J Data Semant (2012) 1:147–185
DOI 10.1007/s13740-012-0008-7

ORIGINAL ARTICLE

Mapping between RDF and XML with XSPARQL

Stefan Bischof · Stefan Decker · Thomas Krennwallner · Nuno Lopes · Axel Polleres

Received: 2 December 2011 / Revised: 4 May 2012 / Accepted: 11 June 2012 / Published online: 4 September 2012
© Springer-Verlag 2012

Abstract One promise of Semantic Web applications is to seamlessly deal with heterogeneous data. The Extensible Markup Language (XML) has become widely adopted as an almost ubiquitous interchange format for data, along with transformation languages like XSLT and XQuery to translate data from one XML format into another. However, the more recent Resource Description Framework (RDF) has become another popular standard for data representation and exchange, supported by its own query language SPARQL, that enables extraction and transformation of RDF data. Being able to work with XML and RDF using a common framework eliminates several unnecessary steps that are currently required when handling both formats side by side. In this paper we present the XSPARQL language that, by combining XQuery and SPARQL, allows to query XML and RDF data using the same framework and transform data from one format into the other. We focus on the semantics of this combined language and present an implementation, including discussion of query optimisations along with benchmark evaluation.

S. Bischof · A. Polleres
Siemens AG Österreich, Siemensstrasse 90, 1210 Vienna, Austria
e-mail: bischof.stefan@siemens.com

A. Polleres
e-mail: axel.polleres@siemens.com

S. Decker · N. Lopes (✉)
Digital Enterprise Research Institute (DERI), National University of Ireland, Galway, Ireland
e-mail: nuno.lopes@deri.org

S. Decker · N. Lopes
IDA Business Park, Lower Dangan, Galway, Ireland
e-mail: stefan.decker@deri.org

T. Krennwallner
Institute of Information Systems, Vienna University of Technology, Favoritenstraße 9-11, 1040 Vienna, Austria
e-mail: tkren@kr.tuwien.ac.at

Keywords Query processing · XML · RDF · SPARQL · XQuery · XSPARQL

1 Introduction

XML [18] has become a well established and widely adopted interchange format for data on the Web. Accompanying standards, such as XSL Transformations (XSLT) by Kay [42] and, more recently, XQuery by Chamberlin et al. [22], both based on the XML Path Language (XPath) [11], are often used to query XML data and convert between different XML representations.

In the effort to convert the Web into a Semantic Web, the Resource Description Framework (RDF) [46,39] has become the language of choice for modelling, interlinking, and merging data. RDF data and applications that consume this data are becoming increasingly present on the Web. Opposed to the tree structure of XML, RDF structures data in sets of triples, representing edges of a directed, labelled graph. Querying RDF graphs and converting between them can be performed using SPARQL [54], the W3C recommended query language for RDF.

In many applications combining and converting between XML and RDF data is a useful but often not trivial task. The importance of this issue is acknowledged within the W3C, for instance in the working groups on Gleaning Resource Descriptions from Dialects of Languages (GRDDL) by Connolly [24] and Semantic Annotations for WSDL (SAWSDL) by Farrell and Lausen [29]. As we will show, common approaches for transformations between XML and RDF, which rely on the standard XML serialisation of RDF by Beckett and McBride [9] and on XML technologies, e.g., XSLT, have several disadvantages. While both XQuery and SPARQL languages operate on different data models, respectively, the XQuery and XPath Data Model (XDM) [30] for XML and RDF, we show that the merge of both query languages in the novel language XSPARQL has the potential to finally bring XML and RDF closer together. XSPARQL provides concise and intuitive solutions for mapping between XML and RDF in either direction: operations where both XQuery and SPARQL struggle. In fact it is not possible to use SPARQL alone for such transformations since the SPARQL query language does not provide the possibility of handling XML data. On the other side, the only way to work with RDF data within XQuery is by relying on the RDF/XML serialisation for RDF graphs. As we show in Sect. 2, this approach is hard to implement due to the different possible serialisations in RDF/XML for a single RDF graph. An additional use for XSPARQL is the conversion between RDF graphs. XSPARQL extends SPARQL's expressiveness for such transformations, by allowing, for instance, nested XSPARQL queries in the graph construction step.

Since its first version by Akhtar et al. [4], XSPARQL has gained community interest and practical use cases have been presented in a W3C Member Submission [49]. Based on these experiences, the present article makes the following main contributions:

- we present syntax and formal semantics of XSPARQL based on the XQuery Formal Semantics by Draper et al. [27]. In comparison with our initial publication, we improved the treatment of nested queries over RDF with respect to blank nodes and allow for assignment of RDF graphs to variables;
- our implementation of XSPARQL is based on rewriting an XSPARQL query into a semantically equivalent XQuery query; as opposed to the preliminary version of this rewriting by Akhtar et al. [4], in this paper we present a more tightly integrated, new prototype implementing several new features;
- we prove various properties of XSPARQL and show soundness and completeness of the new tighter query rewriting;
- we present a set of optimisations for complex queries (containing nested XSPARQL queries) and show their correctness;
- we introduce a novel benchmark suite (XMarkRDF) that extends the XMark XML Benchmark suite by also considering RDF as a data format; and
- based on the XMarkRDF suite, we present benchmark evaluation of the new XSPARQL prototype and compare it with a related system. Furthermore, we discuss the performance impact of the proposed optimisations.

The article is organised as follows: Sect. 2 will illustrate our main motivation to come up with a new language by discussing drawbacks of existing technologies for transformations between RDF and XML. In Sect. 3 we will briefly review the main characteristics of the XQuery and SPARQL query languages and, in Sect. 4, present their combination in the form of the XSPARQL language by defining the formal semantics and showing semantic properties of the novel language. Section 5 shows the architecture and query rewriting techniques for a prototype implementation. Section 6 discusses query optimisation techniques that speed up the evaluation of XSPARQL queries. We compare XSPARQL with another prototype that combines SPARQL and XQuery in Sect. 7 and report on experimental results using the benchmark suite XMarkRDF. We also compare query response times of the presented optimisations, showing promising results. We conclude this work with a discussion of related works in Sect. 8 and wrap up in Sect. 9.

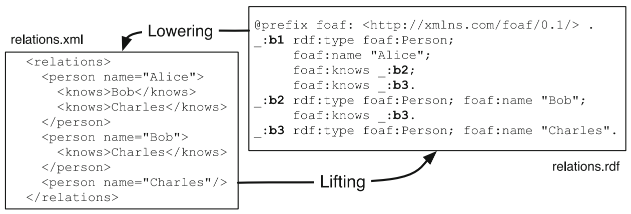
2 Motivation: Lifting and Lowering

XML can be viewed as a tree-like data representation format, with intermediate nodes of this tree being XML elements or attribute names, and the leaf nodes being either empty elements or textual attribute values and element content. The order of child nodes is relevant in XML. As opposed to this, RDF data, i.e., an RDF graph, is an unordered set of subject–predicate–object triples, as follows:

Definition 1 (*RDF Triple, RDF Graph*) Given pairwise disjoint sets of URI references $\mathbb{U}$, blank nodes $\mathbb{B}$, and literals $\mathbb{L}$,[1] a triple $(s, p, o) \in \mathbb{UB} \times \mathbb{U} \times \mathbb{UBL}$ (often written as a "statement" '$s\ p\ o$.') is called an *RDF triple*; sets of RDF triples are called *RDF graphs*. We call elements of $\mathbb{UBL}$ *RDF terms*.

Besides the normative syntax to exchange RDF using XML, RDF/XML [9], there are various serialisation formats for RDF, such as RDFa [2], a format that allows one to embed RDF within (X)HTML, or non-XML representations such as the more human-readable Turtle [7] syntax. Since data in RDF may be considered on a higher level of abstraction than semi-structured XML data, the translation from XML to RDF is often called *lifting*, while the opposite direction is called *lowering*. The importance of converting data between the XML and RDF formats has been acknowledged

[1] For brevity we will denote the concatenation of sets by concatenating their names, e.g., $\mathbb{U} \cup \mathbb{B}$ is represented as $\mathbb{UB}$.



```
<relations>
  <person name="Alice">
    <knows>Bob</knows>
    <knows>Charles</knows>
  </person>
  <person name="Bob">
    <knows>Charles</knows>
  </person>
  <person name="Charles"/>
</relations>
```

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
_:b1 rdf:type foaf:Person;
     foaf:name "Alice";
     foaf:knows _:b2;
     foaf:knows _:b3.
_:b2 rdf:type foaf:Person; foaf:name "Bob";
     foaf:knows _:b3.
_:b3 rdf:type foaf:Person; foaf:name "Charles".
```

Fig. 1 From XML to RDF and back: "lifting" and "lowering"

within the W3C in several related standardisation efforts, such as GRDDL and SAWSDL. The GRDDL working group addressed the issue of extracting RDF data out of existing (X)HTML Web pages (lifting). Likewise, in the Semantic Web Services community, the SAWSDL working group aimed at defining mechanisms (and link them in Web service descriptions) to generate XML messages sent to Web services from RDF data (lowering) and vice versa extract RDF from service result messages in XML (lifting) [see 29,44]. Both GRDDL and SAWSDL use XSLT (although they acknowledge that other mechanisms could be used) in their examples to perform lifting and lowering. In the following, let us illustrate some drawbacks of this approach.

As a running example throughout this paper we use a mapping between a custom XML format and RDF as shown in Fig. 1 (using Turtle syntax for illustration). The task is, in both directions, to extract for all persons the names of people they know. URIs denoting predicates and terms in a particular domain are typically collected under a common namespace in RDF with a designated prefix, such as RDF core terms in the namespace http://www.w3.org/1999/02/22-rdf-syntax-ns# using prefix `rdf:` or terms of the FOAF [20] ontology in the namespace http://xmlns.com/foaf/0.1/ using prefix `foaf:`.[2]

Blank nodes are represented in Turtle by the prefix '`_:`' followed by an identifier/label, or by square brackets '[]'. Blank nodes play a special role in RDF's data model: they allow to model unknown nodes or incomplete data, akin to existential variables. Regarding the serialisation in Turtle that means, if we would replace `_:b1` in Fig. 1 by `_:x`, it would represent an equivalent RDF graph.

RDF/XML [9] is the recommended syntax for RDF, using XML as the underlying representation model. This format enables the use of XML tools such as XSLT or XQuery to translate between RDF/XML and other XML formats. However, such a transformation is greatly complicated by the flexibility the RDF/XML format offers in serialising RDF graphs. Therefore, tools that handle RDF/XML as XML data (and not as a sets of triples) need to take different possible representations into account. Figure 2 shows four versions of a subset of the RDF data from our running example that represent

```
@prefix alice: <alice/> .
@prefix foaf: <...foaf/0.1/> .

_:b1 rdf:type foaf:Person;
     foaf:knows _:b2.
_:b2 rdf:type foaf:Person;
     foaf:name "Bob".
```

(a) Turtle

```
<rdf:RDF xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
   <foaf:knows>
     <foaf:Person foaf:name="Bob"/>
   </foaf:knows>
  </foaf:Person>
</rdf:RDF>
```

(b) Concise XML/RDF

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#">
  <rdf:Description rdf:nodeID="b1">
    <rdf:type
      rdf:resource=".../Person"/>
    <foaf:knows rdf:nodeID="b2"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="b2">
    <rdf:type
       rdf:resource=".../Person"/>
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
</rdf:RDF>
```

(c) XML/RDF

```
<rdf:RDF xmlns:foaf="...foaf/0.1/"
       xmlns:rdf="...rdf-syntax-ns#">
    <rdf:Description rdf:nodeID="x">
      <foaf:knows rdf:nodeID="y"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="x">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <foaf:name>Bob</foaf:name>
  </rdf:Description>
  <rdf:Description rdf:nodeID="y">
    <rdf:type rdf:resource=".../Person"/>
  </rdf:Description>
</rdf:RDF>
```

(d) Verbose XML/RDF

Fig. 2 Different representations of the same RDF graph

the same FOAF data. Figure 2a uses Turtle [7], a simple and readable textual format for RDF, inaccessible to pure XML processing tools though; the remaining three versions are all RDF/XML, ranging from concise (2b) to verbose (2d). These three RDF/XML variants represent different XML trees but the same RDF graph. Note that blank node identifiers may disappear or change through XML serialisation.

For our running example, let us attempt lifting and lowering transformations using XSLT (we will get to XQuery as another alternative in more detail later on).

[2] In listings and figures we sometimes abbreviate well-known namespace URIs with "...".

```
<xsl:stylesheet xmlns:xsl="...XSL/Transform"
  xmlns:foaf="...foaf/0.1/"
  xmlns:rdf="...rdf-syntax-ns#" version="2.0">

<xsl:template match="/relations">
  <rdf:RDF> <xsl:apply-templates /> </rdf:RDF>
</xsl:template>

<xsl:template match="person">
 <foaf:Person>
   <foaf:name>
     <xsl:value-of select="./@name"/>
   </foaf:name>
   <xsl:apply-templates/>
 </foaf:Person>
</xsl:template>

<xsl:template match="knows">
 <foaf:knows><foaf:Person><foaf:name>
   <xsl:apply-templates/>
 </foaf:name></foaf:Person></foaf:knows>
</xsl:template>

</xsl:stylesheet>
```

(a) XSL transform *lifting.xsl*

```
@prefix foaf: <http://xmlns.com/foaf/0.1/>.
_:b1 a foaf:Person;     foaf:name "Alice" ;
     foaf:knows _:b2 ; foaf:knows _:b3 .
_:b2 a foaf:Person ;   foaf:name "Bob" .
_:b3 a foaf:Person ;   foaf:name "Charles" .
_:b4 a foaf:Person ;   foaf:name "Bob" ;
     foaf:knows _:b5 .
_:b5 a foaf:Person ;   foaf:name "Charles" .
_:b6 a foaf:Person ;   foaf:name "Charles" .
```

(b) Result in Turtle

Fig. 3 Lifting attempt by XSLT

```
<xsl:stylesheet version="1.0"
  xmlns:rdf="...rdf-syntax-ns#"
  xmlns:foaf="...foaf/0.1/"
  xmlns:xsl="...XSL/Transform">
<xsl:template match="/rdf:RDF">
  <relations>
    <xsl:apply-templates select=".//foaf:Person"/>
  </relations>
</xsl:template>
<xsl:template match="foaf:Person">
  <person name="{./@foaf:name}">
    <xsl:apply-templates select="./foaf:knows"/>
  </person>
</xsl:template>
<xsl:template match="foaf:knows[@rdf:nodeID]">
  <knows>
    <xsl:value-of
     select="//foaf:Person[@rdf:nodeID=./@rdf:nodeID]/
          @foaf:name"/>
  </knows>
</xsl:template>
<xsl:template match="foaf:knows[foaf:Person]">
  <knows>
    <xsl:value-of select="./foaf:Person/@foaf:name"/>
  </knows>
</xsl:template></xsl:stylesheet>
```

Fig. 4 Lowering using XSLT (*lowering.xsl*)

```
@prefix rel: <http://purl.org/vocab/relationship/>
_:b1 rdf:type foaf:Person; foaf:name "Alice";
     rel:engagedTo _:b2; rel:worksWith _:b3.
_:b2 rdf:type foaf:Person; foaf:name "Bob";
     rel:hasMet _:b3.
_:b3 rdf:type foaf:Person; foaf:name "Charles".
```

Fig. 5 RDF data using the relationship ontology

Lifting. The XSLT stylesheet in Fig. 3a for instance, could be used to generate RDF/XML (in the format presented in Fig. 2b) from the *relations.xml* file in Fig. 1. However, this first attempt does not yet accomplish the intended transformation since unique identifiers are not created for each person. This is easy to see in the Turtle version of the result of this transformation (presented in Fig. 3b): while in our example names should uniquely identify a person, in this transformation the same person is potentially given several different blank nodes.

Although a proper lifting transformation catering for all possible XML serialisations is doable in XSLT, the corresponding stylesheet would need to be far more involved.

Lowering. The simple XSLT stylesheet *lowering.xsl* in Fig. 4b is an attempt to perform the lowering task directly from RDF/XML. However, this XSLT will break if the input RDF/XML serialisation is in any other variant than the version in Fig. 2b. We could create a specific stylesheet for each of the presented variants, but creating one that handles all the possible RDF/XML forms would be much more complicated.

Apart from its syntactic ambiguities, processing RDF/XML via XSLT also loses another feature of RDF, namely its interplay with ontological information, e.g., RDF Schema. RDF Schema [19] (RDFS) allows to express subclass or subproperty hierarchies, which can be exploited by RDF tools capable of ontological inference. The RDF data from Fig. 1 could—rather than `foaf:knows`—use predicates from the relationship ontology,[3] which are all stated as subproperties of `foaf:knows`, as presented in Fig. 5. Similar considerations would apply if we attempted to perform the lifting and lowering using XQuery: since XML tools do not support ontological inference, we literally would need to implement an RDFS inference engine within XSLT or XQuery, to be able to implement a lowering mechanism that also works for this kind of RDF data. Given the availability of RDF tools and engines that readily offer RDFS support, this seems to be a dispensable exercise.

Benefits of an integrated language. In recognition of the above problems, the SAWSDL specification contains a non-normative example which performs a lowering transformation as a sequence of a SPARQL query followed by an XSLT transformation on SPARQL's query results XML format [23]. The advantage of such a two-step approach is first that since SPARQL works on the RDF data model, all the input data from Fig. 2 are considered to be equivalent. Second, if one deploys a SPARQL engine that supports RDFS inference also

[3] http://vocab.org/relationship/.

input data that involves ontologically related RDF vocabularies could be dealt with. For example, to get all persons who work with (`rel:worksWith`) or have met (`rel:hasMet`) Charles from the FOAF data described in Fig. 5, a simple SPARQL query would be enough:

```
select $person from <foaf.rdf>
where { $person foaf:knows [ foaf:name "Charles
     " ] . }
```

Although the approach proposed by the SAWSDL Working Group provides a good starting point, we argue that it can still be improved on several points: first, the detour through SPARQL's XML query results format seems to be an unnecessary burden. Second, a more tightly coupled integration of SPARQL and XML query languages can provide a more expressive language, beyond the capabilities of using SPARQL and XSLT or XQuery sequentially, and directly amenable to query optimisations. XSPARQL, the language proposed in the present paper, aims to provide exactly this: use cases that otherwise would require interleaved calls to SPARQL (typically requiring an implementation using an external programming framework) can be solved in XSPARQL directly, cf. the lowering example in Fig. 10. Moreover, as we will see, the combined language not only allows for concise lifting and lowering, but also may be viewed as an extension of SPARQL for RDF-to-RDF transformations, cf. the example in Fig. 8b below. Before we turn to these examples and XSPARQL in more detail, let us give a short overview of the languages XSPARQL builds on: XQuery and SPARQL.

