Strategies for executing federated queries in SPARQL1.1

Carlos Buil-Aranda[1⋆], Axel Polleres[2⋆⋆], and Jürgen Umbrich[2]

[1] Department of Computer Science, Pontificia Universidad Católica, Chile
cbuil@ing.puc.cl
[2] Vienna University of Economy and Business (WU)
{*first.last*}@wu.ac.at

Abstract A common way for exposing RDF data on the Web is by means of SPARQL endpoints which allow end users and applications to query just the RDF data they want. However, servers hosting SPARQL endpoints often restrict access to the data by limiting the amount of results returned per query or the amount of queries per time that a client may issue. As this may affect query completeness when using SPARQL1.1's federated query extension, we analysed different strategies to implement federated queries with the goal to circumvent endpoint limits. We show that some seemingly intuitive methods for decomposing federated queries provide unsound results in the general case, and provide fixes or discuss under which restrictions these recipes are still applicable. Finally, we evaluate the proposed strategies for checking their feasibility in practice.

1 Introduction

The Linked Open Data initiative promotes the publication and linkage of RDF data on the Web. Under this initiative many organisations (either public or private) expose billions of statements using the RDF data model and also provide links to other RDF datasets. A common way for accessing such RDF datasets is by means of SPARQL endpoints. These endpoints are Web services that implement the SPARQL protocol and then allow end users and applications to query just the RDF data they want. However, servers hosting SPARQL endpoints often restrict the access to the data by limiting the server resources available per received query and client. These physical resource limitations most commonly include restrictions of the size of result set returned to the end users (e.g., a 10.000 result limit for each query) or simply generating errors such as time outs on queries that spend too many resources. Such imposed limitations are necessary due to too many resource limits [**?**] when serving queries concurrently to many clients.

However, in practice, particularly in the context of using SPARQL1.1's Federated Query Extension [**?**], these limitations prevent users from obtaining complete answers to their SPARQL queries. The result set size limitation is particularly relevant when a user wants to federate SPARQL queries over a number of SPARQL endpoints.

⋆ Supported by the Millennium Nucleus Center for Semantic Web Research under Grant NC120004 and CONICYT/FONDECYT project 3130617.
⋆⋆ Supported by the Vienna Science and Technology Fund (WWTF) project ICT12-015.

Using different combinations of SPARQL patterns is possible to overcome the servers' result size limits and obtain complete result sets for SERVICE queries. A common pattern that can be used for that purpose is using the VALUES operator from the new SPARQL 1.1 Recommendation. This operator allows to "ship" the results from a local query to be joined with a remote pattern along with the service query. However this operator is still not widely implemented in currently deployed endpoints [**?**] and thus alternative options might have to be considered. After some preliminaries (§2) this paper presents a study of several alternative strategies (§3), allowing users to obtain sound and complete answers to SPARQL queries; for instance, we show that using naive nested loops or combinations of the UNION and FILTER operators for constraining remote queries may return unsound results; we further discuss how to fix these naive approaches to obtain correct results. Finally in §4+5 we evaluate different settings, depending on the data, the local SPARQL engine and the engine running at the involved remote endpoints in a federated query, before we conclude in § 6.

2 Preliminaries

We first describe the basics of the SPARQL syntax we will use thorough the paper followed by the semantics of the most relevant SPARQL operators used.[1]

Syntax. The official syntax of SPARQL1.1 [**?**] considers operators `OPTIONAL`, `UNION`, `FILTER`, `SELECT`, concatenation via a point symbol (`.`), `{ }` to group patterns, as well as keywords (new in SPARQL 1.1) `SERVICE` to delegate parts of a query to a remote endpoint, and `VALUES` to define sets of variable bindings.

We follow [**?**,**?**] for defining the SPARQL syntax operators including the `VALUES` and `SERVICE` operators. We use letter B,I,L,V for denoting the (infinite) sets of blank nodes, IRIs, RDF literals, and variables as usual.[2]

(1) A triple $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ 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]
(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, then $\text{SELECT } \boldsymbol{W} \ P \ \text{ORDER BY } \boldsymbol{V} \ \text{LIMIT } l \ \text{OFFSET } o$ is a graph pattern (subquery), where ORDER BY, LIMIT, and OFFSET clauses (aka *solution modifiers*) are optional, $\boldsymbol{W}, \boldsymbol{V}$ are sets of variables, and $l, o \in \mathbb{N}$.[4]
(5) If P is a graph pattern and $a \in (I \cup V)$, then $(\text{SERVICE } a \ P)$ is a graph pattern.

[1] Note that we assume a set-based semantics as in [**?**] here, i.e. implicitly we assume DISTINCT queries. We also used DISTINCT queries in our experiments.
[2] In SPARQL patterns, blank nodes and variables can be used interchangeably, which is why we ignore blank nodes in SPARQL patterns.
[3] AND is syntactically written as either a sequence of '{ }'-delimited group graph patterns, or a sequence of '.'-separated triple patterns.
[4] We simplify here, as general ORDER BY clauses in SPARQL allow arbitratry expressions.

(6) VALUES $\boldsymbol{W}\,\boldsymbol{A}$ is a graph pattern where $\boldsymbol{W} = [?X_1, \ldots, ?X_n]$ is a sequence of pairwise distinct variables, and

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1}, & \ldots, & a_{1,n} \\ a_{2,1}, & \ldots, & a_{2,n} \\ & \vdots & \\ a_{m,1}, & \ldots, & a_{m,n} \end{bmatrix}$$

is a matrix of values where $a_{i,j} \in (I \cup L \cup \{\text{UNBOUND}\})$.

