 5(2), 1–24 (2009)
5. Buil, C., Corcho, O.: Federating Queries to RDF repositories. Technical Report (2010), `http://oa.upm.es/3302/`
6. Durst, M,. Suignard, M.: Rfc 3987, Internationalized Resource Identifiers (IRIs), `http://www.ietf.org/rfc/rfc3987.txt`
7. Haase, P., Mathäß, T., Ziller, M.: An evaluation of approaches to federated query processing over linked data. In: I-SEMANTICS (2010)
8. Harris, S., Seaborne, A.: SPARQL 1.1 Query. W3C Working Draft (June 1, 2010), `http://www.w3.org/TR/sparql11-query/`
9. Klyne, G., Carroll, J.J., McBride, B.: Resource description framework (RDF): Concepts and abstract syntax. W3C Recommendation (February 10, 2004), `http://www.w3.org/TR/rdf-concepts/`
10. Lynden, S., et al.: The design and implementation of OGSA-DQP: A service-based distributed query processor. Future Generation Computer Systems 25(3), 224–236 (2009)
11. Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of SPARQL. TODS 34(3) (2009)
12. Prud'hommeaux, E.: SPARQL 1.1 Federation Extensions. W3C Working Draft (June 1, 2010), `http://www.w3.org/TR/sparql11-federated-query/`
13. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF. W3C Recommendation (January 15, 2008), `http://www.w3.org/TR/rdf-sparql-query/`
14. Quilitz, B., Leser, U.: Querying distributed RDF data sources with SPARQL. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 524–538. Springer, Heidelberg (2008)
15. Schenk, S., Staab, S.: Networked graphs: a declarative mechanism for SPARQL rules, SPARQL views and RDF data integration on the Web. In: WWW, pp. 585–594 (2008)
16. Schmidt, M., Hornung, T., Lausen, G., Pinkel, C.: SP2Bench: A SPARQL Performance Benchmark. In: ICDE, pp. 222–233 (2009)
17. Schmidt, M., Meier, M., Lausen, G.: Foundations of SPARQL query optimization. In: ICDT, pp. 4–33 (2010)
18. Stuckenschmidt, H., Vdovjak, R., Geert-Jan, H., Broekstra, J.: Index structures and algorithms for querying distributed RDF repositories. In: WWW, pp. 631–639 (2004)