Prolog:	**P**	`declare namespace` *prefix="namespace-URI"*
Body:	**F**	`for` *var* `[at` *posVar*`] in` *FLWOR expression*
	L	`let` *var* := *FLWOR expression*
	W	`where` *FLWOR expression*
	O	`order by` *FLWOR expression*
Head:	**R**	`return` *XML+ nested FLWOR expressions*

(a) Simplified schematic view on XQuery

Prolog:	**P**	`prefix` *prefix: <namespace-URI>*
Head:	**C**	`construct` { *template* }
Body:	**D**	`from / from named` *<dataset-URI>*
	W	`where` { *graph pattern* }
	M	`order by` *expression*
		`limit` *integer* > 0
		`offset` *integer* > 0

(b) Simplified schematic view on SPARQL

Fig. 6 An overview of XQuery and SPARQL

```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
let $persons := //*[@name or ../knows]
let $positions := distinct-values(
                  for $p in $persons return
                  if( $p[@name] ) then $p/@name
                       else data($p))
return
 <rdf:RDF> {
  for $n in $positions
  let $id := fn:index-of($positions, $n)
  return
   <foaf:Person rdf:nodeId="b{$id}">
    <foaf:name> { $n } </foaf:name>
    { for $k in $persons[@name=$n]/knows
      let $kn := if( $k[@name] ) then $k/@name else
         data($k)
      let $kid := fn:index-of($positions, $kn)
      return
       <foaf:knows>
        <foaf:Person rdf:nodeID="b{$kid}"/>
       </foaf:knows>
    }
   </foaf:Person>
 } </rdf:RDF>
```

Fig. 7 Lifting using XQuery

3 Preliminaries

XQuery allows for a convenient and concise syntax for XML query processing and XML transformation, while SPARQL is the standard for RDF querying and transformation. One of the major differences between XQuery and SPARQL resides on the ordering of their respective data models: while XML (and hence XQuery) is an intrinsically ordered data model, RDF is an unordered data model. As such, necessary mechanisms must be in place to ensure XQuery respects the ordering of the input. Queries in each of the two languages can roughly be divided in two parts: (i) the retrieval part (*body*) and (ii) the result construction part (*head*): this is presented schematically in Fig. 6. Our goal is to combine these components for both languages in a unified language, XSPARQL, where XQuery's and SPARQL's heads and bodies may be used interchangeably and even nested. We next outline some of the main aspects of XQuery and SPARQL relevant to their combination into XSPARQL. For a more detailed overview of XQuery and SPARQL we refer the reader to [22,27] and to [51,54].

3.1 XQuery

XQuery consists mainly of so-called **FLWOR** expressions, denoting the body (**FLWO**) and the head (**R**) of a query. The *ForClause*s (**F**) can be used to declare variables that iterate over XML sequences, returned, e.g., by an XPath expression, while `let` assignments (**L**) allow to bind values, e.g., the entire result of an XPath expression, to variables. A filter condition on the current variable bindings or processing order of results within a *ForClause* can be specified in the `where` part (**W**) and by the `order by` clause (**O**), respectively. In the head (**R**) arbitrary well-formed XML, nested XQuery expressions, or previously assigned variables are allowed following the `return` keyword. Together with a large catalogue of built-in functions [45], XQuery offers a flexible instrument for arbitrary XML transformations.

The lifting task of Fig. 1 can be solved with XQuery as shown in Fig. 7.[4] Please note that, due to the nature of XQuery, in this query we are generating RDF/XML, opposed to the more concise Turtle syntax from Fig. 1. The resulting query is quite involved, but completely addresses the lifting task, including unique blank node generation for each person. We first select, in variable `$persons` (line 3), all nodes representing person names: either `person` or `knows` nodes and next, for each different name, we keep a sequence with all the distinct person names (stored in variable `$positions`), which we will use as the blank node identifier for the person. Iterating over these distinct person names, we determine the person identifier (line 10). The nested `for` (lines 14–23) again iterates over persons in order to create nested `foaf:knows` elements: for each person name (`$n`) from the outer `for` expression, this nested expression selects the XML nodes that correspond to persons which `$n` knows (line 14) and creates the corresponding `foaf:knows` elements (lines 18–20). While this is a valid solution for lifting, we still observe the following drawbacks: (1) We still have to build RDF/XML manually and cannot make use of the more readable and concise Turtle syntax; and (2) if we had to apply XQuery for the lowering task, we still would need to cater for all kinds of different RDF/XML representations. Thus we still face the same problems as discussed in the XSLT solution in Sect. 2. However, both these drawbacks will be alleviated by adding SPARQL to XQuery. By combining XQuery and SPARQL, XSPARQL also simplifies the lifting process by allowing to use SPARQL *ConstructClause*s that generate RDF in Turtle format and by performing automatic validation of the generated RDF graphs.

Semantics. Next, let us give a short overview of the XQuery Formal Semantics [27], on which we will base XSPARQL's semantics; it is defined essentially via three types of rules: (i) *normalisation rules*, (ii) *static typing rules*, and (iii) *dynamic evaluation rules*. *Normalisation rules* are used to rewrite arbitrary XQuery expression to the XQuery Core language—a subset of XQuery that, while semantically equivalent, aims to be easier to define, implement, and optimise [41]. *Static typing rules* are used to assign a type to each XQuery expression, while the *dynamic evaluation rules* are responsible for producing the resulting XML from each expression and guaranteeing that the expression input is consistent with the typing information determined during the static analysis step. Any XQuery expression E is evaluated with regard to an *expression context* C that holds the static environment (statEnv) and the dynamic environment (dynEnv) up until the evaluation of E. Environments are composed of different components and hold information necessary to the evaluation of any XQuery expression: statEnv holds the information available during static analysis, for example the varType component holds variable type information, while the dynEnv environment contains information available during expression evaluation, like the value for variables that is stored in the varValue component. We refer to the static environment of C as statEnv(C) and to the dynamic environment as dynEnv(C) and we can access the different components by name: statEnv(C).varType and the specific value for element var of the a context can be accessed using statEnv(C).varType(var). In case the expression context C is not explicitly presented, statEnv and dynEnv can be used in place of statEnv(C) and dynEnv(C).

Normalisation rules are represented using mapping rules and, as an example, we present the following rule from Draper et al. [27] that illustrates the normalisation of consecutive *ForClause*s into XQuery Core:

$$\left[\!\!\left[\begin{array}{l}\texttt{for } \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\\ \mathit{OptPosVar}_1 \texttt{ in } \mathit{Expr}_1\\ \quad , \ldots ,\\ \quad \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\\ \quad \mathit{OptPosVar}_n \texttt{ in } \mathit{Expr}_n\ \mathit{ReturnClause}\end{array}\right]\!\!\right]_{\mathit{Expr}} \tag{N1}$$
$$==$$
$$\begin{array}{l}\texttt{for } \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\\ \mathit{OptPosVar}_1 \texttt{ in } [\![\mathit{Expr}_1]\!]_{\mathit{Expr}} \texttt{ return}\\ \ \ \cdots\\ \quad \texttt{for } \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\\ \quad \mathit{OptPosVar}_n \texttt{ in } [\![\mathit{Expr}_n]\!]_{\mathit{Expr}}\ \ [\![\mathit{ReturnClause}]\!]_{\mathit{Expr}}\end{array}$$

In normalisation rules, fixed-width font (like `for`) refers to specific keywords, and *italic* font refers to productions in the XQuery Core grammar [27, Appendix A]. *Static type rules*, and *dynamic evaluation rules* are represented using inference rules. For instance, the following static typing rule from Draper et al. [27] ensures that no expression has `empty` type except the empty sequence and functions in the *fs* namespace that are applied to empty parentheses ():

$$\frac{\begin{array}{c}\text{statEnv} \vdash \mathit{Expr} : \mathit{Type}\\ \text{statEnv} \vdash \mathit{Type} <: \texttt{empty}\\ \textbf{not}\left(\begin{array}{l}\text{Expr is the empty parentheses () } \textbf{or}\ \texttt{fn:data()}\\ \textbf{or}\ \text{any } \mathit{fs} \text{ function applied to empty parentheses ()}\end{array}\right)\end{array}}{\text{A static type error is raised for expression } \mathit{Expr}} \tag{S1}$$

The *judgements* statEnv $\vdash \mathit{Expr} : \mathit{Type}$ and statEnv $\vdash \mathit{Type} <:$ `empty` hold when, in the static environment statEnv, both *Expr* has type *Type* and *Type* is a subtype of `empty`, respectively. For all details of the XQuery Semantics we refer the reader to [27].

Typing Fernández et al. [30] describe the XQuery and XPath Data Model (XDM) that is used to define the input to XQuery and the values of any XPath expression. Draper et al. [27, Section 2.4] describe the formal notation for types that we use throughout this paper. This representation of types is used for specification purposes only and is not exposed to the end user by XQuery. As [27, Section 8.3.1] describe, it

[4] We assume this query is executed with the context item set externally to the document node of *relations.xml* file presented in Fig. 1.

is possible to match a *Value* against a specific *Type* by using the judgement *Value* **matches** *Type*.

3.2 SPARQL

In analogy to **FLWOR** in XQuery, we will denote its correspondent "**DWMC** expressions" in SPARQL. The body (**DWM**) offers the following features: a *dataset* (**D**), i.e., the set of (named) source RDF graphs, is specified in `from` (or `from named`) clauses. The `where` part (**W**) allows matching parts of the dataset by specifying a *graph pattern*. Such patterns can be simple triple patterns possibly involving variables, URI references, and literals,[5] or unions of graph patterns, optional patterns matching of parts of a graph, or patterns matching of named graphs, etc.

Definition 2 (*Graph Patterns,* [51]) Let $\mathbb{V}$ be an infinite set of variables, *graph patterns* are inductively defined as follows:

– a tuple $(s, p, o) \in \mathbb{ULV} \times \mathbb{UV} \times \mathbb{ULV}$, called *triple pattern*, is a graph pattern;
– a set of triple patterns, called *Basic Graph Pattern (BGP)*, is a graph pattern;
– if P and P' are graph patterns, then $(P\ P')$, $(P$ `optional` $P')$, and $(P$ `union` $P')$ are graph patterns;
– if P is a graph pattern and $i \in \mathbb{UV}$, then (`graph` $i\ P$) is a graph pattern; and
– if P is a graph pattern and R is a `filter` expression, then $(P$ `filter` $R)$ is a graph pattern.

For any pattern P, we write $vars(P)$ for the set of all variables occurring in P. A `filter` expression R can be composed from constants, elements of $\mathbb{ULV}$, comparison operators ('$=$', '$<$', '$>$', '$\leq$', '$\geq$') and logical connectives ('$\neg$', '$\wedge$', '$\vee$'), and *built-in* functions.[6]

The evaluation semantics of SPARQL consists of computing a sequence of *solution mappings*, i.e., sets of bindings for the variables in these patterns, matching them against the graphs in the dataset. Sequences of solution mappings as referred to simply as *solution sequences*.

SPARQL is agnostic to the actual XML representation of the underlying source graphs, which alleviates the pain of having to deal with different RDF/XML representations of the graphs in the dataset. Also several RDF source graphs [39] specified in consecutive `from` clauses can be merged transparently in SPARQL, however for XML tools this involves renaming of blank nodes at the pure XML level. Solution sequences can be ordered or sliced using *solution modifiers* (**M**) `order by`, `limit`, and `offset`.

```
prefix vc: <...vcard-rdf/3.0#>
prefix foaf: <...foaf/0.1/>
construct {$X foaf:name $FN.}
from <vc.rdf>
where { $X vc:FN $FN .}
```

(a) Mapping full names from vCard to FOAF in SPARQL

```
prefix vc: <...vcard-rdf/3.0#>
prefix foaf: <...foaf/0.1/>
construct { [] foaf:name
              {fn:concat($N," ",$F)}. }
from <vc.rdf>
where { $P vc:Given $N. $P vc:Family $F. }
```

(b) Mapping a given and family name to a full name in XSPARQL

Fig. 8 RDF-to-RDF mappings in SPARQL and in XSPARQL

In the head, SPARQL's `construct` clause (**C**) offers convenient and XML-independent means to create an output RDF graph. A `construct` *template* consists of a list of triple patterns in Turtle syntax. By instantiating this template with the variable bindings computed in the body, a result graph is created, which enables SPARQL to be used as a transformation language between different RDF formats (similar to XSLT and XQuery for transforming between XML formats). A simple example for mapping full names from the vCard/RDF [40] format to `foaf:name` is given by the SPARQL query in Fig. 8. Blank nodes in `construct` templates—as used in the query in Fig. 8b—play a special role, in that they are replaced by a fresh blank node for each solution sequence in the result graph.

Other possible types of SPARQL queries include `select`, `ask`, and `describe` queries: `select` queries simply return the bindings for variables present in the query (instead of using these bindings to instantiate the template like a `construct` clause), `ask` queries return a boolean answer, indicating whether the graph pattern produces any results, and `describe` queries are used to return information about a resource. For the aims of XSPARQL we support the *ConstructClause* that allows to produce RDF and the `select` expression that allows us to input RDF data—although with a different syntax as explained in Sect. 4.1.

Let us remark that SPARQL does not cater for the creation of new values, which on the contrary is an inherent feature of XQuery. By combining XQuery and SPARQL, we are also enabling SPARQL to use the full range of XPath/XQuery built-in functions [45].

Due to this, the query in Fig. 8 which attempts to merge family names and given names into a single `foaf:name` by

[5] Note that we do not allow blank nodes in graph patterns, and thus do not consider them in our definitions. This restriction does not affect the expressivity of SPARQL, implicit in [51], since blank nodes in query patterns can always be replaced equivalently with variables. See discussion in Sect. 4.2 below.

[6] For a complete list of built-in functions we refer the reader to [54].

calling the `fn:concat` function is beyond SPARQL's capabilities. As we will see, XSPARQL will not only reuse SPARQL for transformations from and to RDF, but also enable such advanced RDF-to-RDF transformations.

Semantics The semantics of SPARQL is defined by means of evaluation rules which are presented by [54, Section 12.5]. Here we only give an overview of the notion of Basic Graph Pattern (BGP) matching that we will use later to define the semantics of XSPARQL.

The matching of BGPs is done with regard to a specific RDF graph—the *active graph*—which is a graph contained in the dataset specified to the query. This matching is defined in terms of replacing variables from the BGP with RDF terms present in the active graph, where the function that maps query variables to RDF terms is called a *solution mapping*.

Definition 3 (*Solution Mapping*) A *solution mapping* [see 54, Section 12.1.6] is a partial function mapping SPARQL variables to RDF terms. The *domain* of a solution mapping μ, denoted $dom(\mu)$, is the set of variables for which μ is defined. Furthermore, we denote the value of variable $v \in \mathbb{V}$ according to solution μ as $\mu(v)$. Two solution mappings μ_1 and μ_2 are *compatible* if for any $v \in dom(\mu_1) \cap dom(\mu_2)$ it holds that $\mu_1(v) = \mu_2(v)$. The *union* of two compatible mappings μ_1 and μ_2 consists of the standard set-theoretical union $\mu_1 \cup \mu_2$.

The replacement of variables in a graph pattern according to a solution mapping is defined next.

Definition 4 Let P be a graph pattern and μ be a solution mapping. The *variable substitution of P by* μ, denoted $\mu(P)$, is the graph pattern P with all variables $v \in vars(P) \cap dom(\mu)$ substituted by $\mu(v)$.

Finally, the definition of BGP matching from [54, Section 12.3] specifies the solutions to a query.

Definition 5 (*Basic Graph Pattern Matching*) We say μ is a *solution* for a BGP P with respect to the active graph G, if there exists a solution mapping μ' such that $\mu'(P)$ is a subgraph of G, and μ is the restriction of μ' to the variables in $vars(P)$.

The definition of BGP matching is extended to more complex SPARQL query patterns (including `union`, `optional`, `graph`, `filter`, etc.) by the SPARQL algebra [54, Section 12.4], such that the `where` clause of every SPARQL query—i.e., any **DWM** body—returns a *list* of solutions. We denote this evaluation of a SPARQL Graph Pattern P over a dataset D as $eval(D, P)$. As presented by [51], the evaluation of a SPARQL graph pattern can be specified by mapping the graph pattern to relational algebra operators. Since Perez et al. deal with set-based semantics of SPARQL, here we extend their notion of the *join* operator to solution sequences. Let Ω_1 and Ω_2 be two solution sequences; then $\Omega_1 \bowtie \Omega_2 = ToList(\{\mu_1 \cup \mu_2 \mid (\mu_1, \mu_2) \in ToMultiSet(\Omega_1) \times ToMultiSet(\Omega_2),\ \mu_1$ and μ_2 are compatible$\})$, where by $\times$ we denote the Cartesian product of the multisets and $ToList()$ is, as per the SPARQL specification [cf. 54, Section 12.4], an operation that turns a multiset into a sequence with the same elements and arbitrary ordering. Analogously to the $ToList()$ operation, $ToMultiSet()$ converts a sequence into a multiset by preserving duplicates but disregarding the sequence ordering.

For further details on the SPARQL query language, we refer the reader to the W3C specification [54].

Next, we define the notion of inclusion of solution sequences.

Definition 6 Let Ω_1 and Ω_2 be solution sequences. We say Ω_1 *is included in* Ω_2, denoted $\Omega_1 \preceq \Omega_2$, if for all solution mappings $\mu_1 \in ToMultiset(\Omega_1)$ there exists a solution mapping $\mu_2 \in ToMultiset(\Omega_2)$ such that $\mu_1 \subseteq \mu_2$.

Please note that this definition extends the notion of subset between multisets by considering also the subset relation between their elements, i.e., solution mappings. This definition will be required for the optimisations presented in Sect. 6. Since the presented optimisations are not order preserving we rely only on the notion of inclusion.

In a `construct` query, the solutions of the pattern in the `where` clause of the body are then used to instantiate the `construct` template and the result graph is obtained from the union of all valid RDF triples resulting from such instantiation. As mentioned before, for each solution, blank nodes occurring in a `construct` template are replaced by new blank nodes with new identifiers.

Apart from `construct` queries, which we mainly focus on here, SPARQL also allows `select` queries, which return sequences of variable bindings, obtained from projecting only solution mappings for a given list of variables.