For the exposition of this paper, we leave out further more complex graph patterns such as `GRAPH` graph patterns, or new SPARQL 1.1 [**?**] like aggregates and property paths. We will use the notion of FILTER expressions as defined in [**?**]. We also use unary predicates like *bound*, *isBlank*, and the binary equality predicate '$=$', which herein we consider as synonym for *sameTerm*$(\cdot,\cdot)$ from [**?**].[5]

Let P be a graph pattern; in what follows, we use $var(P)$ to denote the set of variables occurring in P. In particular, if t is a triple pattern, then $var(t)$ denotes the set of variables occurring in the components of t. Similarly, for a built-in condition R, we use $var(R)$ to denote the set of variables occurring in R.

Semantics. As in [**?**], we consider a set-based semantics (which can always be achieved in SPARQL using the keyword `DISTINCT`), since conjunctive query containment is already undecidable for bag-semantics [**?**].

We use terminology defined in [**?**] for compatibility between solution mappings, written $\mu_1 \sim \mu_2$. Let Ω_1 and Ω_2 be sets of mappings; the join , union, difference, and left outer-join operations for Ω_1 and Ω_2 are defined as follows:

$$\begin{aligned} \Omega_1 \bowtie \Omega_2 &= \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \\ &\qquad \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\}, \\ \Omega_1 \cup \Omega_2 &= \{\mu \mid \mu \in \Omega_1 \text{ or } \mu \in \Omega_2\}, \\ \Omega_1 \setminus \Omega_2 &= \{\mu \in \Omega_1 \mid \forall \mu' \in \Omega_2 : \mu \not\sim \mu'\}, \\ \Omega_1 \bowtie\!\!\!\!\sqsubset \Omega_2 &= (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 \setminus \Omega_2). \end{aligned}$$

As usual, we use $dom(\mu)$ for denoting the variables bound within – i.e. the domain of – a SPARQL solution mapping μ. The evaluation semantics of SPARQL patterns with respect to an RDF graph G is defined in Fig. 1.

The evaluation of a FILTER expression R wrt. solution mapping μ relies on a three-valued logic $(\top, \bot, \varepsilon)$, cf. [**?**, §17.2], where
$\mu(R) = \top$, if:
- R is *bound*$(?X)$ and $?X \in dom(\mu)$;
- R is *isBlank*$(?X)$, $?X \in dom(\mu)$ and $\mu(?X) \in B$;
- R is *isIRI*$(?X)$, $?X \in dom(\mu)$ and $\mu(?X) \in I$;
- R is *isLiteral*$(?X)$, $?X \in dom(\mu)$ and $\mu(?X) \in L$;

[5] Note that '=' would otherwise also involve certain datatype inferences, e.g. `"1.0"^^xsd:decimal="1"^^xsd:integer` in SPARQL, whereas $sameTerm$(`"1.0"^^xsd:decimal`, `"1"^^xsd:integer`) = $false$.

- R is $?X = c$, $?X \in dom(\mu)$ and $sameTerm(\mu(?X), c)$;
- R is $?X = ?Y$, $?X \in dom(\mu)$, $?Y \in dom(\mu)$ and $sameTerm(\mu(?X), \mu(?Y))$;
- R is $\neg R_1$ with $\mu(R_1) = \bot$;
- R is $(R_1 \vee R_2)$, and $\mu(R_1) = \top \vee \mu(R_2) = \top$;
- R is $(R_1 \wedge R_2)$, and $\mu(R_1) = \top \wedge \mu(R_2) = \top$

$\mu(R) = \varepsilon$, if:
- R is *isBlank*$(?X)$,$R =$ *isIRI*$(?X)$, or $R =$ *isLiteral*$(?X)$ and $?X \notin dom(\mu)$;
- R is $?X = c$ or $?X = ?Y$ with $?X \notin dom(\mu)$ or, in the latter case $?Y \notin dom(\mu)$;
- R is $\neg R_1$ with $\mu(R_1) = \varepsilon$;
- R is $(R_1 \vee R_2)$, and $\mu(R_1) = \varepsilon \wedge \mu(R_2) = \varepsilon$;
- R is $(R_1 \wedge R_2)$, and $\mu(R_1) = \varepsilon \vee \mu(R_2) = \varepsilon$

$\mu(R) = \bot$, otherwise.

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

$$[\![P]\!]_G = \begin{cases} [\![P_1]\!]_{ep(c)} & \text{if } c \in dom(ep) \\ \{\mu_\emptyset\} & \text{if } c \in I \setminus dom(ep) \\ \left\{\mu \cup [c \to s] \mid \exists s \in dom(ep), \mu \in [\![P_1]\!]_{ep(s))} \wedge [c \to s] \sim \mu\right\} & \text{if } c \in V \end{cases}$$

(7) If $P =$ VALUES $\boldsymbol{W}$ $\boldsymbol{A}$ then

$$[\![P]\!]_G = \{\mu_j | 1 \leq j \leq m, dom(\mu_j) = \{?X_i \in \boldsymbol{W} \mid a_{i,j} \neq \text{UNBOUND}\}, \mu_j(?X_i) = a_{i,j}\}$$

(8) If P is SELECT $\boldsymbol{W}$ P_1 ORDER BY $\boldsymbol{V}$ LIMIT l OFFSET o, then
$[\![P]\!]_G = \Pi_{\boldsymbol{W}}(\text{lmt}(\text{order}([\![P_1]\!]_G, \boldsymbol{V}), l, o))$, where $\Pi_{\boldsymbol{W}}$ is projection as in Rel. Alg., and
order$(\Omega, \boldsymbol{V}) = L$ the sequence of $\mu \in \Omega$ obtained from ordering $\mu(\boldsymbol{V})$ as per [**?**, §15.1].
lmt(L, l, o) $= L'$ obtained from $L$by removing all $L[i]$ with $i \leq o$ or $o + l < i$

Figure 1. Definition of $[\![P]\!]_G$ for a graph pattern P using FILTER and VALUES operators.

Note in Fig. 1, that the semantics of SELECT queries is actually non-deterministic, in the sense that a compliant implementation can give different results, as illustrated by the following example.