4 XSPARQL

Conceptually, XSPARQL is a merge of SPARQL `construct` queries into XQuery. This combination of languages allows us to benefit from the facilities of SPARQL for retrieving RDF data and to use Turtle-like syntax for constructing RDF graphs, while still having access to all the features from XQuery for XML processing. In XSPARQL we allow any native XQuery query and we extend XQuery's **FLWOR** expressions to what we call **FLWOR'** expressions:

(i) In the body we allow SPARQL-style **F'DWM** blocks alternatively to XQuery's **FLWO** blocks. The new **F'** clause of the form `for`*varlist* is very similar to XQuery's native *ForClause*, but instead of allowing a single variable (which is assigned to the results of an XPath expression), the new clause supports a white space-separated list of variables (*varlist*). Each variable in *varlist*

```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
let $persons := //*[@name or ../knows]
let $positions := distinct-values(
                     for $p in $persons return
                     if( $p[@name] ) then $p/@name
                          else data($p))
return
 for $n in $positions
 let $id := fn:index-of($positions, $n)
 construct {
  _:b{$id} a foaf:Person ; foaf:name {data($n)} .
  { for $k in $persons[@name=$n]/knows
    let $kn := if( $k[@name] ) then $k/@name else data
         ($k)
    let $kid := fn:index-of($positions, $kn)
    construct {
     _:b{$id} foaf:knows _:b{$kid} .
     _:b{$kid} a foaf:Person .
    } } }
```

Fig. 9 Lifting in XSPARQL

```
declare foaf = "http://xmlns.com/foaf/0.1/";
<relations> {
  for $Person $Name from <relations.rdf>
  where { $Person foaf:name $Name }
  order by $Name
  return
    <person name="{$Name}"> {
      for $FName
      where {
        $Person foaf:knows $Friend .
        $Friend foaf:name $Fname
      }
      return <knows>{$FName}</knows>
    } </person>
} </relations>
```

Fig. 10 Lowering using XSPARQL

is then assigned the value resulting from evaluating a SPARQL query of the form: `select`*varlist* **DWM**.

(ii) In the head we allow to create RDF graphs directly using `construct` templates (**C**) alternatively to XQuery's native `return` (**R**).

(iii) Different forms of nesting are allowed, for example subqueries that construct RDF graphs may appear in `let` assignments which are later used in SPARQL-style `from` clauses, or can be used for value construction within SPARQL-style `construct` templates.

These modifications allow us to reformulate the lifting query of Fig. 7 on page 151 into its slightly more concise XSPARQL version of Fig. 9. The real power of XSPARQL in our example becomes apparent on the lowering part, where all of the other languages observed so far struggled. The lowering query for our running example is shown in Fig. 10.

4.1 Syntax of XSPARQL

In more detail, the XSPARQL syntax is an extension of the grammar rules in XQuery [22]. Figure 11 shows a schema of our merge of XQuery and SPARQL. For the definition of the XSPARQL syntax, we assume to inherit all the grammar productions of SPARQL [54] and XQuery [22] and mark any

Prolog:	**P**	`declare namespace` *prefix*="*namespace-URI*" or `prefix` *prefix*: <*namespace-URI*>	
Body:	**F**	`for` *var* [`at` *posVar*] `in` *FLOWR' expression*	
	L	`let` *var* := *FLWOR' expression*	
	W	`where` *FLWOR' expression*	
	O	`order by` *FLWOR' expression*	**or**
	F'	`for` *varlist* [`at` *posVar*]	
	D	`from` / `from named` (<*dataset-URI*> or *var*)	
	W	`where` { *pattern* }	
	M	`order by` *expression*	
		`limit` *integer* > 0	
		`offset` *integer* > 0	
Head:	**C**	`construct` { *template (with nested FLWOR' expressions)* }	**or**
	R	`return` *XML+ nested FLWOR' expressions*	

Fig. 11 Schematic view of XSPARQL

modified grammar productions with the prime symbol ($'$). We introduce two new productions: *SparqlForClause* and *ConstructClause*, corresponding to roughly to SPARQL `select` queries and `construct` templates; we present the grammar productions for these in Fig. 12. The full XSPARQL grammar can be found in [15]. In these grammar productions, the *WhereClause* and *SolutionModifier* correspond, respectively, to rules [13] and [14] from the SPARQL grammar, cf. [54, Appendix A.8].

The newly introduced *SparqlForClause*(rule [33a]) is similar to an XQuery `for` clause that can be used to iterate over SPARQL results.[7] This expression stands at the same level as XQuery's `for` and `let` expressions, i.e., such type of clauses are allowed to start new **FLWOR'** expressions, or may occur inside deeply nested XSPARQL queries.

The *ConstructTemplate*' expression is defined in the same way as the production *ConstructTemplate* in SPARQL [54], but we additionally allow nested XSPARQL expressions (*FLWORExpr*') in subject, predicate, and object positions; we achieve this by replacing SPARQL syntax rules *Verb* and *VarOrTerm* for *ConstructTemplate* with the rules *VarOrTerm*' and *Verb*' represented in Fig. 12b.[8]

The rules for `SourceSelector` from the SPARQL syntax are also extended (as presented in Fig. 12c), i.e., we allow for graphs in a SPARQL dataset to be specified by a variable which must evaluate to a URI.

In analogy to SPARQL's `select *` shortcut, we allow to write `for *` in place of `for` [*list of all unbound variables appearing in the WhereClause*] for *SparqlForClauses*; as syntactic sugar this is also the default value for the **F'** clause whenever a SPARQL-style *WhereClause* is found and a corresponding **F'** clause is missing. Please note that for *SparqlForClauses* we do not allow XQuery *QNames* as variable

[7] This expression does not have the exact semantics of a SPARQL `select` clause—returning bindings to variables—but rather adds new variables to the query; hence a syntax inspired by the existing XQuery *ForClauses* was chosen.

[8] These changes are highlighted in Fig. 12 by using **bold face**.

```
[33]  FLWORExpr'       ::= (((ForClause | LetClause)+ XQWhereClause? OrderByClause?) | SparqlForClause)
                           ("return" ExprSingle | ConstructClause)
[33a] SparqlForClause  ::=  "for" (VarName+ | "*") DatasetClause? WhereClause? SolutionModifier
[33b] ConstructClause  ::=  "construct" ConstructTemplate'
```

(a) XSPARQL core syntax elements, extending [22, Appendix A]

```
[37] Verb'            ::=  VarOrIRIref | "a" | "{" FLWORExpr' "}"
[42] VarOrTerm'       ::=  Var | GraphTerm | "{" FLWORExpr' "}"
```

(b) Modified *ConstructTemplate* syntax elements, extending [54, Appendix A]

```
[12] SourceSelector'   ::=  IRIref | "$" VarName
```

(c) Modified DatasetClause syntax elements, extending [54, Appendix A]

Fig. 12 Overview of XSPARQL syntax

names (further details are available in [27, Section 3.1.1.1]) and assume that only unprefixed variables are shared between the XQuery and SPARQL expressions of XSPARQL. By this treatment, XSPARQL becomes a syntactic superset of native SPARQL `construct` queries, since we additionally allow the following:

(1) XQuery and SPARQL namespace declarations (**P**) may be used interchangeably; and
(2) SPARQL-style `construct` result forms (**C**) may appear before the retrieval part for queries. This feature is mainly added in order to encompass SPARQL style queries, but in principle, we expect the (**R/C**) parts to appear in the end of a **FLWOR'** expression.

Thus, the query of Fig. 8a on page 153 or any other SPARQL `construct` queries remain valid syntax for XSPARQL.

4.2 Semantics of XSPARQL

Next we define the semantics of XSPARQL. After introducing some new types, used in the semantics, and an extension to the normalisation rules of XQuery *ForClause*s, we will turn to extending the notion of Basic Graph Pattern matching (Sect. 4.2) to make SPARQL clauses aware of the bindings for variables from XQuery. Then, we present the semantics of the newly introduced expressions: *SparqlForClause*(Sect. 4.2) and *ConstructClause* (Sect. 4.2), based on XQuery's formal semantics [27], by defining normalisation, static type and dynamic evaluation rules for each of the new expressions.

XSPARQL Types. We extend the XQuery and XPath Data Model (XDM), described by Fernández et al. [30], with the following new types that accommodate for SPARQL specific parts of XSPARQL:

(1) the `RDFTerm` type further consists of the subtypes `uri`, `bnode` and `literal` and is used as the type of SPARQL variables;

```
define type URI-reference restricts xs:anyURI;
define type Literal extends xs:string {
      attribute datatype of type URI-reference?,
      attribute lang of type xml:lang? };
define type RDFTerm {
      element uri of type URI-reference |
      element bnode of type xs:string |
      element literal of type Literal };
define type Binding extends RDFTerm {
      attribute name of type xs:string };
define element binding of type Binding;
define type Result {
      element binding* };
define type PatternSolution {
      element result of type Result };
define type RDFGraph {
      element triple of type RDFTriple* };
define type RDFTriple {
      element subject of type RDFTerm,
      element predicate of type RDFTerm,
      element object of type RDFTerm };
define element dataset of type RDFDataset;
define type RDFDataset {
      element defaultGraph of type RDFGraph,
      element namedGraphs of type RDFNamedGraphs };
define type RDFNamedGraphs {
      element namedGraph of type RDFNamedGraph* };
define type RDFNamedGraph {
      attribute name of type xs:string,
      element graph of type RDFGraph };
```

Fig. 13 XSPARQL Type Definitions

(2) the `PatternSolution` type consists of a set of pairs (*variableName*, `RDFTerm`) representing SPARQL variable bindings;
(3) the `RDFGraph` is the type of `construct` expressions; and
(4) the `RDFDataset` as the type for *DatasetClause*s.

The formal definition of (1)–(4) is given in Fig. 13. The `RDFTerm` type is used to represent RDF terms (composed of URIs, blank nodes or literals). The type of SPARQL variables are represented by the `Binding` type, that consists of the variable name and the RDF term that is assigned to it. Finally, sequences of SPARQL variable bindings are represented by the type `PatternSolution`. This representation of SPARQL results is similar to the XML Schema of the SPARQL Query

Results XML Format, available at http://www.w3.org/2007/SPARQL/result.xsd.

The `RDFGraph` type corresponds to a sequence of `RDFTriples` which are in turn a complex type composed of `subject`, `predicate` and `object`. The `RDFDataset` type is defined as an `RDFGraph` that is considered the default graph and a sequence of `RDFNamedGraph`s represented by the `name` of the graph and the corresponding `RDFGraph`.

The following definition presents the translation between a SPARQL solution sequence and a sequence of `Result` type elements that we implement in XSPARQL.

Definition 7 (*Serialisation of Solution Sequences*) Given a solution sequence $\Omega = (\mu_1, \ldots, \mu_n)$ a serialisation of Ω into a sequence of `PatternSolution` is defined as follows:

– $serialise(\Omega) \Rightarrow serialise(\mu_1), \ldots, serialise(\mu_n)$

– $serialise(\mu) \Rightarrow$ `<result>` $\{\forall x \in dom(\mu), serialise(\mu, x)\}$ `</result>`

– $serialise(\mu, x) \Rightarrow$ `<binding name="`x`">` $\{term(\mu(x))\}$ `</binding>` ,

where $term(\mu(x))$ is

– `<uri>`$\mu(x)$`</uri>` if $\mu(x) \in \mathbb{U}$
– `<bnode>`$\mu(x)$`</bnode>` if $\mu(x) \in \mathbb{B}$
– `<literal>`$\mu(x)$`</literal>` if $\mu(x) \in \mathbb{L}$

Following the definition of the *serialise* function, in evaluation rules, we will refer to sequences of elements of type `PatternSolution` as Ω and to elements of type `Result` as μ.

Query Prolog Normalisation As stated previously, XQuery and SPARQL namespace declarations can be used interchangeably in the query prolog. Hence, we convert any SPARQL syntax prefix declaration to XQuery namespace declarations by the following normalisation rules:

$$[\![\texttt{prefix}\ NCName\texttt{:}\ \texttt{<}URILiteral\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare namespace}\ NCName\ \texttt{=}\ URILiteral\ \texttt{;}]\!]_{Expr} \tag{N2}$$

$$[\![\texttt{prefix :}\ \texttt{<}URILiteral\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare default namespace =}\ URILiteral\ \texttt{;}]\!]_{Expr} \tag{N3}$$

$$[\![\texttt{base}\ \texttt{<}URILiteral\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare base-uri}\ URILiteral\ \texttt{;}]\!]_{Expr} \tag{N4}$$

XQuery `for` normalisation In accordance with the SPARQL semantics, blank nodes in *ConstructTemplates* need to be distinctly instantiated for any solution mapping matching the body, i.e., for every solution for the *WhereClause* a new blank node identifier needs to be created in the resulting graph. To ensure this behaviour in XSPARQL *ConstructTemplates*, we will use *position variables*[9] from XQuery in *ForClauses* to generate these new blank node identifiers, i.e., we introduce position variables in any XQuery `for` expressions without position variables and also to make sure that XSPARQL *SparqlForClause*expressions have position variables. To handle the XQuery `for` expression, we change the normalisation rule of `for` expressions to XQuery Core `for` expressions (cf. Sect. 3.1):

$$\left[\!\!\left[\begin{array}{l}\texttt{for}\ \$VarName_1\ OptTypeDeclaration_1\\ OptPositionalVar_1\ \texttt{in}\ Expr_1,\\ \quad \ldots,\\ \quad \$VarName_n\ OptTypeDeclaration_n\\ \quad OptPositionalVar_n\ \texttt{in}\ Expr_n\\ \quad ReturnClause\end{array}\right]\!\!\right]_{Expr} \\ == \\ \begin{array}{l}\texttt{for}\ \$VarName_1\ OptTypeDeclaration_1\\ [\![OptPositionalVar_1]\!]_{PosVar}\ \texttt{in}\ [\![Expr_1]\!]_{Expr}\\ \texttt{return}\\ \quad \ldots\\ \quad \texttt{for}\ \$VarName_n\ OptTypeDeclaration_n\\ \quad [\![OptPositionalVar_n]\!]_{PosVar}\ \texttt{in}\ [\![Expr_n]\!]_{Expr}\\ \quad [\![ReturnClause]\!]_{Expr}\end{array} \tag{N5}$$

A new normalisation rule $[\![\cdot]\!]_{PosVar}$ takes care of introducing new positional variables where necessary. We assume that the introduced position variables are distinct from any of the variables in scope, represented by the formal semantics variable $\$fs{:}new$ [cf. 27, Section 4.12.6]: $[\![]\!]_{PosVar}$ == `at` $\$fs{:}new$. In case a positional variable is already present it is reused: $[\![$`at` $\$PosVar]\!]_{PosVar}$ == `at` $\$PosVar$.

We also assume a new static environment component statEnv.posVars which consists of a sequence holding all positional variables in the given static environment, that is, the variables defined in the `at` clause of enclosing `for` expressions. The static type rules for the `for` expression [cf. 27, Section 4.8.2] need to be extended accordingly to store these positional variables, similar to the rules for *SparqlForClause*s in Sect. 4.2 below.

XSPARQL BGP Matching In this section we extend the notion of Basic Graph Pattern (BGP) matching described by [54, Section 12.3], to provide SPARQL with the variable bindings from XQuery. For this we rely on the XQuery varValue dynamic environment component that maps variable names to their value and consider this environment component as defining a set of bindings in the spirit of SPARQL solution mappings (as presented in Definition 3). Along these lines, we will consider the varValue component of the dynamic environment in which a SPARQL graph pattern P is executed the basis for the *XSPARQL instance mapping* of P. The transformation from the dynEnv.varValue into the XSPARQL instance mapping is defined next:

[9] Position variables are variables that appear in an XQuery *ForClauses* after the optional `at` keyword—cf. Fig. 6a on page 151—and bind to an integer indicating the current position in the `for`-expression.

Definition 8 (*XSPARQL instance mapping*) Let C be an expression context, and $D_C = \text{dynEnv}(C)$.varValue be the varValue and $T_C = \text{statEnv}(C)$.varType be the varType component of the static environment of C, respectively. The *XSPARQL instance mapping* μ_C is a solution mapping where, for each mapping $v_i \rightarrow x_i \in D_C$, x_i is converted into an instance of type `RDFTerm` or an RDF *Collection* according to the following conditions:

– if $x_i = ()$ and T_C.varType$(v_i) =$ `RDFTerm` then $\mu_C(x_i)$ is undefined;
– if $x_i = ()$ and T_C.varType$(v_i) \neq$ `RDFTerm` then $\mu_C(x_i) = ()$ is an empty RDF Collection;
– if x_i is a singleton sequence, then $\mu_C(x_i) = RDFTerm(x_i)$;
– if $x_i = (e_1, \ldots, e_n)$, $n > 1$, is a sequence then $\mu_C(x_i) = (RDFTerm(e_1) \cdots RDFTerm(e_n))$ to be read as an RDF Collection [46, Section 4.2] in Turtle notation [see 7, Section 3.5];

where $RDFTerm(x_i)$ is

– x_i if T_C.varType$(v_i) =$ `RDFTerm`,
– `"`x_i`"` if T_C.varType$(v_i) =$ `xsd:string`,
– `"`x_i`"^^rdf:XMLLiteral` if T_C.varType$(v_i) =$ `element()`,
– `"`$data(x_i)$`"` if T_C.varType$(v_i) =$ `attribute()`, and
– `"`x_i`"^^`T_C.varType(v_i), otherwise.

For a graph pattern P, we call the XSPARQL instance mapping of the expression context in which P is executed the *XSPARQL instance mapping of P*.

Next we define the notion of XSPARQL BGP matching based on the semantics of SPARQL BGP matching presented in Sect. 3.2.

Definition 9 (*Extended solution mapping*) Let C be an expression context. An *extended solution mapping* of a graph pattern P in C is a solution mapping *compatible* with the *XSPARQL instance mapping* of C.

XSPARQL BGP matching is defined analogously to the SPARQL BGP matching with the exception that we consider only extended solution mappings:

Definition 10 (*XSPARQL BGP matching*) Let P be a basic graph pattern, C be the expression context of P, and G be an RDF graph. We say that μ is a *solution* for P with respect to active graph G, if there exists an extended solution mapping μ' of C such that $\mu'(P)$ is a subgraph of G and μ is the restriction of μ' to the variables in *vars*(P).

This definition quasi injects the variable bindings inherited from XQuery into SPARQL patterns occurring within XSPARQL; by considering *extended solution mappings* the bindings returned for a BGP P will not only match the input graph G but also respect any bindings for variables in the dynamic environment. We can extend the XSPARQL BGP matching to generic graph patterns by following the SPARQL evaluation semantics, as described by [54, Section 12.4]. Considering a graph pattern P and μ_C the XSPARQL instance mapping of P, we similarly denote by $eval_{xs}(D, P, \mu_C)$ the evaluation of P over dataset D following XSPARQL BGP matching.