Example 1. Let us assume the default graph $G_1 = \{(a, b, 1), (a, b, 2)\}$. and the pattern P_1 = SELECT $?X(a, b, ?X)$ LIMIT 1 then obviously both $\{[?X \to 1]\}$ and $\{[?X \to 2]\}$ would be allowed results, since there is no order prescribed among the results of $[\![(a, b, ?X)]\!]_G$. Likewise, P_2 = SELECT $?X(a, b, ?X)$ LIMIT 1 OFFSET 1 could have the same two possible results, indeed a compliant SPARQL engine could – according to the specification return the same result for both P_1 and P_2.

Note that even an ORDER BY does not necessarily remedy such ambiguities in all cases, since, according to the ordering rules in [**?**, §15.1], not all RDF terms are or-

dered; particularly, no order is specified for instance for blank nodes. To illustrate this, assume the default graph $G_1 = \{(a, b, \_:b1), (a, b, \_:b2)\}$ such that $\_:b1, \_:b2 \in B$. Then, similarly, both $\{[?X \rightarrow \_:b1]\}$ and $\{[?X \rightarrow \_:b2]\}$ would be allowed results for either P_3 = SELECT $?X(a, b, ?X)$ ORDER BY $?X$ LIMIT 1 or P_4 = SELECT $?X(a, b, ?X)$ ORDER BY $?X$ LIMIT 1 OFFSET 1.

3 Evaluation Strategies for SPARQL SERVICE patterns

In this section we outline several potential evaluation strategies for queries to a remote SPARQL endpoint using SERVICE patterns of the form $P = P_1$ AND (SERVICE c P_2).

Symmetrical Hash Join (SYMHASH) A classical alternative to implement SERVICE patterns is to use a symmetrical hash join, a type of hash join commonly used in data streams. We evaluate both query parts P_1 and P_2 separately and locally join the results ($[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2]\!]_{G_c}$) using two hash tables (one for each sub query). Depending on the interim result sizes of P_1 and P_2, this join algorithm can be a very efficient solution due to its possible parallelisation. However, the symmetrical hash join is expensive if the interim result sets are much larger then the join result size, plus, particularly, if the remote endpoint c imposes a result size limit of n being smaller than the size of $[\![P_2]\!]_{G_c}$, then join results may be lost.

Pagination with ORDERBY and LIMIT (SYMHASHP) An alternative to circumvent the problems of a local symmetric hash join would be to use "pagination". Here, by "pagination" we mean issuing queries of the form P_1 AND (SERVICE c (SELECT $*$ $\{P_2\}$ ORDER BY $inScopeVars(P_2)$) LIMIT n OFFSET o) where $o = n * i$ with increasing $i \in \mathbb{N}$ until less than n results are returned from service c. Each batch of remote results can again be joined with the local results of P_1 using a hash join.

However, Ex. 1 above already shows that there is no simple work-around for circumventing result size limits when querying remote SPARQL endpoints in terms of "pagination": that is, let us assume $P = P_1$ AND (SERVICE c P_2) be a SPARQL pattern, where service c limits result size to delivering at most n results. Then, as a consequence of the Ex. 1, one cannot simply use "pagination" of the results of the remote endpoint.

Getting back to the ordering rules in [**?**, §15.1], we note that there are still cases where we can safely use pagination, namely, (1) if the total result size of the remote endpoint query is below the remote result size limit n, then there are no problems. This can be easily checked by issuing a query `ASK { SERVICE` c `{ SELECT * {` P_2 `} LIMIT` n `}}` to the remote endpoint. In case this delivers less than n results, we do not need pagination anyway. Otherwise, we have to check whether we can safely order the results to guarantee that we can get all available results by "pagination". To this end, we need to be sure that all bindings to "output variables" (i.e., variables that are "in-scope" [6]) in the service pattern P_2 are either unbound or can be ordered by the

[6] these are just the variables that would be included in a `SELECT *`, cf. http://www.w3.org/TR/sparql11-query/#ins

"<" operator according to [**?**, §15.1]: *the "<" operator (see [?, §17.3.1] Operator Extensibility) defines the relative order of pairs of numerics, simple literals, xsd:strings, xsd:booleans and xsd:dateTimes. Pairs of IRIs are ordered by comparing them as simple literals.* In fact, we can check this "orderability" condition — which would allow us to impose a total order on the in-scope variables of the service pattern by using ORDER BY — by another ASK query:[7]

```
ASK { SERVICE c { P2  FILTER (  ! (
          ⋀ for each variable ?v ∈ inScopeVars(P2)
           ( !bound(?v) ⋁
             isNumeric(?v) ⋁
             datatype(?v) = xsd:boolean ⋁
             datatype(?v) = xsd:dateTime ⋁
             datatype(?v) = xsd:string ) ) ) }  }
```

If this query returns false, the remote results can be indeed ordered, and we can proceed with "pagination". However, while this approach is feasible, it is not applicable in general, plus executing several consecutive ORDER BY queries might actually be quite expensive on the remote side and trigger resource limits nonetheless. We thus look for other, more feasible alternatives, that allow to "push" the results from the local side into the remote query, which we will describe in the following.

Nested Loop Join (NESTED) A straightforward method to alternatively evaluate a query of the form $P = P_1$ SERVICE c P_2 – with the particular advantage to keep the intermediate size of results shipped from the remote endpoint low, and thus potentially circumventing result size limits – is to use a nested loop join. Here, we first evaluate P_1, iterate over the solution bindings $\mu \in [\![P_1]\!]_G$, execute $\mu(P_2)$ remotely and extend μ with the additionally obtained variable bindings.

One potential problem of this approach is that we issue one request for each binding of P_1; this can lead to denial of service attacks if there is no appropriate wait time between two requests for large interim result sets.

Even worse, when done naively, this method fails in relatively simple queries as shown in the following example.

Example 2. Assume P_1 = (?X, c, d) and P_2 = ((?Y, ?Z, ?T) UNION (?X, ?Y, b)) FILTER (?X = ?Y). With the local default graph $G_1 = \{$(a, c, d)$\}$ and the remote service's default graph $G_2 = \{$(a, a, b), (e, c, d)$\}$, we obtain: $[\![P_1]\!]_{G_1} = \{\mu\}$, with $\mu = \{[?X \to a]$, whereas $[\![P_2]\!]_{G_2} = \{[?X \to a, ?Y \to a]\}$. However, if we proceed as suggested above, then $\mu(P_2) = ((?Y, ?Z, ?T)$ UNION $(a, ?Y, b))$ FILTER $(a = ?Y)$ which yields an additional solution $[?Y \to a, ?Z \to a, ?T \to b]$ that was not admissible in the original P_2 but is also compatible with $\{[?X \to a]\}$.

Another problem is with blank nodes. Assume $P_1 = P_2 =$ (?X, c, d) with $G_1 = \{(\_ : \mathtt{b}, c, d)\}$ and $G_2 = \{(a, c, d)\}$. Here, as replacement would yield $\mu(P_2) = (\_ : \mathtt{b}, c, d)$ and since SPARQL engines treat blank nodes in patterns as variables, again a non-admissible solution would arise.

[7] Note that the `datatype(·)` function also returns `xsd:string` on simple literals.

Thus, applying a nested loop join with naive replacement in a federation scenario, would potentially obtain inconsistent results.

SPARQL 1.1 VALUES **operator** (VALUES). As a further alternative, the new VALUES operator in SPARQL 1.1 can be used for "shipping" the local result bindings from $[\![P_1]\!]_G$ along with the remote query that is sent using the SERVICE operator as follows: let again $P = P_1$ AND (SERVICE c P_2). If we pre-evaluate the solution bindings for P_1, written $[\![P_1]\!]_G$, the SERVICE operator could then be equivalently evaluated by replacing pattern P_2 with

$$P_2^{\text{VALUES}_{P_1}} = P_2 \text{ VALUES } \boldsymbol{W}\,\boldsymbol{A}$$

where $\boldsymbol{W} = [var(P_1) \cap var(P_2)]$ and $\boldsymbol{A} = [\![P_1]\!]_G$ to endpoint c, i.e. if G_c is the default graph of service c

$$[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2^{\text{VALUES}_{P_1}}]\!]_{G_c}$$

However, a potential problem with this approach is that the VALUES operator is not yet widely deployed in existing endpoints [**?**] and other operators have to be used in order to simulate the desired behaviour.

SPARQL FILTER (FILTER) As an alternative to the usage of VALUES one may consider using FILTERs to "inject" the results of P_1 into P_2, namely, by replacing P_2 with

$$P_2^{\text{FILTER}_{P_1}} = \{P_2 \text{ FILTER} \bigvee_{\mu \in [\![P_1]\!]_G} (\bigwedge_{v \in dom(\mu) \cap vars(P_2)} v = \mu(v))\}$$

The FILTER expression here makes sure that only those solution bindings of P_2 survive that join with some solution binding of of P_2.

SPARQL UNION (UNION). Yet another alternative is the use of the UNION operator in combination with FILTERs. Here, the idea is to use the results of P_1 to create a large UNION query, where in each branch of the UNION the bindings of one solution for P_1 are 'injected" by means of a FILTER, instead of one large FILTER. I.e., for $[\![P_1]\!]_G = \{\mu_1, \ldots, \mu_n\}$ we replace P_2 by

$$P_2^{\text{UNION}_{P_1}} = \{(\mu_1^{\text{FILTER}}(P_2)) \text{ UNION} \ldots \text{UNION}(\mu_n^{\text{FILTER}}(P_2))\}$$

Here, $\mu^{\text{FILTER}}(P_2) = P_2 \text{ FILTER}(\bigwedge_{v \in dom(\mu) \cap vars(P_2)} v = \mu(v))$.

However, there are problems with unbound variables in both $P_2^{\text{FILTER}_{P_1}}$ and $P_2^{\text{UNION}_{P_1}}$, as shown in the following example.

Example 3. Assume P_1 = (?X, b, c), $c = I$ and P_2 = ((?Y,d,e) UNION (?X,d,e)). With the local default graph $G_1 = \{(a, b, c)\}$ and the remote service' default graph $G_2 = \{(a, d, e)\}$, we obtain: $[\![P_1]\!]_{G_1} = \{[?X \to a]\}$ and $[\![P_2]\!]_{G_2} = \{[?X \to a], [?Y \to a]\}$; here, the second solution for $[\![P_2]\!]_{G_2}$, i.e. $\mu_2 = [?Y \to a]$ is compatible with the single solution for $[\![P_1]\!]_{G_1}$, i.e., $\mu_1 = [?Y \to a]$ yielding overall $\mu = [?X \to a, ?Y \to a]$. However, $P_2^{\text{FILTER}_{P_1}} = P_2^{\text{UNION}_{P_1}} =$ { (?Y,d,e) UNION (?X,d,e) } FILTER(?X = a)} which would not yield μ as a solution.

So, while the use of nested loops may yield incorrect additional results, the version using FILTER and UNIONs seem to miss some results,. In the next subsection we discuss refined versions of these three alternatives, that solve these issues.

3.1 Two Equivalence theorems for SPARQL Federated Queries

As we have seen, some queries may return unexpected result mappings when substituting a variable for a specific value in nested loops. Thus, we aim at finding out a restricted class of SPARQL remote queries for which we obtain correct results. It turns out that one class of queries which avoid the above-mentioned problems is the class of queries where all join variables are *strongly bound*[**?**]: strong boundedness ensures, by the following syntactic restrictions, that a variable $?X$ in a SPARQL pattern P will be bound to a value in each solution binding, independent of the underlying data.