Matching blank nodes in nested queries As for the handling of explicit *DatasetClause*s we briefly review the scoping graph concept from SPARQL's semantics, presented in [54, Section 12]. Query solutions are taken from the *scoping graph*, a graph that is equivalent to the active graph but does not share any blank nodes with it or any graph pattern within the query. Although in XSPARQL we are not considering blank nodes in graph patterns, in the presence of nested *SparqlForClause*s XSPARQL instance mappings may in fact contain assignments of variables to blank nodes, injected from the outer *SparqlForClause* into the inner *SparqlForClause*. For example, in Fig. 10 on page 155, blank nodes bound in the outer *SparqlForClause* to the variable $*Person* will be injected into the inner *SparqlForClause*expression. In XSPARQL—as opposed to SPARQL patterns—such injected bnodes will be matched like constants against the blank nodes from the data, to enable coreference within nested queries over the same dataset. To ensure this behaviour, we introduce the notion of *active dataset*; nested queries over the same active dataset keep the same the scoping graphs. Any *SparqlForClause* with an *explicit DatasetClause* causes the *active dataset* to change, i.e., new scoping graphs (with fresh blank nodes) for each graph within it are created; if no *DatasetClause* is present in a nested *SparqlForClause* (implicit dataset), the active dataset remains unchanged.

We introduce another auxiliary function in the XSPARQL semantics, *fs*:*dataset*(*DatasetClause*), which returns an element of type `RDFDataset` based on the evaluation of its argument. This conversion is performed according to the SPARQL semantics presented in Sect. 3.2 and detailed in [54]. The static type signature of this function is

```
fs:dataset($datasetClause as xs:string)
as RDFDataset
```

We allow the `SourceSelector` of a *DatasetClause* to be specified by an element of type `uri` or `RDFGraph`. Elements of the type `uri` in the position of a graph will be mapped to graphs where the `uri` is used as its name. XSPARQL—just like the SPARQL specification—leaves the exact mapping of URIs to graphs open to particular implementations, but for the rest of this paper, we assume obtaining the RDF graph just by dereferencing the URI via HTTP.

SparqlForClause* and XQuery *ForClauses The semantics of *SparqlForClause* (Rule [33a], Fig. 12a) is defined by the following normalisation rules, static type analysis rules, and dynamic evaluation rules. We will also need to slightly adapt the static analysis rules for regular *ForClauses* (in order to properly deal with the extra position variables introduced by rule (N5)). We start with normalisation rules for *SparqlForClauses* with implicit variable selection (by means of "`for *`") and with explicitly stated variables:

$$\left[\!\!\left[\begin{array}{l}\texttt{for * } \mathit{OptDatasetClause\ WhereClause}\\ \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ExprSingle}\end{array}\right]\!\!\right]_{Expr} \\ == \\ \left[\!\!\left[\begin{array}{l}\texttt{for } [\![\mathit{WhereClause}]\!]_{vars}\\ \mathit{OptDatasetClause\ WhereClause}\\ \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ExprSingle}\end{array}\right]\!\!\right]_{Expr} \qquad \text{(N6)}$$

The normalisation rule $[\![\mathit{WhereClause}]\!]_{vars}$ determines all statically *unbound variables* present in the *WhereClause*, i.e., returns a whitespace separated list of all variables in the *WhereClause* that are not present in the statEnv.varType environment component. The next normalisation rule introduces a new position variable, analogously to the before-mentioned XQuery `for` normalisation rule, where $[\![\cdot]\!]_{PosVar}$ is as described above:

$$\left[\!\!\left[\begin{array}{l}\texttt{for } \$\mathit{VarName}_1 \ldots \$\mathit{VarName}_n\\ \mathit{OptDatasetClause\ WhereClause}\\ \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ExprSingle}\end{array}\right]\!\!\right]_{Expr} \\ == \\ \begin{array}{l}\texttt{for } \$\mathit{VarName}_1 \ldots \$\mathit{VarName}_n\ [\![]\!]_{PosVar}\\ \mathit{OptDatasetClause\ WhereClause}\\ [\![\texttt{return}\ \mathit{ExprSingle}]\!]_{Expr}\end{array} \qquad \text{(N7)}$$

Static type analysis The following static rule takes care of defining the types of variables present in a `for` expression as `RDFTerm`, adds the introduced position variables to statEnv.posVars, and determines the static type of the *SparqlForClause* expression:

$$\frac{\begin{array}{c}\text{statEnv.posVars} = (\mathit{PosVar}_1, \cdots, \mathit{PosVar}_n)\\ \text{statEnv} \vdash \mathit{PosVarName}\ \textbf{of var expands to}\ \mathit{PosVar}\\ \text{statEnv} + \text{posVars}(\mathit{PosVar}_1, \cdots, \mathit{PosVar}_n, \mathit{PosVar})\\ + \text{varType}\begin{pmatrix}\mathit{PosVar} \Rightarrow \texttt{xs:integer;}\\ \mathit{Var}_1 \Rightarrow \texttt{RDFTerm;}\\ \cdots;\ \mathit{Var}_n \Rightarrow \texttt{RDFTerm}\end{pmatrix}\\ \vdash \mathit{ExprSingle} : \mathit{Type}\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n\ \texttt{at}\ \mathit{PosVarName}\\ \mathit{DatasetClause\ WhereClause}\\ \mathit{SolutionModifier}\\ \texttt{return}\ \mathit{ExprSingle} : \mathit{Type}*\end{array}} \qquad \text{(S2)}$$

Please note that, since the variables included in a *SparqlForClause* are not allowed to contain a namespace prefix, we omitted the rules handling the namespace expansion for the respective variables. The static type rule for a *SparqlForClause*without an explicit *DatasetClause* is analogous. Likewise, note that we need to slightly adapt the standard static type checking rules for standard XQuery [27, Section 4.8.2], to populate the XSPARQL specific new static environment component statEnv.posVars:[10]

$$\frac{\begin{array}{c}\text{statEnv.posVars} = (\mathit{PosVar}_1, \cdots, \mathit{PosVar}_n)\\ \text{statEnv} \vdash \mathit{Expr}_1 : \mathit{Type}_1\\ \text{statEnv} \vdash \mathit{VarName}\ \textbf{of var expands to}\ \mathit{Var}\\ \text{statEnv} \vdash \mathit{VarName}_{pos}\ \textbf{of var expands to}\ \mathit{Var}_{pos}\\ \text{statEnv} + \text{posVars}(\mathit{PosVar}_1, \cdots, \mathit{PosVar}_n, \mathit{Var}_{pos})\\ + \text{varType}\begin{pmatrix}\mathit{Var} \Rightarrow \text{prime}(\mathit{Type}_1);\\ \mathit{Var}_{pos} \Rightarrow \texttt{xs:integer}\end{pmatrix}\\ \vdash \mathit{ExprSingle} : \mathit{Type}\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{VarName}\ \texttt{at}\ \$\mathit{VarName}_{pos}\\ \texttt{in}\ \mathit{Expr}_1\ \texttt{return}\ \mathit{ExprSingle} :\\ \quad \mathit{Type} \cdot \text{quantifier}(\mathit{Type}_1)\end{array}} \qquad \text{(S3)}$$

Dynamic Evaluation For the dynamic evaluation we have to introduce a new dynamic environment component called activeDataset that will be used to evaluate *WhereClauses*. Initially, this component is empty (or set to a system default) and is changed by a *DatasetClause* appearing in a *SparqlForClause*. We further introduce two auxiliary functions *fs*:*value* and *fs*:*sparql*.

fs:value The *fs*:*value*($\$PS$, $\$var$) function returns the value of the specified SPARQL variable $\$var$ in a `Pattern Solution` specified by $\$PS$. If $\$var$ is not bound in $\$PS$, the empty sequence is returned. This function is defined as

```
fs:value($ps as PatternSolution,
         $variable as xs:string)
  as RDFTerm?
```

fs:sparql The *fs*:*sparql* function corresponds to the adapted version of the *eval* function, the $eval_{xs}$ function, that evaluates graph patterns implementing the extended notion of BGP Matching (cf. Definition 10).

The static type signature of this function is defined as

```
fs:sparql($dataset as RDFDataset,
          $SparqlWhere as xs:string,
          $solutionModifiers as xs:string)
   as PatternSolution*
```

The parameters of the $eval_{xs}$ function correspond to the dataset `$dataset`, the SPARQL algebra expression generated from the graph pattern `$SparqlWhere` and `$solutionModifiers`, as described by [cf. 54, Section 12.2.3], and the XSPARQL instance mapping μ_C that is derived from the expression context C over which the *fs*:*sparql* function is evaluated. The result of $eval_{xs}$ consists of a solution sequence which, as a result of applying the *serialise* function (cf. Definition 7), can be translated directly into an XQuery sequence of XML elements of type `PatternSolution`.

We can now define the dynamic evaluation rules for the *SparqlForClause* expression. Intuitively, these rules state

[10] We show here only the adapted rule for *ForClauses* with position variables without type declaration, the rule handling both position variables and type declarations is adapted analogously.

that the return expression *ExprSingle* will be executed for each `PatternSolution` that is returned from the evaluation of the *fs*:*sparql* function. The following two dynamic rules specify the evaluation of the *SparqlForClause* with an explicit *DatasetClause*:

$$\frac{\begin{array}{l} \text{dynEnv} \vdash \mathit{fs}{:}\mathit{dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}\begin{pmatrix}\mathit{Dataset}, \mathit{WhereClause}, \\ \mathit{SolutionModifier}\end{pmatrix} \Rightarrow \mu_1, \ldots, \mu_m \\ \text{dynEnv} + \text{activeDataset}(\mathit{Dataset}) \\ \qquad + \text{varValue}\begin{pmatrix} \mathit{PosVar} \Rightarrow 1; \\ \mathit{Var}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_1, \mathit{Var}_1); \\ \ldots; \\ \mathit{Var}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_1, \mathit{Var}_n) \end{pmatrix} \\ \qquad \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_1 \\ \qquad\qquad \vdots \\ \text{dynEnv} + \text{activeDataset}(\mathit{Dataset}) \\ \qquad + \text{varValue}\begin{pmatrix} \mathit{PosVar} \Rightarrow n; \\ \mathit{Var}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_m, \mathit{Var}_1); \\ \ldots; \\ \mathit{Var}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_m, \mathit{Var}_n) \end{pmatrix} \\ \qquad \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_m \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for}\ \$\mathit{Var}_1 \cdots \$\mathit{Var}_n\ \texttt{at}\ \$\mathit{PosVar} \\ \mathit{DatasetClause}\ \mathit{WhereClause} \\ \mathit{SolutionModifier}\ \texttt{return} \\ \mathit{ExprSingle} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}} \tag{D1}$$

This rule ensures that the activeDataset component of the dynamic environment is updated to reflect the explicit *DatasetClause* of the *SparqlForClause*. If the evaluation of the *fs*:*sparql* function does not yield any solutions, i.e., evaluates to an empty sequence, the overall result will also be the empty sequence:

$$\frac{\begin{array}{l} \text{dynEnv.activeDataset} \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}\begin{pmatrix}\mathit{Dataset}, \mathit{WhereClause}, \\ \mathit{SolutionModifier}\end{pmatrix} \Rightarrow () \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for}\ \$\mathit{Var}_1 \cdots \$\mathit{Var}_n\ \texttt{at}\ \$\mathit{PosVar} \\ \mathit{DatasetClause}\ \mathit{WhereClause} \\ \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ExprSingle} \end{array} \Rightarrow ()} \tag{D2}$$

The rule that handles the *SparqlForClause* without an explicit *DatasetClause* is presented next:

$$\frac{\begin{array}{l} \text{dynEnv.activeDataset} \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}\begin{pmatrix}\mathit{Dataset}, \mathit{WhereClause}, \\ \mathit{SolutionModifier}\end{pmatrix} \Rightarrow \mu_1, \ldots, \mu_m \\ \text{dynEnv} + \text{varValue}\begin{pmatrix} \mathit{PosVar} \Rightarrow 1; \\ \mathit{Var}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_1, \mathit{Var}_1); \\ \ldots; \\ \mathit{Var}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_1, \mathit{Var}_n) \end{pmatrix} \\ \qquad \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_1 \\ \qquad\qquad \vdots \\ \text{dynEnv} + \text{varValue}\begin{pmatrix} \mathit{PosVar} \Rightarrow n; \\ \mathit{Var}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_m, \mathit{Var}_1); \\ \ldots; \\ \mathit{Var}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_m, \mathit{Var}_n) \end{pmatrix} \\ \qquad \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_m \end{array}}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for}\ \$\mathit{Var}_1 \cdots \$\mathit{Var}_n\ \texttt{at}\ \$\mathit{PosVar} \\ \mathit{WhereClause}\ \mathit{SolutionModifier} \\ \texttt{return}\ \mathit{ExprSingle} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}} \tag{D3}$$

Analogously to the *SparqlForClause* with an explicit dataset, whenever the *fs*:*sparql* function evaluates to an empty sequence, the result will also be an empty sequence.

ConstructClause We now define the semantics of the *ConstructClause* (Rule [33b], Fig. 12a) by means of normalisation rules. SPARQL stand-alone `construct` queries (as described in Sect. 4.1) are normalised into `construct` queries with a surrounding *ForClause*:

$$\begin{array}{c} \left[\!\!\left[\begin{array}{l} \texttt{construct}\ \mathit{ConstructTemplate}' \\ \mathit{DatasetClause}\ \mathit{WhereClause} \\ \mathit{SolutionModifier} \end{array} \right]\!\!\right]_{\mathit{Expr}} \\ == \\ \left[\!\!\left[\begin{array}{l} \texttt{for}\ *\ \mathit{DatasetClause} \\ \mathit{WhereClause}\ \mathit{SolutionModifier} \\ \texttt{construct}\ \mathit{ConstructTemplate}' \end{array} \right]\!\!\right]_{\mathit{Expr}} \end{array} \tag{N8}$$

The resulting query will be further rewritten according to aforementioned normalisation rule (N6). As introduced in Sect. 4.1, we allow nested XSPARQL expressions in subject, predicate, and object positions of *ConstructTemplate*'. These nested expressions are identified by the shortcuts `{Expr}`, `<{Expr}>`, and `_:{Expr}`, that construct elements of type `literal`, `uri`, and `bnode`, respectively.

Similar to the normalisation rule for stand-alone *ReturnClauses* presented in [27, Section 4.8.1], the following normalisation rule transforms `construct` clauses into XQuery `return` *ExprSingle* s.

$$\begin{array}{c} [\![\texttt{construct}\ \mathit{ConstructTemplate}']\!]_{\mathit{Expr}} \\ == \\ \texttt{return}\ \mathit{fs}{:}\mathit{evalCT}\big([\![\mathit{ConstructTemplate}']\!]_{\mathit{normCT}}\big) \end{array} \tag{N9}$$

In the following we assume that *ConstructTemplate*' is a simple "." separated list of *Subject*, *Predicate*, and *Object*. The $[\![\cdot]\!]_{normCT}$ rule transforms any Turtle shortcut notation used in *ConstructTemplate*' to these simple lists. As an example of this rule, we present the rule for normalising Turtle ";" abbreviations [cf. 7, Section 2.3]:

$$\begin{array}{c} [\![\mathit{Subject}\ \mathit{Pred}_1\ \mathit{Obj}_1; \ldots; \mathit{Pred}_n\ \mathit{Obj}_n]\!]_{\mathit{normCT}} \\ == \\ \mathit{Subject}\ \mathit{Pred}_1\ \mathit{Obj}_1\ \ldots\ \mathit{Subject}\ \mathit{Pred}_n\ \mathit{Obj}_n \end{array} \tag{N10}$$

The normalisation rules for the other Turtle shortcuts that are allowed in the SPARQL *ConstructTemplate*' syntax are similar to this one and are not presented here. Since anonymous blank nodes can be written in numerous ways in Turtle, the $[\![\cdot]\!]_{normCT}$ normalisation rule transforms each anonymous blank node into a labelled blank node where the identifier/label is distinct from any other blank node labels present in the *ConstructTemplate*'. Take, as an example, the *ConstructTemplate* in Fig. 8b on page 153. It is normalised as

```
{ _:b foaf:name {
    fn:concat($N, " ", $F)
  }.
}
```

fs:evalCT The *fs:evalCT* function is a new built-in function that ensures the created RDF graph is valid and rewrites any blank nodes inside of *ConstructTemplates* to comply with the SPARQL semantics (as described in Sect. 4.2). The auxiliary *fs:validTriple* function checks if each triple is valid according to the RDF semantics and is defined by rules (D5) and (D6). The static type signatures of these functions are defined as

```
fs:evalCT($template as RDFTerm*) as RDFGraph
fs:validTriple($subject as RDFTerm,
               $predicate as RDFTerm,
               $object as RDFTerm)
    as RDFTriple
```

The *fs:evalCT* function, and hence `construct` expressions, return elements of the previously defined type `RDFGraph`, thus allowing the result of `construct` expressions to be used in a *DatasetClause* of a subsequent *SparqlForClause*. In more detail, the *fs:evalCT* function checks the constructed RDF graph for validity according to the conditions described in Definition 1, filtering out any non-valid RDF triples where *subjects* are literals, *predicates* are literals or blank nodes, etc. This is illustrated by the following dynamic evaluation rules:

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \textit{fs:validTriple}(\textit{Subj}_1, \textit{Pred}_1, \textit{Obj}_1) \Rightarrow \textit{Triple}_1\\ \vdots \\ \text{dynEnv} \vdash \textit{fs:validTriple}(\textit{Subj}_n, \textit{Pred}_n, \textit{Obj}_n) \Rightarrow \textit{Triple}_n\end{array}}{\begin{array}{c}\text{dynEnv} \vdash \textit{fs:evalCT}\begin{pmatrix}\textit{Subj}_1\ \textit{Pred}_1\ \textit{Obj}_1\\ \dots\\ \textit{Subj}_n\ \textit{Pred}_n\ \textit{Obj}_n\end{pmatrix}\\ \Rightarrow \texttt{<triples>}\textit{Triple}_1 \dots \textit{Triple}_n\texttt{</triples>}\end{array}} \quad \text{(D4)}$$

The following dynamic evaluation rule for the *fs:validTriple* function checks, relying on the *fs:bnode* function defined below, if a triple is valid according to the RDF semantics:

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \textit{fs:bnode}(\textit{Subject}) \Rightarrow \textit{ValS}\\ \text{statEnv} \vdash \textit{ValS}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode})\\ \text{dynEnv} \vdash \textit{Predicate} \Rightarrow \textit{ValP}\\ \text{statEnv} \vdash \textit{ValP}\ \textbf{matches}\ \texttt{uri}\\ \text{dynEnv} \vdash \textit{fs:bnode}(\textit{Object}) \Rightarrow \textit{ValO}\\ \text{dynEnv} \vdash \textit{ValO}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode} \mid \texttt{literal})\end{array}}{\begin{array}{l}\text{dynEnv} \vdash \textit{fs:validTriple}\begin{pmatrix}\textit{Subject},\\ \textit{Predicate},\\ \textit{Object}\end{pmatrix}\\ \quad\Rightarrow \texttt{element triple of type RDFTriple}\,\{\\ \quad\quad \texttt{element subject of type RDFTerm}\,\{\textit{ValS}\}\\ \quad\quad \texttt{element predicate of type RDFTerm}\,\{\textit{ValP}\}\\ \quad\quad \texttt{element object of type RDFTerm}\,\{\textit{ValO}\}\\ \quad\}\end{array}} \quad \text{(D5)}$$