Definition 1 (Strong boundedness (from [?]). *Let P be a SPARQL pattern; 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 $\mathrm{SB}(P) = var(t)$;*
- *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 $P = (P_1 \text{ FILTER } R)$, 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 $P = (P_1 \text{ VALUES } S\ \{A_1, \ldots, A_n\})$, 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 $?X \in dom(\mu_{S,A_i})\}$.*
- *if $P = (\text{SELECT } W\ P_1)$, then $\mathrm{SB}(P) = (W \cap \mathrm{SB}(P_1))$.*

Indeed, one source of problems in Ex. 2 was the query results containing the empty result mapping. That empty result mapping combined with other operators generates result sets different to the original one (aside of being unexpected) since the empty result mapping is not *null rejecting*.

The following Lemma essentially states that replacing strongly bound variables with IRIs or literals in a pattern will not yield additional results for P.

Lemma 1. *Given a SPARQL pattern P with $v \in SB(P)$, let $\mu_e = [v \rightarrow e]$ for an $e \in I \cup L$, then $[\![\mu_e(P)]\!]_G \bowtie \mu_e = \{\mu \in [\![P]\!]_G | v \in dom(\mu) \wedge \mu(v) = e\}$.*

Indeed, we can remedy the aforementioned issue of blank nodes if we replace $\mu(P_2)$ with $\mu^B(P_2) = \{\mu(P_2)\,\mathrm{FILTER}(\neg(\bigvee_{v \in dom(\mu) \cap vars(P_2)} isBlank(\mu(v))))\}$ within the nested loop, i.e. the problematic solutions containing blank nodes are filtered out.

Indeed, we confirm that nested loop replacement with this modification works for remote patterns with only strongly bound variables; plus, it turns out that remote queries with only strongly bound join variables also are evaluated correctly using the $FILTER$ and $UNION$ approaches:

Theorem 1. *Let $P = P_1$ AND (SERVICE $c\ P_2$) such that $(vars(P_2) \cap vars(P_1)) \subseteq \mathrm{SB}(P_2)$, i.e. all variables that participate in a join are strongly bound in the pattern appearing on the service side, and let G_c be the default graph of service c and let $P_2^{\mathrm{UNION}_{P_1}}$ and $P_2^{\mathrm{FILTER}_{P_1}}$ be as defined above, then*

(i) $[\![P]\!]_G = \bigcup_{\mu \in [\![P_1]\!]_G} (\mu \bowtie [\![\mu^B(P_2)]\!]_{G_c})$

(ii) $[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2^{\mathrm{UNION}_{P_1}}]\!]_{G_c} = [\![P_1]\!]_G \bowtie [\![P_2^{\mathrm{FILTER}_{P_1}}]\!]_{G_c}$

We note that if the local graph G does not contain any blank nodes[8], then Theorem 1(i) would also hold using the original replacement $\mu(P_2)$. Moreover, it turns out that we can generalise the result in Theorem 1(ii) to also work in the general case with potentially unbound variables in the service pattern. To this end, in both $P_2^{\mathrm{FILTER}_{P_1}}$ and $P_2^{\mathrm{UNION}_{P_1}}$ expressions we replace $v = \mu(v)$ by $v = \mu(v) \vee \neg bound(v)$, obtaining $P_2^{\mathrm{FILTER}'_{P_1}}$ and $P_2^{\mathrm{UNION}'_{P_1}}$, resp.

The trick to only filter for variables bound within P_2 fixes the problem from Ex. 3 above, as stated in the following theorem.

Theorem 2. *Let* $P = P_1$ AND (SERVICE c P_2) *and* G_c *be the default graph of service* c *then*

$$[\![P]\!]_G = [\![P_1]\!]_G \bowtie [\![P_2^{\mathrm{FILTER}'_{P_1}}]\!]_{G_c} = [\![P_1]\!]_G \bowtie [\![P_2^{\mathrm{UNION}'_{P_1}}]\!]_{G_c}$$

4 Evaluation

In total, we have 6 alternative evaluation strategies for a $P = P_1$ SERVICE P_2 query, as listed in the overview given in Table 1. The goal of our evaluation is to study how systems implement the SERVICE keyword and how the alternative evaluation strategies behave for different queries.

Table 1. Overview of evaluation strategies as detailed in §3.

ID	Description
SERVICE	Baseline evaluation strategy
VALUES	Evaluation as described in §3
SYMHASH	A symmetrical hash join without pagination
SYMHASHP	A symmetrical hash join with pagination of the remote results
NESTED	A naive nested loop evaluation strategy
UNION	Evaluation as described in Theorem 1
FILTER	Evaluation as described in Theorem 2

All of our evaluation strategies are implemented in Java7 using the Jena ARQ library (version 2.9.4)[9]. The three systems under test are 1) Jena Fuseki 0.2.7 with on disk index (TDB), 2) Sesame workbench 2.7.11 and 3) Virtuoso Open Source Edition 7.10.

We did not pose any result limit to the systems and followed the official documentation for the installation.

[8] Existence of blank nodes in a dataset could be easily tested with a query such as `ASK {?S ?P ?O FILTER isBlank(?S) \/ isBlank(?O)) }`.

[9] The implementation and all queries are online at https://github.com/cbuil/sparql_strategies

4.1 Methodology

The methodology followed in our evaluation consists in two parts: first we evaluate the correctness of our strategies by using the data and queries presented in §3.1 and next we evaluate the strategies using real data from the Bio2RDF project[10].

SERVICE implementation test
First, we test the implementation of the selected SPARQL HTTP servers and how they deal with the problems addressed in this work. We created two small datasets that contain the RDF data in the examples before. Next we evaluate each of the strategies presented using these datasets to check the correctness of our approach and the engines serving SPARQL.

Compare alternative strategies
Next, we verify that all strategies can produce the right results if 1) the queries do not involve unbound join variables and 2) if the queries contain variables that may be unbound. We remove the result set size limit that the servers usually impose to the query execution. We also use this test to measure the performance differences of the strategies depending on the query characteristics and we also test how the symmetrical hash joins perform with and without pagination. We measured for each configuration of query and join implementation result size, and query times.

4.2 Data & Queries

SERVICE evaluation: we use the data and queries from Examples 2 and 3 to test the SERVICE implementation of the different stores. These two queries are used to check how unbound variables affect the execution of a federated SPARQL query. The first query $Q1$ contains the UNION pattern presented in Example 2 in P_1 (i.e. the pattern that queries the local dataset). The second query $Q2$ is the query presented in Example 3. That query contains a single graph pattern for querying the local dataset (P_1) and the previous UNION pattern in the remote SERVICE call (P_2).

In addition, we downloaded two datasets from the Bio2RDF domain: the Mouse Genome Database (MGI) and the Database of Human Gene Names (HGNC). The MGI dataset consists of 2,454,589 triples and the HGNC of 919,738 triples and we created 8 queries for these two datasets. Each query has two SERVICE patterns P_1 and P_2, the first one (P_1) querying the local endpoint hosting the MGI dataset and the second (P_2) querying the remote endpoint, hosting the HGNC data.

The first 4 queries (`B0`– `B3`) plus query `B7` do not contain blank nodes in the interim results or unbound join variables while queries `B4`– `B6` contain variables that may be unbound. These three queries will allow us to study the behaviour of the proposed strategies with unbound join variables. The results obtained for these last queries may be unsound, i.e. they differ in the amount of results returned to the users, according to the theoretical results presented in Section 3.

[10] http://bio2rdf.org

Table 2. Result size of P_1 (local) and P_2 (remote) for our example queries. $|P|$ is the result size of executing first P_1, next P_2 and finally do the join locally.

	Cardinality		#Triple patterns			
Query	$[\![P_1]\!]_G$	$[\![P_2]\!]_G$	$\|P_1\|$	$\|P_2\|$	$\|P\|$	Comment
B0	27	1	1	1	1	
B1	27	33562	1	1	1	
B2	17817	33562	2	1	17547	
B3	16753	2	2	3	2	
B4	250924	23	1	2	6	P_1 contains one non strongly bound join variable
B5	16753	8771	2	2	3274	P_2 contains one non strongly bound join variable
B6	268743	27132	3	7	23873	P_1 and P_2 with non strongly bound join variable
B7	35636	33134	3	4	17545	Two join variables

5 Results

In this section we summarise the results we obtained from the execution of the proposed strategies using the evaluation queries for each dataset configuration. We first present the results of the strategies over the data and queries of §4 and next we present the results for the Bio2RDF dataset queries.

5.1 SERVICE implementation test

At the time of writing, we initially tested the current version of Fuseki (version 1.0.1) but due to a bug in the evaluation of FILTER expression we had to settle for a previous version not containing that error. This behaviour could be not observed with Fuseki version 0.2.7 which returned the correct results.

Overall, our evaluation confirmed that the SPARQL engines do not properly deal with unbound join variables for their SERVICE implementation. Table 3 shows the number of returned results for our two queries and the three tested systems. We observe that for all queries the FILTER, UNION and symmetrical hash join strategies returns the correct number of results in Sesame and Virtuoso. Considering Fuseki, all evaluation strategies fail to return the correct results except for in the symmetrical hash join strategy. We contacted the lead developer of the Fuseki system to verified that the SERVICE evaluation strategy is similar to a nested loop approach as the results indicated. Interestingly, Fuseki does not differ from Sesame and Virtuoso in the execution of SERVICE queries since they all return the same amount of results. That means that all three systems seem to implement a Nested Loop Join algorithm for implementing the SERVICE operator.

5.2 Performance of alternative strategies

The next evaluation task had the goal to see how the different strategies behave with real world data and different queries.

Table 3. Returned amount of results for theorem queries and synthetical data for different SPARQL engines: the table shows result size differences in the query evaluation for the different strategies by the SPARQL servers.

	Q1 (Example2)			Q2 (Example3)		
	Fuseki	Sesame	Virtuoso	Fuseki	Sesame	Virtuoso
SERVICE	2	2	2	2	2	2
VALUES	2	1	1	2	2	2
FILTER	2	1	1	2	1	1
UNION	2	1	1	2	1	1
NESTED	2	2	2	2	2	2
SYMHASH	1	1	1	1	1	1

Implementation restrictions & solution In our initial tests we observed two technical exceptions with the used systems and libraries.

HTTP GET *vs* HTTP POST *queries:* At first, we used HTTP GET requests to send the queries to the local and remote endpoints. However, the servers threw an exception if the created query URL exceeds the maximum or allowed URL length. This happened for queries with thousands of interim results for P_1. Our solution is to use HTTP POST requests with the query in the request body.

Large FILTER *and* UNION *expressions:* The second exception happened due to internal stack overflows in the ARQ library while parsing the FILTER (or UNION expression, caused by large numbers of filter statements for P_2. We mark such exception with a "+". Our technical solution for these exception is to split the results for P_1 into batches which can be handled by the remote endpoint. If such an exception occurred in our test we also evaluate the strategy with batch processing.

Results The runtime in ms for the various strategies and queries are presented in Table 4 for Fuseki, Table 5 for Sesame and Table 6 for Virtuoso. We use the superscript "$-$" to indicate an incorrect number of results. Results marked with superscript "*" indicate a `500 Server Error` response error from Fuseki, `400 Bad Request` from Sesame[11] or `HttpException` from Virtuoso. These errors indicate that a problem occurred in the server while the processing of the query was ongoing. In such cases, we report the times taken from a run with a batch size of 750 results. In this evaluation we use the symmetrical hash join with pagination strategy (SYMHASHP) with a batch size of 750 instead of the SYMHASH strategy since the former is more suitable for queries with larger amounts of results. Those queries that have as result `to` mean that were automatically stopped from the client after 30 minutes. The bold results present the best runtime for each query across the strategies.