In case any of the subject, predicate or object do not match an allowed type, the empty sequence is returned. Effectively this suppresses any invalid RDF triples from the output graph.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \textit{fs:bnode}(\textit{Subject}) \Rightarrow \textit{ValueS}\\ \text{dynEnv} \vdash \textit{Predicate} \Rightarrow \textit{ValueP}\\ \text{dynEnv} \vdash \textit{fs:bnode}(\textit{Object}) \Rightarrow \textit{ValueO}\\ \text{dynEnv} \vdash \textbf{not}\begin{pmatrix}\textit{ValueS}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode})\ \textbf{and}\\ \textit{ValueP}\ \textbf{matches}\ \texttt{uri}\ \textbf{and}\\ \textit{ValueO}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode} \mid \texttt{literal})\end{pmatrix}\end{array}}{\text{dynEnv} \vdash \textit{fs:validTriple}(\textit{Subject}, \textit{Predicate}, \textit{Object}) \Rightarrow ()} \quad \text{(D6)}$$

Blank Node Skolemisation In order to comply with the SPARQL `construct` semantics, all blank nodes inside a *ConstructTemplate*' need to be *skolemised*, i.e., for each solution a new distinct blank node identifier needs to be generated. Since we normalise every XQuery `for` expression and *SparqlForClauses* by assigning them position variables (as described in Sect. 4.1), we just need to retrieve the available position variables from the static environment component statEnv.posVars, and create the new distinct identifier based on the values of these variables. The *fs:bnode* function takes care of skolemising blank nodes. If the argument of this function is of type `bnode` a new blank node identifier is generated using rule (D7):

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \textit{ValueR}\ \textbf{matches}\ \texttt{bnode}\\ \text{statEnv.posVars} = (\textit{PosVar}_1, \dots, \textit{PosVar}_n)\\ \text{dynEnv.varValue}(\textit{PosVar}_1) = \textit{PosValue}_1\\ \vdots\\ \text{dynEnv.varValue}(\textit{PosVar}_n) = \textit{PosValue}_n\\ \text{dynEnv} \vdash \textit{fs:skolemConstant}\begin{pmatrix}\textit{ValueR},\\ \textit{PosValue}_1,\\ \dots,\\ \textit{PosValue}_n\end{pmatrix} \Rightarrow \textit{ValueRS}\end{array}}{\begin{array}{l}\text{dynEnv} \vdash \textit{fs:bnode}(\textit{ValueR}) \Rightarrow\\ \quad \texttt{element bnode of type xs:string}\,\{\textit{ValueRS}\}\end{array}} \quad \text{(D7)}$$

Otherwise, *fs:bnode* returns its argument unchanged as represented by rule (D8):

$$\frac{\text{dynEnv} \vdash \textit{Value}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{literal})}{\text{dynEnv} \vdash \textit{fs:bnode}(\textit{Value}) \Rightarrow \textit{Value}} \quad \text{(D8)}$$

Both rules above use the *fs:skolemConstant* function for the generation of the new identifiers based on the specified blank node label and on positional variables in the dynamic environment. An example of XSPARQL Semantics Evaluation is included in Appendix A.

4.3 Correspondence Between XSPARQL, XQuery, and SPARQL

Since XSPARQL syntactically extends XQuery, and—by the remarks in the end of Sect. 4.1—also any SPARQL `construct` query is syntactically valid in XSPARQL, these queries are considered semantically equivalent to the semantics in their base languages. The next propositions formally establish this intuitive correspondence.

The proofs for these propositions and lemmas are included in Appendix C.1–C.3.

Proposition 1 *XSPARQL is a conservative extension of XQuery.*

A similar correspondence holds for native SPARQL `construct` queries. We show the equivalence between SPARQL BGP Matching [54, Section 12.3.1] and XSPARQL BGP Matching (presented in Sect. 4.2) and prove the equivalence of XSPARQL semantics for native SPARQL `construct` queries with those of the SPARQL semantics.

Lemma 1 *Given a graph pattern P, a dataset D and the XSPARQL instance mapping μ_C of the expression context C over which P is evaluated, let $\Omega_1 = eval_{xs}(D, P, \mu_C)$ and $\Omega_2 = eval(D, P)$ be solution mappings. If $vars(P) \cap dom(\mu_C) = \emptyset$, then $\Omega_1 = \Omega_2 \bowtie \{\mu_C\}$.*

We define, based on [54, Section 10.2], the semantics of the SPARQL `construct` clause to show their equivalence to the XSPARQL `construct` clause.

Definition 11 (*SPARQL* `construct` *semantics*) Let C be a *ConstructTemplate* and Ω a *solution sequence*. The SPARQL `construct` returns an RDF graph generated by the set-union of the triples obtained from substituting variables in C with their bindings from Ω and satisfying the following conditions:

1. any invalid RDF triples that may be produced by the instantiation of the *ConstructTemplate* are ignored; and
2. blank node labels within the *ConstructTemplate* are considered scoped to the template for each solution, i.e., if the same label occurs twice in a template, then there will be one blank node created for each solution in Ω, but there will be different blank nodes for triples generated by different query solutions. Blank nodes in the graph template be shared only within the same query solution $\mu_i \in \Omega$.

For SPARQL `construct` queries we can state the following:

Proposition 2 *XSPARQL is a conservative extension of SPARQL* `construct` *queries.*

5 Implementation

In this section we present a prototype implementation of the XSPARQL language. The prototype translates an XSPARQL query into an XQuery query with interleaved calls to a SPARQL engine. The architecture of our implementation is shown in Fig. 14 and consists of three main components:(1) a query rewriter, which turns an XSPARQL query into an XQuery; (2) an XQuery engine for evaluating the XQuery; and (3) a SPARQL engine, for querying RDF from within the rewritten XQuery.

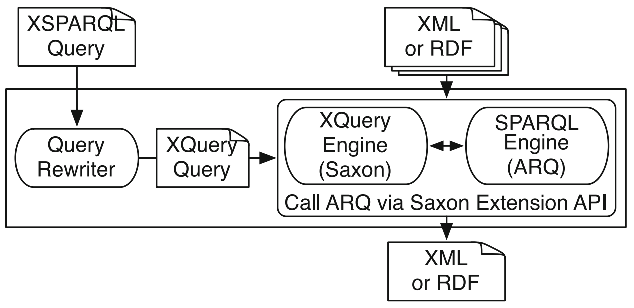


Fig. 14 XSPARQL implementation architecture

Our current prototype is meant first to demonstrate that XSPARQL can be implemented directly on top of off-the-shelf components, and provides convenient means to model and execute XML2RDF/RDF2XML transformations. Second, as illustrated in our evaluation section (Sect. 6) we show that a clever implementation of the XSPARQL language, again integrating an XQuery and a SPARQL engine, but with several optimisations in place, can improve efficiency significantly compared with a naive implementation.

In general it is possible to use any XQuery and SPARQL engines to evaluate XSPARQL queries. The current prototype implements the interface between the XQuery engine, Saxon 9.3,[11] and the SPARQL engine, ARQ 2.8.7,[12] using the Saxon Extension API which allows calling Java methods from within XQuery queries. The main function, called `xsp:sparqlCall`, evaluates a SPARQL query and returns its result using the SPARQL XML results format [8]. By using Saxon's extension mechanism the two query engines are tighter integrated allowing a more efficient communication than our former prototype [4] which used a SPARQL endpoint via HTTP to evaluate SPARQL queries. To implement the blank node handling as presented in Sect. 4.2 we changed the behaviour of ARQ accordingly using its Java API. Instead of implementing all newly introduced types as given in Sect. 4.2 as custom types in XQuery, we reuse types as given by the XML Schema of the SPARQL Query Results XML Format,[13] where the `sr:binding` type corresponds directly to XSPARQL's `RDFTerm` type. An `RDFGraph`, e.g., the result of a *ConstructClause*, is serialised using Turtle syntax by building the output as `xs:string`. The remaining types `RDFDataset` and `RDFNamedGraph` are adapted accordingly.

[11] http://saxon.sourceforge.net/.

[12] http://jena.sourceforge.net/ARQ/.

[13] See http://www.w3.org/2007/SPARQL/result.xsd, for this paper we assume this schema is associated with the namespace prefix `sr`.

Next we present how *SparqlForClause*s and *ConstructClause*s are processed using functions—called *rewriting functions*—that operate on syntactic objects of XSPARQL and returning an XQuery expression. In the resulting XQuery expressions we assume the namespace prefix `xsp:` associated with http://xsparql.deri.org/demo/xquery/xsparql.xquery. This prefix is not allowed to be used in any XSPARQL query and defines XQuery functions (presented below) that are available to the rewriting functions and used as the namespace for any variables introduced by the rewriting, thus avoiding clashes with variables from the XSPARQL query.

SparqlForClause First, our implementation defers SPARQL queries in a *SparqlForClause* to the external SPARQL engine and extracts the bindings for the SPARQL variables from the SPARQL XML results document that is returned, by the following rewriting function. Let $\mathbb{XS}$ and $\mathbb{XQ}$ denote the set of all XSPARQL core and XQuery core expressions, respectively. Any *SparqlForClause*is translated into an XQuery query that calls the SPARQL engine with a `select` query according to the rewriting function $tr\colon \mathbb{XS} \rightarrow \mathbb{XQ}$. Given an XSPARQL expression Q of form

$$\begin{array}{l} \texttt{for}\ \mathit{Vars}\ \texttt{at}\ \$\mathit{PosVar} \\ \mathit{DatasetClause} \\ \mathit{WhereClause} \\ \mathit{SolutionModifier} \\ \texttt{return}\ \mathit{ExprSingle} \end{array} \tag{Q1}$$

then $tr(Q)$ is defined as the XQuery Core expression

$$\begin{array}{ll} \multicolumn{2}{l}{tr(Q) =} \\ (1) & \texttt{let \$xsp:results :=} \\ & \texttt{xsp:sparqlCall}\left(\begin{array}{l}\texttt{select}\ \mathit{Vars}\ \mathit{DatasetClause} \\ \mathit{WhereClause} \\ \mathit{SolutionModifier}\end{array}\right)\ \texttt{return} \\ (2) & \texttt{for \$xsp:result at}\ \$\mathit{PosVar}\ \texttt{in \$xsp:results//sr:result} \\ & \texttt{return} \\ (3) & \texttt{let}\ \$v := \qquad\qquad \text{for each } \$v \in \mathit{Vars} \\ & \texttt{\$xsp:result/sr:binding[@name} = v\texttt{]/*}\ \texttt{return} \\ (4) & \mathit{ExprSingle} \end{array}$$

That is, we implement the *fs*:*sparql* formal semantics function by translating Q to a SPARQL `select` query, which is then executed by the custom runtime function `xsp:sparqlCall` that receives a SPARQL `select` query and returns the result in SPARQL's XML result format. The `xsp:sparqlCall` function also takes care of XSPARQL's BGP matching, as described in Sect. 4.2, by replacing in the SPARQL query any previously bound variables with their current value according to the rules presented in Definition 8, thus mimicking XSPARQL's BGP matching behaviour while relying on an off-the-shelf SPARQL engine. This replacement of variables is performed by producing XQuery code that generates the SPARQL `select` query string that is given as a parameter to `xsp:sparqlCall` function using XQuery's `fn:concat` function. We parse the query string for variables and, having access to the list of previously declared variables it is possible to determine whether variables should be replaced by their previously bound value or kept as a variable. Within a *SparqlForClause*, whenever we encounter fresh variables, i.e. that have not been declared before, we leave the variable name as a string within the `fn:concat` (effectively postponing evaluation of the variable to the SPARQL engine). On the other hand, if a variable has been declared before, the XQuery variable name is inserted into the `fn:concat` function, meaning that it is evaluated and replaced by its current value when the `fn:concat` function is evaluated during the execution of the rewritten query. Furthermore, the `xsp:sparqlCall` function implements the *matching blank nodes in nested queries* feature (as described in Sect. 4.2): here, we rely on external Java code to call the ARQ API in such a way that blank node labels are preserved over consecutive SPARQL calls that use the same dataset; during query rewriting we trigger the correct matching of blank nodes in nested queries whenever we encounter a *SparqlForClause* without an explicit *DatasetClause*) as follows: the respective custom Java code is used to maintain a *stack* of the previous datasets. More specifically, we collect the blank node identifiers for each dataset created by an explicit *DatasetClause* in this stack; when a *SparqlForClause* without explicit *DatasetClause* is encountered, we take the first element of the stack as the implicit dataset for the *SparqlForClause* along with its current blank node identifiers.

ConstructClause As for the construction of RDF graphs (i.e., whenever the *ReturnClause* is a *ConstructClause*), our implementation within XQuery simply produces a string in Turtle syntax, where we need to ensure that each produced RDF triple is syntactically valid. This is implemented by means of a number of additional custom functions. First, the auxiliary function `xsp:rdfTerm`($\$VarName$), presented in Fig. 19a (Appendix B), returns the correctly formatted RDF term corresponding to the variable's value in Turtle syntax given a variable of a SPARQL result type. This is done by matching the type of the variable and adding the necessary syntactic elements for each type. Next, the `xsp:validTriple` presented in Fig. 19b (Appendix B) implements the semantics function *fs*:*validTriple* by calling the `xsp:rdfTerm` function to correctly format triples to text (using the Turtle syntax); `xsp:validTriple` further uses the auxiliary functions `xsp:validSubject`, `xsp:validPredicate` and `xsp:validObject` that, respectively, determine, according to the RDF semantics, if their argument is a valid subject, predicate or object. Also available to our implementation are the functions `xsp:isBlank`, `xsp:isURI` and `xsp:isLiteral` which determine, respectively, if a term is a blank node, URI or literal. Our implementation of the *fs*:*skolemConstant* function consists of appending all the position variables from the static context (that are stored in the statEnv.posVars component) to the respective blank node identifier using "_" as a separator, represented here by the rewriting function

$$tr_{sk}(\$BNodeName, \{\$PosVar_1, \cdots, \$PosVar_n\}) = \\ \texttt{fn:concat}\begin{pmatrix} \texttt{"\_:"}, \$BNodeName, \texttt{"\_"}, \$PosVar_1, \cdots, \\ \texttt{"\_"}, \$PosVar_n \end{pmatrix}.$$

Finally, the function `xsp:evalCT` (without details) implements *fs:evalCT* by simply concatenating all the triples generated by the `xsp:validTriple` function to a string representation of the RDF graph to be constructed in Turtle syntax.

The next lemma states that the results of the evaluation of a Basic Graph Pattern P under XSPARQL BGP matching semantics can be determined based on the results of evaluating $\mu(P)$ under SPARQL semantics. The proofs for the following proposition and lemma are included in Appendices C.4 and C.5.

Lemma 2 *Let P be a BGP, D a dataset, and μ the XSPARQL instance mapping of P. Considering $P' = \mu(P)$, we have that $eval_{xs}(D, P, \mu) = eval(D, P') \bowtie \{\mu\}$.*

The following result presents the equivalence of our implementation function *tr* and the XSPARQL semantics.[14]

Proposition 3 *Let Q be a SparqlForClauseof form (Q1) and dynEnv the dynamic environment of Q, then dynEnv $\vdash Q \Rightarrow$ Val if and only if dynEnv $\vdash tr(Q) \Rightarrow$ Val.*

6 Towards Optimisations of Nested `for` Expressions

In this section we present different rewriting strategies for XSPARQL queries containing nested expressions. We are specifically interested in nested expressions with an inner *SparqlForClause*, as the number of interleaved calls to the SPARQL engine can be reduced drastically using these rewritings. As the cost inherent with XQuery `for` clauses is much lower, we do not present different rewritings for nested expressions where the inner expression is an XQuery `for`. These different rewritings proposed in this section constitute the initial step towards defining a set of optimisations for the current implementation of the XSPARQL language.

We start by presenting the definitions and conditions under which we can perform these rewritings.

Definition 12 (*Dependent Join*) We call two nested XSPARQL `for` expressions (*ForClause* or *SparqlForClause*), where the inner expression is a *SparqlForClause* and at least one variable in the inner expression is bound by the outer expression, a *dependent join*. The shared variables between the `for` expressions are called *dependent variables*.

[14] Please note that, for presentation purposes, we are omitting the initial *empty line* in case the proof trees require no premises.

Note that the strategies presented here are only applicable for dependent joins satisfying the following restrictions:

1. An explicit *DatasetClause* of the inner query needs to be statically determined i.e., it cannot be determined based on variables bound from the outer expression;
2. The return clause of the inner expression can not be a *ConstructClause*; and
3. The dependent variable in the inner query's graph pattern must be *strictly bounded* as defined next.

Definition 13 (*Strict Boundedness*) The set of *strictly bound variables* in a graph pattern P, denoted $bVars(P)$, is recursively defined as follows: if P is

- a basic graph pattern, then $bVars(P) = vars(P)$;
- $(P_1\ P_2)$, then $bVars(P) = bVars(P_1) \cup bVars(P_2)$;
- $(P_1$ `optional` $P_2)$, then $bVars(P) = bVars(P_1)$;
- $(P_1$ `union` $P_2)$, then $bVars(P) = bVars(P_1) \cap bVars(P_2)$;
- (`graph` $i\ P_1$), then $bVars(P) = bVars(P_1) \cup (\{i\} \cap \mathbb{V})$; and
- $(P_1$ `filter` $R)$, then $bVars(P) = bVars(P_1)$.

Informally, the dependent variables must occur (i) in a basic graph pattern, (ii) in every alternative of `union`s pattern, and (iii) also outside of the optional graph pattern in case of optionals. Strict boundedness essentially ensures that the join variable does not occur only in a `filter` expression, which would lead to problems in case the inner expression is called unconstrained, see below.

The rewritings introduced for the implementation of dependent joins can be grouped into two categories, depending whether the join is performed in XQuery or SPARQL. For performing the join in XQuery, we use already known join algorithms from relational databases, namely nested-loop joins or sort-merge joins. For performing the join in SPARQL, if the outer expression is a *SparqlForClause* we can implement the join by rewriting both the inner and the outer expressions into a single SPARQL call. In case the outer query consists of an XQuery *ForClause*, we can still consider this approach, but we need to convert the result of the outer XQuery *ForClause* to an RDF graph, for instance relying on a SPARQL engine that supports SPARQL Update [32] to add this temporary graph to a triple store. The proofs for the propositions presented in this section are included in Appendices C.6–C.8.

6.1 Dependent Join implementation in XQuery

The intuitive idea with these rewritings is, instead of using the naïve rewriting that performs one SPARQL query for each iteration of the outer expression, to execute only one unconstrained SPARQL query, before the outer query. The resulting set of SPARQL solution mappings is then joined in

XQuery with the results of the outer expression, using one of the following strategies.