Overall the FILTER and VALUES evaluation strategies provide in general the fastest query times for Fuseki (cf. Table 4). We can also see that the internal SERVICE evaluation strategy has similar runtimes as our nested loop implementation. This is no surprise since Fuseki uses a nested loop style evaluation strategy. The UNION strategy

[11] Sesame's `400 Bad Request` errors indicate that the query contained to many patterns.

Table 4. Query times in ms (Fuseki 0.2.7). The best strategy for a Fuseki server is to either use a FILTER strategy injecting the values in the FILTER expression or use the VALUES operator. The SYMHASHP is the third best strategy.

	B0	B1	B2	B3	B4	B5	B6	B7
SERVICE	1436	642	31620	39119	62243*	858681^{-}	60276*	to
FILTER	439	360^{+}	**7235**$^{+}$	6022^{+}	13633^{+}	**3638**$^{+}$	**26577**$^{+}$	62947^{+}
UNION	730	678^{+}	10657^{+}	15033^{+}	22269*	7732*	60814*	63335^{+}
VALUES	**211**	**227**	8247	5223	16728*	899667^{-}	19289*	**11646**
NESTED	758	643	52160	55445	to^{-}	922805^{-}	to	to
SYMHASHP	403	16462	17563	**1937**	**9494**	13082	53592	28759

is used for large remote queries which we needed to break down into batches to be accepted by the remote endpoint. However, comparing the runtimes with the FILTER evaluation strategy, which also required use of batch processing, we measured approximately two times slower performance. It is important to notice that queries B4 to B6 returned some type of error when using the strategies SERVICE, UNION, VALUES and NESTED. These queries contain one or more non strongly bound join variables which that not only increase the complexity of the query evaluation but also return wrong results. We observed that for query B4 using the NESTED strategy and removing all timeouts the amount of results returned was 5,361,421 (it should return 6 results) and it took more than 12 hours to finish (query marked with the superscript $^{-}$). Query B5 also took longer using the SERVICE, NESTED and VALUES strategies, and returned 4,576,843 results, when it should return 3,274 results. The most reliable strategies were FILTER and SYMHASHP, which managed to finish all query executions.

Table 5. Query times in ms (Sesame). The best strategy for Sesame is to use either SERVICE or FILTER strategies.

	B0	B1	B2	B3	B4	B5	B6	B7
SERVICE	**77**	320	**1555**	**73**	618852^{-}	to	to	**1968**
FILTER	149	596	4340^{+}	4788^{+}	7815^{+}	**3230**$^{+}$	**15746**$^{+}$	7032^{+}
UNION	260	645	6415	10807	12502	5443*	43648	13673
VALUES	167	**153**	4289	2893	8095*	2276*	8482*	7042
NESTED	443	385	52051	62083	to^{-}	to	to	89487
SYMHASHP	101	14520	15037	1203	**4662**	5409	20915	21242

The results of our evaluation with the Sesame system shows a different picture compared to the Fuseki test (cf. Table 5). Overall, the internal SERVICE implementation provides the best performance for 4 out of the 8 queries. This is surprising since our previous experiment suggested that Sesame implements the SERVICE operator as a Nested Loop Join. However, the results in Table 5 show that the SERVICE strategy outperforms the NESTED strategy by order of magnitudes. The results indicate that

Sesame uses some internal optimisations (e.g., based on statistics) which results in the observed runtime difference. For the other 3 queries, the FILTER strategy showed the fastest query answering time for queries B5 and B6 while the SYMHASHP strategy was the fastest for query B4. It is important to note that the only evaluation strategies that managed to finish executing were the FILTER and SYMHASHP strategies, while we observed the same problems regarding the non strongly bound join variables: query B4 returned 5,361,421 results in the SERVICE and NESTED strategies (the NESTED strategy needed almost 12 hours to complete). Again, the most reliable strategies were FILTER and SYMHASHP.

Table 6. Query times in ms (Virtuoso 7.10). The best strategy for a Virtuoso Server is a SYMHASHP, specially for low selective patterns (queries B6 and B7).

	B0	B1	B2	B3	B4	B5	B6	B7
SERVICE	45*	41*	60*	35*	61*	56*	54*	75*
FILTER	159	159+	**1731**	31720+	26329+	31324+	68749+	37444+
UNION	267	237+	30904+	72495+	55321+	3737*	7134*	11990+
VALUES	137	**117**	6611	7561	260736−	781657	to	13291−
NESTED	559	500	88240	128399	2721065*	to	to	196280
SYMHASHP	**102**	1905	2733	**2205**	**5525**	**2306**	**8149**	**3407**

Again, the results from the evaluation using the Virtuoso Open Source server are very different from the evaluation for other stores. The best strategy when using Virtuoso is a SYMHASHP strategy, which is the fastest in almost all queries. Only in query B1 the strategy VALUES and in query B2 the strategy FILTER strategy were faster. As before, a similar situation happens when running the queries containing non strongly bound join variables. In this case the VALUES strategy for query B4 returned 23 results, which differs from the 6 results that the FILTER and SYMHASHP strategies return. Again, the SYMHASHP and the FILTER strategies are the only strategies that were able to finish all query executions.

6 Related work & Conclusions

Although there is a lot of theoretical work on distributed query processing and query planning, both in the database world [**?**] and in the context of the semantic Web [**?**,**?**,**?**], this is indeed one of the first works that considers *practical* limitations of existing SPARQL endpoints when executing already established federated query plans using SPARQL1.1's federated query extension. For instance, FedX [**?**] describes a similar evaluation strategy to our UNION strategy (called *bound join*), but does not consider the corner cases we discuss in Theorems 1+2 above. Likewise, whereas various works have addressed equivalences and optimisations for local SPARQL query patterns [**?**,**?**,**?**], few have considered SERVICE patterns.