The straightforward way to implement the join over dependent variables directly in XQuery is by nesting two XQuery `for` expressions, much like a regular nested-loop join [1] in standard relational databases. In our proposed rewriting, the join consists of restricting the values of variables from the inner expression to the values taken from the current iteration of the outer expression, which does not require an incremental solution construction step. The approach for query rewriting applied to XSPARQL is similar to already known optimisations from the relational databases realm and also presented for XQuery queries by [47].

Similarly to Sect. 5, we will describe the implementation of this nested-loop join by means of the rewriting function opt_{nl}. We use $A \triangle B = (A \cup B) \setminus (A \cap B)$ to denote the symmetric difference of two sets A and B.

Let Q be an XSPARQL expression of form

```
(1) for $Var^{out}$ at $PosVar^{out}$ in $ExprSingle_1$ return
(2)   for $Vars^{in}$ at $PosVar^{in}$
        $DatasetClause\ WhereClause\ SolutionModifier$          (Q2)
(3)   return $ExprSingle_2$
```

the application of the rewriting function $opt_{nl}(Q)$ can be split into two cases:

– if $ExprSingle_1$ and $ExprSingle_2$ do not contain any occurrences of (Q2) then, assuming $Vars^{sp}$=$Vars(WhereClause)$, we have that

```
$opt_{nl}(Q) =$
(1) let $xsp:results :=
      xsp:sparqlCall $\left(\begin{array}{l}\texttt{select}\ \{\$Var^{out}\} \cup Vars^{in}\\ DatasetClause\\ WhereClause\\ SolutionModifier\end{array}\right)$
      return
(2) for $Var^{out}$ at $PosVar^{out}$ in $ExprSingle_1$ return
(3)   for $xsp:result at $PosVar^{in}$
        in $xsp:results//sr:result return
(4)   if $\left(join_{nl}\left(\begin{array}{l}\{\$Var^{out}\} \cap Vars^{sp},\\ \texttt{\$xsp:result}\end{array}\right)\right)$ then
(5)   let $v :=                          for each $\$v \in \{Var^{out}\} \triangle Vars^{sp}$
        $xsp:result/sr:binding[@name = $v$]/* return
(6)   $ExprSingle_2$
(7)   else ()
```

Note that here we slightly abuse notation, using '∪' to denote the concatenation of two lists of variables. The auxiliary function $join_{nl}$ aggregates the actual join-comparison in an XPath expression, where two variables are considered compatible if the outer value is a blank node, their values are equal, or the inner value ($\$VarRes_i$) is unbound:

$$join_{nl}(\{\$Var_1, \cdots, \$Var_n\}, \$res) =$$
$$\left(\begin{array}{l}\texttt{xsp:isBlank}(\$Var_1)\ \texttt{or}\\ \texttt{fn:empty}(\$res\texttt{/sr:binding[@name} = Var_1\texttt{]/*})\ \texttt{or}\\ (\$Var_1\ \texttt{eq}\ \$res\texttt{/sr:binding[@name} = Var_1\texttt{]/*})\end{array}\right)$$
$$\texttt{and}\cdots$$
$$\texttt{and}\left(\begin{array}{l}\texttt{xsp:isBlank}(\$Var_n)\ \texttt{or}\\ \texttt{fn:empty}(\$res\texttt{/sr:binding[@name} = Var_n\texttt{]/*})\\ \texttt{or}\\ (\$Var_n\ \texttt{eq}\ \$res\texttt{/sr:binding[@name} = Var_n\texttt{]/*})\end{array}\right)$$

– otherwise:

$$opt_{nl}(Q) = opt_{nl}\left(\begin{array}{l}\texttt{for}\ \$Var^{out}\ \texttt{at}\ \$PosVar^{out}\ \texttt{in}\ opt_{nl}(ExprSingle_1)\\ \texttt{return}\\ \quad\texttt{for}\ Vars^{in}\ \texttt{at}\ \$PosVar^{in}\\ \quad\quad DatasetClause\ WhereClause\ SolutionModifier\\ \quad\texttt{return}\ opt_{nl}(ExprSingle_2)\end{array}\right)$$

When Q is an XSPARQL expression of form

```
(1) for $Vars^{out}$ at $PosVar^{out}$ $DatasetClause^{out}$
(2) $WhereClause^{out}$ $SolutionModifier^{out}$
(3) return                                                    (Q3)
(4)   for $Vars^{in}$ at $PosVar^{in}$ $DatasetClause^{in}$
(5)   $WhereClause^{in}$ $SolutionModifier^{in}$
(6)   return $ExprSingle$
```

the application of the rewriting function $opt_{nl}(Q)$ can be split into two cases:

– in case $ExprSingle$ does not contain any occurrences of (Q3) then, considering $Vars^{sp} = vars(WhereClause^{in})$ being the set of variables from the inner $WhereClause$, we have that

```
$opt_{nl}(Q) =$
(1) let $xsp:res_in :=
      xsp:sparqlCall $\left(\begin{array}{l}\texttt{select}\\ Vars^{in} \cup Vars^{out} \cap Vars^{sp}\\ DatasetClause^{in}\\ WhereClause^{in}\\ SolutionModifier^{in}\end{array}\right)$
      return
(2) let $xsp:res_out :=
      xsp:sparqlCall $\left(\begin{array}{l}\texttt{select}\ Vars^{out}\\ DatasetClause^{out}\\ WhereClause^{out}\\ SolutionModifier^{out}\end{array}\right)$
      return
(3) for $xsp:rout at $PosVar^{out}$
      in $xsp:res_out//sr:result return
(4)   let $v :=                          for each $\$v \in Vars^{out}$
      $xsp:rout/sr:binding[@name = $v$]/* return
(5) for $xsp:rin at $PosVar^{out}$
      in $xsp:res_in//sr:result return
(6)   if $\left(join_{sr}\left(\begin{array}{l}Vars^{out} \cap Vars^{sp},\\ \texttt{\$xsp:res\_out},\\ \texttt{\$xsp:res\_in}\end{array}\right)\right)$ then
(7)     let $v :=                        for each $\$v \in Vars^{out} \triangle Vars^{sp}$
        $xsp:res_in/sr:binding[@name = $v$]/* return
(8)     $ExprSingle$
(9)   else ()
```

where the $join_{sr}$ function is defined as

$$
\begin{aligned}
&join_{sr}(\{\$Var_1, \cdots, \$Var_n\}, \$resOut, \$resIn) = \\
&\quad join_{nl}(\{\$resOut/\texttt{sr:binding[@name} = Var_1\texttt{]/*}\}, \$resIn) \\
&\quad \texttt{and} \cdots \texttt{and} \\
&\quad join_{nl}(\{\$resOut/\texttt{sr:binding[@name} = Var_n\texttt{]/*}\}, \$resIn)\ .
\end{aligned}
$$

– otherwise:

$$
opt_{nl}(Q) = opt_{nl}\left(\begin{array}{l}
\texttt{for } Vars^{out} \texttt{ at } \$PosVar^{out}\ DatasetClause^{out} \\
WhereClause^{out}\ SolutionModifier^{out} \\
\texttt{return} \\
\quad \texttt{for } Vars^{in} \texttt{ at } \$PosVar^{in}\ DatasetClause^{in} \\
\quad WhereClause^{in}\ SolutionModifier^{in} \\
\quad \texttt{return } opt_{nl}(ExprSingle)
\end{array}\right)
$$

This rewriting to the nested-loop join reduces the number of needed SPARQL calls from $1 + N$ (where N is the number of iterations of the outer expression) to two SPARQL calls.

Next we show that the opt_{nl} rewriting function is sound and complete.

Proposition 4 *Let Q be a XSPARQL expression of form (Q2) or (Q3) and dynEnv the dynamic environment of Q, then $dynEnv \vdash Q \Rightarrow Val$ if and only if $dynEnv \vdash opt_{nl}(Q) \Rightarrow Val$.*

6.2 Dependent Join Implementation in SPARQL

This form of rewriting of nested expressions aims at improving the runtime of the query by delegating the execution of the join to the SPARQL engine, as opposed to performing the join within XQuery only.

***SparqlForClause* within a *SparqlForClause*.** For nested expressions where both expressions consist of *SparqlForClauses* we can implement the join by rewriting the *SparqlForClauses* into a single SPARQL query. The idea here is that a join encoded as nested *SparqlForClauses* in XSPARQL can just be implemented by a SPARQL query that merges the `where` clauses of the outer and inner *SparqlForClause*. However, there are some restrictions to the applicability of this rewriting: (i) both queries must be done over the same dataset; (ii) apart from `order by`, no other solution modifiers can be used in the queries; and (iii) the original queries must not require any nesting of the XML output or use of aggregators.

As indicated before, for the next rewriting we are only allowing the `order by` solution modifier and the concatenation of "`order by` \$o1" and "`order by` \$o2" is "`order by` \$o1 \$o2". For presentation purposes, *GGP* and *OC* are, respectively, a short representation for *GroupGraphPattern* and *OrderCondition*.

For an XSPARQL query Q of form

$$
\begin{array}{ll}
(1) & \texttt{for } Vars^{out} \texttt{ at } \$PosVar^{out}\ DatasetClause \\
(2) & \texttt{where } GGP^{out} \\
(3) & \texttt{order by } OC^{out} \\
(4) & \texttt{return} \\
(5) & \quad \texttt{for } Vars^{in} \texttt{ at } \$PosVar^{in}\ DatasetClause \\
(6) & \quad \texttt{where } GGP^{in} \\
(7) & \quad \texttt{order by } OC^{in} \\
(8) & \quad \texttt{return } ExprSingle
\end{array}
\tag{Q4}
$$

then

– in case *ExprSingle* does not contain any occurrences of (Q4), we have that

$$
\begin{array}{ll}
\multicolumn{2}{l}{opt_{sr}(Q) =} \\
(1) & \texttt{let \$xsp:results :=} \\
 & \quad \texttt{xsp:sparqlCall}\left(\begin{array}{l}
\texttt{select } Vars^{out} \cup Vars^{in} \\
DatasetClause \\
\texttt{where } \{GGP^{out}\ .\ GGP^{in}\} \\
\texttt{order by } OC^{out}\ OC^{in}
\end{array}\right) \\
 & \quad \texttt{return} \\
(2) & \texttt{for \$xsp:result at } \$PosVar^{out} \\
 & \quad \texttt{in \$xsp:results//sr:result return} \\
(3) & \quad \texttt{let } \$v \texttt{ :=} \qquad \text{for each } \$v \in Vars \\
 & \quad \texttt{\$xsp:result/sr:binding[@name} = \$v\texttt{]/* return} \\
(4) & \quad ExprSingle
\end{array}
$$

Please note that the group graph patterns GGP_1 and GGP_2 include the surrounding curly braces: { and }.

– otherwise,

$$
opt_{sr}(Q) = opt_{sr}\left(\begin{array}{l}
\texttt{for } Vars^{out} \texttt{ at } \$PosVar^{out}\ DatasetClause \\
\texttt{where } GGP^{out} \\
\texttt{order by } OC^{out} \\
\texttt{return} \\
\quad \texttt{for } Vars^{in} \texttt{ at } \$PosVar^{in}\ DatasetClause \\
\quad \texttt{where } GGP^{in} \\
\quad \texttt{order by } OC^{in} \\
\quad \texttt{return } opt_{sr}(ExprSingle)
\end{array}\right)
$$

Proposition 5 *Let Q an XSPARQL expression of form (Q4) and dynEnv the dynamic environment of Q; then $dynEnv \vdash Q \Rightarrow Val$ if and only if $dynEnv \vdash opt_{sr}(Q) \Rightarrow Val$.*

***SparqlForClause* within an XQuery `for`** In case the outer expression is an XQuery `for` a similar strategy of deferring the join to a single SPARQL query is still possible. Since the optimisation proposed here does not preserve the ordering of results, it can only be applied if the order of the outer XQuery expression is not relevant. Cases where ordering can be disregarded in XQuery, as discussed by [35], include not only the `unordered` ordering mode in XQuery but also the use of aggregate functions and other built-in functions or the quantifiers `some` and `every`. This optimisation relies on first transforming the outer expressions' XML results into RDF and then joining this newly created RDF graph with the inner *SparqlForClause*'s `where` pattern in a single SPARQL query. To implement this, we can, for instance, rely on a triple store with support for named graphs to temporarily store the RDF data corresponding to the outer XQuery `for` expression's bindings for dependent variables. We can then execute a combined query with an adapted graph pattern that joins the pattern in the `where` clause of the inner *SparqlForClause* with the bindings stored in the newly created named graph. The opt_{ng} rewriting function (presented below) starts by creating RDF triples representing the XML input which are then collected into the variable `$xsp:ds` that corresponds to the RDF graph to be inserted into the triple store. This operation is achieved by the XSPARQL functions `xsp:createNG` that returns a URI for the newly inserted RDF named graph, which is distinct from any other URIs for named graphs used in the query, while finally the function `xsp:deleteNG` takes care of deleting the temporary graph. Let Q be an XSPARQL expression of form

```
(1) for $VarName OptTypeDeclaration
(2)   OptPositionalVar in ExprSingle_1
(3)   return for Vars DatasetClause WhereClause           (Q5)
(4)     SolutionModifier return ExprSingle_2
```

then,

– in case *ExprSingle₁* and *ExprSingle₂* do not contain any occurrences of (Q5), we have that

```
opt_ng(Q) =
(1)  let $xsp:ds :=
       xsp:createNG ( for $VarName OptTypeDeclaration
                      OptPositionalVar in ExprSingle_1
                      return xsp:evalCT(NGP) )
       return
(2)  let $xsp:results :=
       xsp:sparqlCall ( select Vars ∪ {$VarName}
                        DatasetClause ∪
                          {from named $xsp:ds}
                        WhereClause ∪
                          where {graph $xsp:ds
                        NGP}
                        SolutionModifier )
       return
(3)  for $xsp:result at $xsp:result_pos
        in $xsp:results//sr:result return
(4)    let $v := for each $v ∈ Vars ∪ {$VarName}
        $xsp:result/sr:binding[@name = $v]/*
(5)    return (ExprSingle_2, xsp:deleteNG($xsp:ds))
```

where *NGP* is the graph pattern $\{[]$ `:value` $\$VarName\}$.

– otherwise,

```
opt_ng(Q) =
  opt_ng ( for $VarName OptTypeDeclaration
             OptPositionalVar in opt_ng(ExprSingle_1)
             return for Vars DatasetClause WhereClause
               SolutionModifier return opt_ng(ExprSingle_2) )
```

Proposition 6 *Let Q be an XSPARQL expression of form (Q5) and dynEnv the dynamic environment of Q, then dynEnv $\vdash Q \Rightarrow Val$ if and only if dynEnv $\vdash opt_{ng}(Q) \Rightarrow Val$.*

7 Experimental Evaluation

In this section we present an experimental evaluation of our prototype presented in Sect. 5 using a novel benchmark suite, called XMarkRDF, that is based on the well-known XMark benchmark suite for XQuery. We compare our XSPARQL prototype with the SPARQL2XQuery engine, an implementation of the direct translation of SPARQL to XQuery presented by [33] and test—where possible—the effects of the optimisations presented in Sect. 6.

A detailed description of the XMarkRDF benchmark suite is included in Appendix D. We denote the 20 original XMark queries as q_1–q_{20} and the variants of the nested queries to which we apply our different rewritings as q'_8–q'_{11} and q''_8–q''_{11}. Further details regarding these queries are included in Appendix D.

We would like point the reader to available benchmark results for XQuery[15] and SPARQL[16] which present better results than our XSPARQL implementation benchmarked in this section. However, the comparison with such native SPARQL and XQuery engines is beyond scope of the paper since we specifically address a combined use case, where components from both XQuery and SPARQL are needed. The benchmark queries presented here cannot be comparably solved by relying on a single SPARQL or XQuery engine.

7.1 Experimental Setup

Using the data generators and translators, provided by the XMark benchmark and the XSPARQL translation to RDF (as presented in Sect. 6), we created datasets with scaling factors of 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, and 1.0 and translated them into XMarkRDF. An overview of the generated data is presented in Appendix D, including dataset sizes and, for each of the dataset size considered, the number of persons and item categories modelled.

Furthermore, we converted the XMarkRDF datasets into the RDF/XML format required by the SPARQL2XQuery system. The resulting dataset sizes and translation times for

[15] Benchmark results for the XMark dataset can be found at http://www.monetdb.org/XQuery/Benchmark/XMark/ and http://www.informatik.uni-freiburg.de/~mschmidt/smp/xmark.html, retrieved 20-04-2012.

[16] Benchmark evaluation of RDF stores can be found at http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/V6/index.html, retrieved 20-04-2012.

Table 1 Query response times (in seconds) of the 2MB dataset. Query rewriting error (*err*)

	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
XS	9.25	**10.65**	**10.43**	9.43	10.15	11.38	**11.97**	358.27	355.71	**35.89**
$S2XQ$	**2.63**	19.47	*err*	**3.71**	**2.82**	**2.58**	*err*	**3.44**	**18.91**	178.71
	q_{11}	q_{12}	q_{13}	q_{14}	q_{15}	q_{16}	q_{17}	q_{18}	q_{19}	q_{20}
XS	371.46	**81.96**	10.83	10.04	11.61	11.66	10.14	10.93	10.73	19.93
$S2XQ$	**17.99**	128.89	**3.93**	**2.72**	**3.00**	**3.12**	**7.58**	**10.92**	**3.05**	**16.64**

the different scaling factors of the XMarkRDF dataset are also presented in Appendix D.

The benchmark system consists of a dual-core AMD Opteron 250 2.4GHz, 4GB memory running a 64-bit installation of Ubuntu 10.04.1 LTS. For the XQuery engine, we rely on Saxon version 9.3 Enterprise Edition and Java version 1.6.0 64 bit. For evaluating SPARQL queries we used ARQ 2.8.7. We ran each query with a timeout of 10 min per query, with the Java Heap size set to 1GB and the Saxon configuration set as schema-unaware. The response time of each query was measured using GNU time 1.7 and the process startup time was deduced to each response time. For the evaluation we defined the following run configurations:

XS using the XSPARQL implementation over the XMarkRDF datasets (translated data and queries) without optimisation;
XS$_Z$ using the XSPARQL implementation over the XMarkRDF datasets (translated data and queries) with nested expression optimisation opt_Z for $Z \in \{nl, sr, ng\}$;
S2XQ using the SPARQL2XQuery implementation over the translation of the XMarkRDF datasets into the required XML format (XMarkRDF$_{S2XQ}$) without optimisation; and
S2XQ$_Z$ using the SPARQL2XQuery implementation over the translation of the XMarkRDF datasets into the required XML format (XMarkRDF$_{S2XQ}$) with nested expression optimisation opt_Z for $Z \in \{nl, sr\}$.

7.2 Results and Interpretation

The response times of the *XS* and *S2XQ* runs for the benchmark queries over the 2MB dataset size are shown in Table 1. We present the 2MB dataset as it is the largest dataset our unoptimised implementation can process within the time limit of 10 min. Both the data and query translation times are not measured in our benchmarks since this process can be done a priori. The response times for the XMark queries evaluated using the Saxon XQuery engine are not presented in this table since these queries do not cater for our heterogenous data sources scenario.

The comparison of the response times of the different rewriting functions presented in Sect. 6 is shown graphically in Figs. 15 and 16. The response times of these queries for the 2MB are presented in Table 2 as a reference, where *n/a* indicates that the combination of query and optimisation is not applicable.

We next present the interpretation of the benchmark results when comparing to the SPARQL2XQuery system and then proceed to then interpretation of the results from the different rewriting strategies.

Evaluation of *XS* and *S2XQ* without optimisation Table 1 shows that for most of the queries the *S2XQ* runs are faster than the interleaved calls to a SPARQL engine in the *XS* runs. Even considering that the response times do not include the data translation times, this suggests that an alternative implementation of XSPARQL where the SPARQL queries are translated into native XQuery is a viable alternative to interleaving calls to a SPARQL engine. However, for such translations to be possible we need access to the full RDF dataset to perform the query translation which is not possible for example in the case where we are querying data behind a SPARQL endpoint. Another issue related to the implementation of the SPARQL2XQuery system is that response times deteriorate considerably for larger datasets. This was observed for all the queries in the benchmark and can be seen in the graphs of Figs. 15 and 16.

Queries q_8–q_{12} have the highest execution times of all the benchmark queries since they contain nested expressions (as can be seen in q_9 presented in Fig. 20). For these nested queries, our interleaved XSPARQL implementation can only handle small datasets: the 2MB dataset is the largest for which all queries finish within the time limit and for the 20MB dataset all queries result in a *timeout*. For these nested queries we applied the different optimisations described in Sect. 6 and next we present their benchmark evaluation.

Evaluation of *XS* and *S2XQ* with Nested Expression Optimisation. As we can see from Table 2 and Figs. 15 and 16, the opt_{nl} optimisation provides significant reduction in the query evaluation times. For queries q_8, q_9, and q_{11} the difference in response times is one order of magnitude.

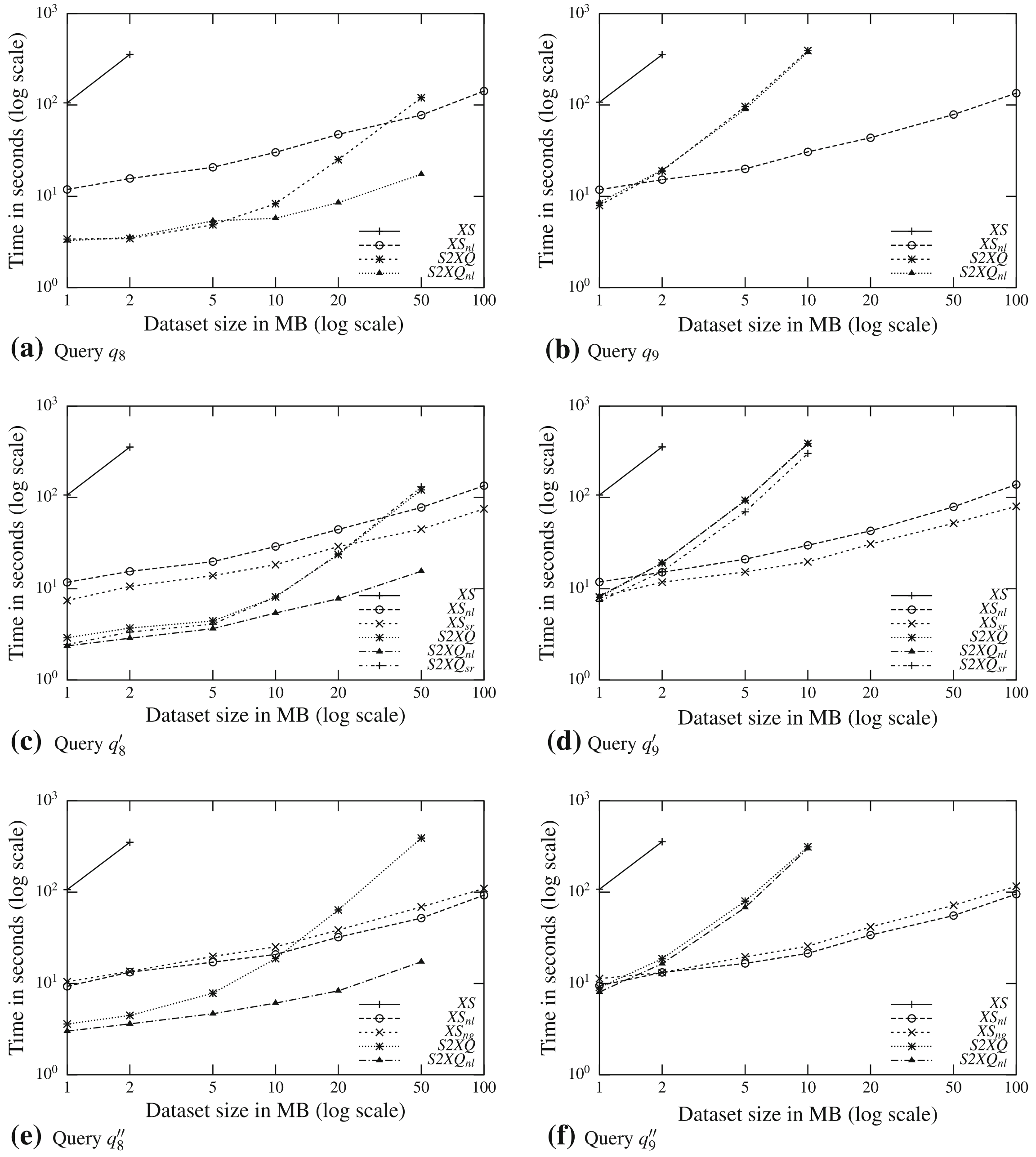
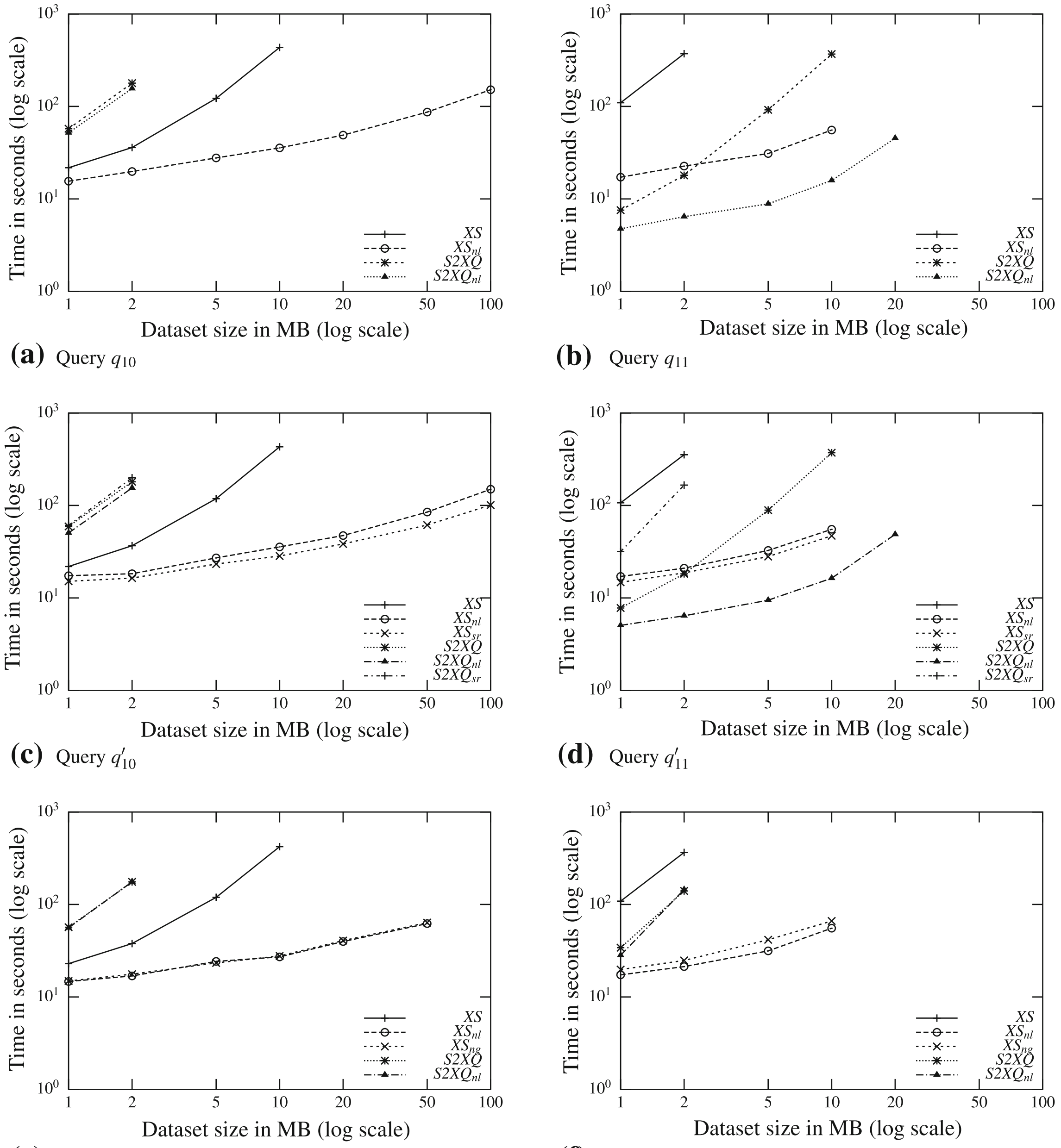


Fig. 15 Query response times for (variants of) q_8 and q_9 on all XMarkRDF datasets

The improvement in the execution time for query q_{10} is less drastic. This can be explained by the fact that the outer expression of q_{10} iterates over "categories" which increases at a much smaller rate than "persons" do in the outer expressions of queries q_8, q_9, and q_{11} (cf. Appendix D).

However, for the *S2XQ* runs this optimisation provides virtually no improvement in the query response times for queries q_8 and q_9 and their variants. In queries q_{10}, q_{11}, q_{10}', and q_{11}' we can observe an improvement in response times. This can be attributed to the fact that the rewriting for queries q_{10} and q_{11} and their variants are not as suitable for optimisation by the XQuery engine when compared with queries q_8 and q_9. For these cases our rewriting strategy is capable of performing the optimisation task for the XQuery engine.

For the *XS* run, it is possible to see in Fig. 15c, d that opt_{sr} (presented in Sect. 6.2) is generally more efficient in terms of response times than the XQuery based. This can be justified by the smaller amount of information that is necessary to transfer from SPARQL to the XQuery engine. This effec-



(a) Query q_{10} **(b)** Query q_{11} **(c)** Query q'_{10} **(d)** Query q'_{11} **(e)** Query q''_{10} **(f)** Query q''_{11}

Fig. 16 Query response times for (variants of) q_{10} and q_{11} on all XMarkRDF datasets

tively reduces the overhead of using an external SPARQL engine for the evaluation of queries. Considering the *S2XQ*$_{sr}$ run, *opt*$_{sr}$ produces no improvement in the query response times and in some cases (q'_{10} and q'_{11} from Table 2) even deteriorates considerably the response times when compared with *S2XQ*. This further supports our previous claims that the XQuery engine is not capable of optimising the rewritten code from complex SPARQL queries.

Furthermore, the *S2XQ*$_{sr}$ runs could not evaluate the higher dataset sizes for query q_8, whose response times deteriorate considerably with the larger dataset sizes—as opposed to the *XS*$_{sr}$ runs which behaves consistently similar to *XS*$_{nl}$. This indicates that *S2XQ* is not as efficient as the ARQ-based native SPARQL engine runs *XS*$_{sr}$ and *XS*$_{nl}$ for larger datasets.

We can draw similar conclusions for the *opt*$_{ng}$ when comparing the query evaluation times of the *opt*$_{sr}$ rewriting. However, the response times for this approach are deteriorated by the overhead of creating, inserting, and deleting the RDF Named Graph. This slowdown makes queries q''_8, q''_{10} and q''_{11} of the of the *opt*$_{nl}$ rewriting outperform this optimisation.

Table 2 Query response times in seconds of different optimisations for the 2MB XMarkRDF dataset. Optimisation not applicable (*n/a*)

	XS	$S2XQ$	XS_{nl}	$S2XQ_{nl}$	XS_{sr}	$S2XQ_{sr}$	XS_{ng}
q_8	358.27	**3.44**	15.66	3.54	*n/a*	*n/a*	*n/a*
q_9	355.71	18.91	**15.20**	19.35	*n/a*	*n/a*	*n/a*
q_{10}	35.89	178.71	**19.78**	156.09	*n/a*	*n/a*	*n/a*
q_{11}	371.46	17.99	22.67	**6.43**	*n/a*	*n/a*	*n/a*
q'_8	355.63	3.71	15.48	**2.87**	10.60	3.35	*n/a*
q'_9	357.13	19.16	15.12	19.10	**11.79**	15.44	*n/a*
q'_{10}	36.73	180.32	18.24	154.95	**16.37**	199.28	*n/a*
q'_{11}	354.20	18.21	21.00	**6.40**	18.63	165.99	*n/a*
q''_8	352.10	4.46	13.29	**3.61**	*n/a*	*n/a*	13.53
q''_9	356.63	18.64	13.23	16.46	*n/a*	*n/a*	**13.17**
q''_{10}	37.84	175.70	**16.88**	175.47	*n/a*	*n/a*	17.62
q''_{11}	365.24	139.96	**21.27**	145.05	*n/a*	*n/a*	24.79

8 Related Work

With the establishment of XML and RDF, tools and methods were introduced that rely on existing standards for retrieving and querying both languages. Most of the existing proposals to merge XML and RDF rely on translating the data from different formats and/or translating the queries from different languages. With this in mind, we divided the proposals into two categories: (1) **Translation of data:** these tools aim at integrating the heterogeneous data by translating between different formats, usually relying on user predefined mappings. (2) **Integration of query languages:** this category of approaches (where XSPARQL is also included) considers the integration and/or expansion of query languages to allow querying different formats. Next we give a short overview of some of the tools and proposals available in each category.

Data translation The TriX format [21] consists of an alternative serialisation for RDF in XML, with the aim of being compatible with standard XML tools. It uses XSLT as an extensibility mechanism, allowing to specify syntactic extensions and defining macros. *R3X*[17] uses an RDF processor and XSLT to transform RDF data into a predictable form of RDF/XML also catering for RSS. Similarly, Grit[18] is designed to be a simplified normalisation for RDF, easier to process with XSLT than RDF/XML. Gloze [6] aims at directly interpreting an XML document as RDF data by providing transformations between XML and RDF based on the XML Schema definition. The transformation tries to map each XML element and attribute to an RDF property. The resulting transformation makes extensive use of RDF sequences to maintain the ordering from the XML structure. Droop et al. [28] translate the XML document into RDF, annotating it with necessary information to answer XPath queries. The XPath queries are, in turn, translated into SPARQL queries, and the result of the execution of the SPARQL query is then translated into a format equivalent to the result of the XPath query.

Deursen et al. [26] present an approach for the transformation between XML and RDF in an ontology-dependent manner, introducing a language that allows to convert existing XML Schema documents (and XML documents conforming to the schemas) by defining mappings relating the schema to specified ontologies. Other approaches [17,56] aim at translating an XML Schema into an equivalent OWL ontology. However, in our approach, we are focusing on translation and integration of instance data, rather than aiming at providing a semantic interpretation for XML data.

The approaches that propose a batch translation of data pose problems such as the replication of data and the need for constant synchronisation between the original data and the transformed data, for instance in the case of a frequently updated database. We argue that this approach is not optimal for most enterprise and Web scenarios and dynamic translations are the best way to describe and implement such integration of data.

Last, but not least, as we have also discussed XSPARQL as a means to transform between different RDF representations beyond the capabilities of SPARQL [53] in this paper, we should mention the forthcoming SPARQL 1.1 [38] specification, that will add features to SPARQL addressing such use cases (aggregation, value generation, etc.). Whereas no detailed studies of SPARQL 1.1's expressivity exist as of yet, we emphasise that XSPARQL—being a Turing-complete scripting language for RDF—will be able to encompass all features within SPARQL 1.1 and more.

Language integration Berrueta et al. [12] present a framework that allows to perform SPARQL queries from XSLT: XSLT+SPARQL. It relies on adding functions to XSLT that

[17] http://wasab.dk/morten/blog/archives/2004/05/30/transforming-rdfxml-with-xslt.

[18] http://code.google.com/p/oort/wiki/Grit.

provide the ability to query SPARQL endpoints and uses standard XSLT to process the SPARQL XML results format. Similarly to our current implementation, they rely on a clear separation between the SPARQL query and XSLT parts of the query.

The following proposals suggest compiling a SPARQL query to XSLT/XQuery: Bikakis et al. [14] translate each SPARQL query into an XQuery using a previously defined mapping from OWL to XML Schema and [33] propose to embed SPARQL into XSLT or XQuery, presenting extensions to these languages to enable SPARQL querying where each SPARQL query is also translated into an equivalent XQuery. This language is very close to the XSPARQL language, but it does, however, require converting the RDF data to XML according to a predefined schema. Assuming the queried dataset is available this translation carries overhead into the query and in case the dataset is not available, for example due to being stored behind a SPARQL endpoint, such translation is not possible. Ding and Buxton presented an approach to translate SPARQL into XQuery at the 2011 Semantic Technology Conference.[19] This rewriting generates XQuery specifically tailored for the Marklogic Server XML database engine. On a similar approach, integrating XPath into SPARQL [25], also promises to bridge the gap between XML and RDF. This is approach is similar to XSPARQL, although the choice here was to extend the SPARQL query language. While also catering for SQL queries, [31] presents a translation of SPARQL queries into XQuery and present encouraging benchmark results. Another similar approach is presented by [48], where the authors again translate SPARQL to XQuery by relying on a normal form of RDF/XML. Also according to our benchmarks, encoding SPARQL in XQuery seems a viable option—assuming that we have access to the RDF dataset beforehand—that would allow to compile XSPARQL to pure XQuery without the use of a separate SPARQL engine.

Zhou and Wu [59] propose another approach to represent RDF data as XML trees, based on translating RDFS into an XML Schema, and then translating SPARQL queries into XPath and XQuery queries.