In summary, in this paper we have, firstly, illustrated that querying remote SPARQL endpoints with SERVICE patterns is a non-trivial task due to the limitations that the

servers hosting these endpoints impose; the most common restriction is a result size limit that prevents users from obtaining complete results to their queries. Secondly, we have also shown some results in terms of defining equivalences for SPARQL queries involving SERVICE patterns that may help remedy these limits in practice. Thirdly, our evaluation should give some hints on which strategies are practically feasible in a particular setting, depending on the data, the local SPARQL engine and the engine running at the involved endpoints in a federated query. It is important to notice that only the FILTER and SYMHASHP strategies returned results for all queries in all systems. In addition, these strategies were sound, i.e. returned correct results for all queries since they do not "inject" any new value in the remote query (instead they either filter the unwanted results out or perform the join locally). The NESTED strategy (along with the SERVICE strategy) not only failed to return results in several queries but also returned incorrect results when non strongly bound join variables were present (confirming thus the theoretical results obtained). We believe that the investigation of the issues around executing federated SPARQL queries in practice deserves increased attention if we seriously intend to make the the Semantic Web vision work.

7 Acknowledgements

Thanks to Aidan for his last bulletproof reading and to Martín for his support in finding the missing variable bindings.

A Proof for Theorem 1(i)

Proof. We now show that there is a 1:1 correpsondence between those solution mappings of P_1 joining with (SERVICE c P_2) and the ones joining with $\mu^B(P_2)$.

The theorem trivially holds for $c \in I \setminus dom(ep)$, i.e. if c is not the IRI of a SPARQL endpoint, cf. [**?**]. Otherwise, let now $\mu \in [\![P]\!]_G$, then we know for each mapping[12] $\mu_i \subseteq \mu \in [\![P_1]\!]_G$ there is a mapping $\mu_j \subseteq \mu \in [\![P_2]\!]_{G_c}$ such that for each variable $v \in dom(\mu_i) \cap var(P_2)$ it holds that $\mu_i(v) = \mu_j(v)$ $inI \cup L$: since (a) blank nodes from the local graph queried by P_1 are always disjoint with blank nodes from the remote endpoint P_2 and (b) all join variables (i.e., $var(P_1) \cap var(P_2)$) are strongly bound in P_2, which means that indeed all $v \in dom(\mu_i) \cap var(P_2)$ are also in $dom(\mu_j)$.
It is now easy to see that for each such solution μ_j there is a solution $\mu'_j \in [\![\mu^B(P_2)]\!]_{G_c}$ with $dom(\mu'_j) = dom(\mu_j) \setminus dom(\mu_i)$ and that corresponds to μ_j on all variables in $dom(\mu'_j)$, which proves that $[\![P]\!]_G \subseteq [\![P_1]\!]_G \bowtie [\![\mu^B(P_2)]\!]_{G_c}$

On the other hand, there are no additional mappings in $[\![\mu^B(P_2)]\!]_G$ that do not correspond to a μ_j which follows from Lemma 1 and the construction of $\mu^B(P_2)$, which concludes the argument. □

We leave Theorem 1(ii) – which can be shown with similar arguments – without proof and directly skip to the proof of the more general Theorem 2.

B Proof of Theorem 2

Proof. Without making any assumptions about (strong) boundedness of variables, again, we want to show that there is a 1:1 correpsondence between the solution mappings of P_1 joining with (SERVICE c P_2) and the ones joining with (SERVICE c $P_2^{\mathrm{FILTER}'_{P_1}}$). Again, the theorem trivially holds for $c \in I \setminus dom(ep)$, i.e. if c is not the IRI of a SPARQL endpoint. Otherwise, let now $\mu \in [\![P]\!]_G$, then we know for each $\mu_i \subseteq \mu \in [\![P_1]\!]_G$ there is a compatible mapping $\mu_j \subseteq \mu \in [\![P_2]\!]_{G_c}$ such that for each join variable $v \in var(P_1) \cap var(P_2)$ it holds that either: **(i)** $v \notin dom(\mu_i)$, **(ii)** $v \notin dom(\mu_j)$, or **(iii)** $\mu_i(v) = \mu_j(v)$. We now treat these cases separately to show that $\mu_j \in [\![P_2^{\mathrm{FILTER}'_{P_1}}]\!]_{G_c}$ in each case:

(i): since the FILTER expression in $P_2^{\mathrm{FILTER}'_{P_1}}$ only considers variables in $dom(\mu_i)$ the FILTER leaves bindings for v in P_2 unaffected.

(ii) : the FILTER expression in $P_2^{\mathrm{FILTER}'_{P_1}}$ evaluates to true, due to $\neg bound(v)$

(iii) : the FILTER expression in $P_2^{\mathrm{FILTER}'_{P_1}}$ evaluates to true, due to $v = \mu(v)$

In total, we have shown that each $[\![P]\!]_G \subseteq [\![P_1]\!]_G \bowtie [\![P_2^{\mathrm{FILTER}_{P_1}}]\!]_{G_c}$.

For the other direction, it is easy to see that $[\![P_2^{\mathrm{FILTER}_{P_1}}]\!]_{G_c} \subseteq [\![P_2]\!]_{G_c}$, i.e. again, the rewritten query cannot deliver any additional results on the service side, which proved the opposite direction. □

Again, the proof for $P_2^{\mathrm{UNION}'_{P_1}}$ follows similar arguments.

[12] Slightly abusing notation, when we write here $\mu_i \subseteq \mu$, we mean that μ_i is a submapping of μ, i.e. that $dom(\mu_i) \subseteq dom(\mu)$ and that $\mu_i(v) = \mu(v)$ for all $v \in dom(\mu_i)$.