RDF Twig [58] suggests XSLT extension functions that provide views on the sub-trees of an RDF graph. The main idea of RDF Twig is that while RDF/XML is hard to navigate using XPath, a subtree of an RDF graph can be serialised into a more useful form of RDF/XML. *RDFXSLT*[20] provides an XSLT preprocessing stylesheet and a set of helper functions, similar to RDF Twig, yet implemented in pure XSLT 2.0, readily available for multiple platforms. The CORESE framework[21] also provides extensions of SPARQL to process XPath and XSL transformations in SPARQL queries and defines an extension to the XSLT language to allow to perform SPARQL queries.

Other approaches for querying heterogenous data were presented by Berger et al. [10]. This query language follows a different syntax than the W3C standardised SPARQL and XQuery and allows to write queries in the form of logical rules over an abstraction of the XML and RDF data models (represented as a graph).

The nSPARQL query language [50] proposes to extend SPARQL with navigational capabilities using nested regular expressions. With this addition, the language is sufficiently expressive to capture the semantics of RDFS. In addition to this, it introduces a number of graph navigation operators and adds the ability to selectively traverse the graph. This work is different than our current proposed approach for XSPARQL, but one of the possibilities for extending XSPARQL is to enable to perform XQuery enriched SPARQL queries.

Related to our nested queries optimisation, initial work has been presented by [5] over an extension of SPARQL that caters for nested queries and presented preliminary equivalences between types of nested queries with the aim of determining if query unnesting can be successfully applicable.

9 Conclusion and Future Work

In this paper we presented a novel query language, called XSPARQL, that combines XQuery and SPARQL to provide simplified transformations between the XML and RDF data models. We covered the semantics of XSPARQL, defined as an extension of the XQuery semantics, and presented our current implementation which consists of rewriting each XSPARQL query to an XQuery query. The implementation is available for download at http://xsparql.deri.org/ where we also provide an online XSPARQL query evaluator at http://xsparql.deri.org/demo/.

We also presented different rewriting strategies for a particular category of XSPARQL queries, namely those containing nested expressions involving SPARQL queries and presented benchmark evaluation of these different rewritings. For these optimisations we detailed the rewriting functions describing their application in our current implementation of the XSPARQL language. We presented two types of optimisations for nested expressions: one based on reordering the expressions in the XQuery rewriting to minimise the number of calls to the SPARQL endpoint and another based on performing a more complex SPARQL query that takes care of joining the variables. The benchmarks carried out to determine the impact of our optimisations have shown encour-

[19] http://semtech2011.semanticweb.com/sessionPop.cfm?confid=62&proposalid=4015.

[20] http://www.wsmo.org/TR/d24/d24.2/v0.1/20070412/rdfxslt.html.

[21] http://www-sop.inria.fr/edelweiss/software/corese/.

aging results, hinting on a large potential for optimisations in XSPARQL. Among the rewriting strategies presented in this paper and on our test data, pushing joins into a SPARQL engine appeared the most promising strategy. Our benchmark results showed that our optimisations are not only specific to XSPARQL having also improved the response times of the SPARQL2XQuery system to which we compared XSPARQL.

Future Work In this paper we have shown that nested queries can be efficiently evaluated by applying particular rewritings. Nonetheless all the tested rewriting strategies were created ad-hoc. A declarative algebra model would help to correctly and systematically study further optimisations for XSPARQL. As starting points, [34] and [36] have presented translations of XQuery to SQL, whereas in our own earlier works we have likewise translated SPARQL essentially to Relational Algebra [52]. These works seem to indicate valid starting points for further research on equivalences and optimisations in our language. Initial steps for defining such a declarative algebra can also be based on subsets of the XQuery language, for example XQuery core presented by Koch [43]. A proposal towards the declarative model of XSPARQL has been done by Bischof [16]. Although the initial set of optimisations proposed in this paper shows that our current implementation of performing interleaved calls to a SPARQL engine can be improved upon, a more tightly integrated implementation of the XSPARQL language should yield better results. Such an integrated implementation is planned for the near future where we can leverage optimisations proposed for SPARQL [37] or the proposed implementations of XQuery over relational databases [34, 13]. Finally, the current XSPARQL language specification already allows to query data contained in XML and RDF datastores. However, updating data of these datastores is still not directly possible. We plan to extend the XSPARQL language to a full data manipulation language allowing to update, insert, and delete data contained in RDF tripestores. Here, similar to our combination of query languages, we will aim at combining common data manipulation languages for XML and RDF, such as SPARQL Update [32] and XQuery Update [55].

Acknowledgments This work was supported by SFI grant SFI/08/-CE/I1380 (Líon-2), by an IRCSET scholarship, and by the Marie Curie IRSES Grant 24761 (Net2). We would like to thank Sven Groppe for his help with the SPARQL2XQuery system.

A Example of XSPARQL Semantics Evaluation

As an example we show the application of the XSPARQL evaluation semantics (presented in Sect. 4.2) to the sample query from Fig. 8b. The example query features both, the new *SparqlForClause* as well as the new *ConstructClause*.

```
prefix vc: <...vcard-rdf/3.0#>
prefix foaf: <...foaf/0.1/>

_:b vc:Given "Charles" .
_:b vc:Family "Brown" .
```

(a) Example input

```
declare namespace vc="...vcard-rdf/3.0#";
declare namespace foaf="...foaf/0.1/";
for $P $N $F at $pos from <vc.rdf>
where { $P vc:Given $N. $P vc:Family $F.}
return fs:evalTemplate( _:gen foaf:name
                        {fn:concat($N," ",$F)}.)
```

(b) Query after normalisation

Fig. 17 Example input and normalised query of Fig. 8

Table 3 Result of *fs:sparql*

$P	$N	$F
_:gen	"Charles"	"Brown"

We assume the input graph *vc.rdf* as given in Fig. 17a. Let us go through the three phases of XQuery semantics evaluation, i.e., the normalisation, static type checking, and dynamic evaluation steps.

Normalisation In the normalisation step the SPARQL-style namespace declarations are rewritten to XQuery namespace declarations (see Rule (N3)). After that, the whole `construct` query is rewritten to a *SparqlForClause*by Rule (N8), the `for *` is expanded according to Rule (N6) and the resulting *SparqlForClause*is then handled by Rule (N7). Rule $[\![\cdot]\!]_{PosVar}$ then adds a new positional variable (e.g., $*pos*). Finally the `construct` is normalised by Rule (N9). The whole normalisation phase results in the query given in Fig. 17b.

Static Type Analysis By Rule (S2) the variables occurring in the *WhereClause*, namely $*P*, $*N*, and $*F*, are typed as `RDFTerm`, and the positional variable, $*pos*, is typed as `xs:integer`. The whole *SparqlForClause* inherits its type from the contained `return` *ExprSingle* , which in turn inherits its type from the function *fs*:*evalCT* which is `RDFGraph`.

Dynamic Evaluation First the new environment component activeDataset is changed from empty to the one given in the *DatasetClause*, i.e., the graph contained in *vc.rdf*. According to Rule (D3) the *WhereClause* is evaluated using the *fs*:*sparql* function with the active dataset, as just initialised, the *WhereClause* as given in the query, and empty *SolutionModifier*s. The *fs*:*sparql* function call results in a sequence of `PatternSolutions` (in this case a singleton solution) as given in Table 3. Next, the same rule extracts the variable bindings for all variables, using the *fs*:*value* function, and assigns them to the corresponding XQuery variables, typed as `RDFTerm`. After that the return expression *ExprSingle* is evaluated, using the just initialised variables. The

```
<RDFGraph>
  <triples>
    <triple>
      <subject><bnode>_:gen_1</bnode></subject>
      <predicate><uri>foaf:name</uri></predicate
      <object><literal>Charles Brown</literal></object>
    </triple>
  </triples>
</RDFGraph>
```

Fig. 18 Query results

fs:*evalCT* function calls the *fs*:*validTriple* function passing it a blank node generated by the *fs*:*bnode* function as subject, "`foaf:name`" as predicate and the result of *fn*:*concat* as object. The *fs*:*bnode* function (as given by Rule (D7)) generates a fresh blank node label for each element of the `PatternSolution`. For this example we assume that the function returns the new blank node label "_:gen_1". The *fs*:*validTriple* function tests these three values for validity. Since the subject is of type `bnode`, the predicate is a QName (and therefore considered as being of type `uri`), and the object is of type `literal`, namely an `xs:string`, the function returns them as a valid `RDFTriple`. The *fs*:*evalCT* function eventually returns the result of the single *fs*:*validTriple* function call and thus the result of the whole query as an element of type `RDFGraph` as shown in Fig. 18. Serialised to Turtle the query result, including QName expansion, is the expression `_:gen_1 <http://...foaf/0.1/name> "Charles Brown"`.

B Implementation Functions Example

Figure 19 presents some of the XQuery functions defined in the XSPARQL language implementation, namely to correctly format RDF terms (Fig. 19a) and to validate triples resulting from a `construct` expression (Fig 19b).

C Proofs

C.1 Proof for Proposition 1

Proposition 1 *XSPARQL is a conservative extension of XQuery.*

Proof We show that the additional rules introduced in Sect. 4.2 do not modify the semantics of any native XQuery. The XSPARQL semantics—expressed in terms of normalisation rules, static typing rules, and dynamic evaluation rules—strictly extend the native semantics of XQuery. In the semantics definition we also define new environment components, namely statEnv.posVars and dynEnv.activeDataset, which are not used in the XQuery semantics and thus do not interfere with query evaluation. However, for the XSPARQL semantics we also extend the normalisation rules and static analysis rules for native XQuery `for` clauses. More specifically, rule (N5) extends the XQuery `for` normalisation by adding a new variable to each position-variable free `for` expression (i.e., that does not have an `at` clause). As stated these new position variables are disjoint from the variables in scope, and thus this rewriting does not interfere with the semantics of the original query. The only rules which use the newly created position variables are (i) the slightly modified static type analysis rule (S3) which extends the XQuery `for` static analysis rule by collecting the position variables in the static environment component statEnv.posVars, thus also maintaining the original semantics of the original XQuery `for` rule and (ii) the dynamic evaluation rule (D7) which accesses statEnv.posVars to generate Skolem-identifiers for blank nodes in `construct` parts. However, rule (D7) only applies to XSPARQL queries which fall outside the native XQuery fragment, whereas the semantics of native XQuery queries remains untouched and independent of the extra environment components in XSPARQL. □

```
declare function xsp:rdfTerm($VarName) {
  typeswitch $VarName
  case $e as literal
    let $DT := data($e/@datatype)
    let $L:= data($e/@xml:lang)
    return concat("""", $e,
          if($L) then concat("@", $L) else "",
          if($DT) then concat("^^<", $DT,">") else "",
        """")
  case $e as bnode  return concat("_:", $e)
  case $e as uri return concat("<", $e, ">")
  default return "" };
```

(a) `xsp:rdfTerm` function

```
declare function xsp:validTriple($sub, $pred, $obj) {
  if(xsp:validSubject($sub)
     and xsp:validPredicate($pred)
     and xsp:validObject($obj))
  then concat(xsp:rdfTerm($sub), " ",
              xsp:rdfTerm($pred), " ",
              xsp:rdfTerm($obj), ".")
  else "" };
```

(b) `xsp:validTriple` function

Fig. 19 Implementation functions

C.2 Proof for Lemma 1

Lemma 1 *Given a graph pattern P, a dataset D and the XSPARQL instance mapping μ_C of the expression context C over which P is evaluated, let $\Omega_1 = eval_{xs}(D, P, \mu_C)$ and $\Omega_2 = eval(D, P)$ be solution mappings. If $vars(P) \cap dom(\mu_C) = \emptyset$, then $\Omega_1 = \Omega_2 \bowtie \{\mu_C\}$.*

Proof The XSPARQL BGP matching, $eval_{xs}(D, P, \mu_C)$, extends SPARQL's BGP matching, $eval(D, P)$, by defining that the solutions of the BGP are the ones *compatible* with the *XSPARQL instance mapping* μ_C. Since the evaluation of graph patterns (such as `union`, `optional`, `graph`, and `filter`) remains unchanged from the SPARQL semantics let us focus on the evaluation of a BGP P. If there are no shared values between the graph pattern and the XSPARQL instance mapping, $vars(P) \cap dom(\mu_C) = \emptyset$, then each solution $\mu \in \Omega_2$ returned by the SPARQL BGP evaluation semantics is trivially compatible with μ_C and the result of the XSPARQL BGP matching is $\mu \cup \mu_C$. Extending this result to all solution mappings in Ω_2, we obtain that $\Omega_1 = \Omega_2 \bowtie \{\mu_C\}$. □

C.3 Proof for Proposition 2

Proposition 2 *XSPARQL is a conservative extension of SPARQL* `construct` *queries.*

Proof For XSPARQL queries consisting of a SPARQL `construct` query, there cannot exist any previous bindings for variables in XSPARQL and thus the XSPARQL instance mapping μ_C over which the `construct` query will be executed is empty. Let P represent the graph pattern of the `construct` query and D the dataset, trivially there are no shared variables between μ_C and P and so, following Lemma 1 the bindings Ω_1 for XSPARQL BGP matching are the same bindings Ω_2 as SPARQL BGP matching, since $\Omega_1 = \Omega_2 \cup \{\emptyset\}$ and hence $\Omega_1 = \Omega_2$. Furthermore, the formal semantics function *fs*:*evalTemplate* returns an RDF graph satisfying all the conditions of Definition 11: (1) Ignoring invalid RDF triples—Item 1—is guaranteed by Rules D5 and D6; and (2) The generation of distinct blank nodes for each solution sequence—Item 2—is enforced by the blank node skolemisation rules (Rules (D7) and (D8)). □

C.4 Proof for Lemma 2

Lemma 2 *Let P be a BGP, D a dataset, and μ the XSPARQL instance mapping of P. Considering $P' = \mu(P)$, we have that $eval_{xs}(D, P, \mu) = eval(D, P') \bowtie \{\mu\}$.*

Proof Since, according to the variable substitution operation we have that $vars(P') = vars(P) \setminus dom(\mu)$, we also have that $vars(P') \cap dom(\mu) = \emptyset$, and it follows directly from Lemma 1 that $eval_{xs}(D, P, \mu) = eval(D, P') \bowtie \{\mu\}$. □

C.5 Proof for Proposition 3

Proposition 3 *Let Q be a SparqlForClauseofform (Q1) and dynEnv the dynamic environment of Q, then dynEnv $\vdash Q \Rightarrow Val$ if and only if dynEnv $\vdash tr(Q) \Rightarrow Val$.*

Proof ($\Leftarrow$) Let us show that if dynEnv $\vdash tr(Q) \Rightarrow Val$ then dynEnv $\vdash Q \Rightarrow Val$. The evaluation of Q consists of the application of Rule (D1) as

$$\dfrac{\dfrac{\dfrac{\text{dynEnv} \vdash \mathit{fs}{:}\mathit{dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{DS}}{\text{dynEnv} \vdash \mathit{fs}{:}\mathit{sparql}\begin{pmatrix}\mathit{DS}, \mathit{WhereClause},\\ \mathit{SolutionModifier}\end{pmatrix} \Rightarrow \mu_i^{xs}}}{\text{dynEnv}_1^{xs} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_i} \quad \iddots}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \texttt{ at } \$\mathit{PosVar}\\ \mathit{DatasetClause}\ \mathit{WhereClause}\\ \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ExprSingle}\\ \Rightarrow \mathit{Value}_1 \cdots \mathit{Value}_m\end{array}}$$

where, for each μ_i^{xs},

$$\begin{aligned}\text{dynEnv}_1^{xs} = \text{dynEnv} \ &+ \text{activeDataset}(\mathit{DS})\\ &+ \text{varValue}\begin{pmatrix}\mathit{PosVar} \Rightarrow i;\\ \mathit{Var}_1 \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_i^{xs}, \mathit{Var}_1);\\ \cdots;\\ \mathit{Var}_n \Rightarrow \mathit{fs}{:}\mathit{value}(\mu_i^{xs}, \mathit{Var}_n)\end{pmatrix}.\end{aligned} \tag{T1}$$

Let μ_C be the XSPARQL instance mapping of the expression context that includes dynEnv and Ω_{tr} the pattern solution resulting from evaluating the `xsp:sparqlCall` function, i.e., $\Omega_{tr} = eval(DatasetClause, P)$, where P is the rewriting of *WhereClause* according to μ_C. Furthermore, let $\mu_i \in \Omega_{tr}$ be the solution mapping from which Val is generated, i.e., there exists some dynamic environment dynEnv^{tr} based on dynEnv and extended with the variable bindings from μ_i such that dynEnv^{tr} $\vdash ExprSingle \Rightarrow Val$.

Consider $\Omega_{xs} = eval_{xs}(DatasetClause, WhereClause, \mu_C)$ as the solution sequence resulting from the evaluation of the *fs*:*sparql* function. As we know from Lemma 2, $\Omega_{xs} = \Omega_{tr} \bowtie \{\mu_C\}$ and thus there must exist a solution mapping $\mu_{xs} \in \Omega_{xs}$ such that $\mu_{xs} = \mu_i \bowtie \mu_C$. From (T1) we infer that there exists a dynamic environment dynEnv^{xs} that results from extending dynEnv with the variable bindings from μ_{xs} and thus this environment will also contain all the variable mappings from dynEnv^{tr}. Since we know that dynEnv^{tr} $\vdash ExprSingle \Rightarrow Val$, we also have that dynEnv^{xs} $\vdash ExprSingle \Rightarrow Val$ and thus dynEnv $\vdash Q \Rightarrow Val$.

($\Rightarrow$) Next we will show that if dynEnv $\vdash Q \Rightarrow Val$ then dynEnv $\vdash tr(Q) \Rightarrow Val$. We present the proof tree for each of the XQuery core expressions in the $tr(Q)$ rewriting. The proof trees are presented for each line of the $tr(Q)$ rewriting and, in each proof tree, *Expr* corresponds to the XQuery expressions of the following lines:

– `let` expression of line (1):

$$\dfrac{\dfrac{\text{dynEnv} \vdash \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } \textit{Vars}\\ \textit{DatasetClause}\\ \textit{WhereClause}\\ \textit{SolutionModifier}\end{pmatrix} \Rightarrow \Omega_{tr}}{\text{dynEnv}_1^{tr} \vdash \textit{Expr} \Rightarrow \textit{Res}}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{let \$xsp:results :=}\\ \quad \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } \textit{Vars}\\ \textit{DatasetClause}\\ \textit{WhereClause}\\ \textit{SolutionModifier}\end{pmatrix}\\ \texttt{return } \textit{Expr} \Rightarrow \textit{Res}\end{array}}$$

where

$$\text{dynEnv}_1^{tr} = \text{dynEnv} \ + \text{varValue}(\texttt{xsp:results} \Rightarrow \Omega_{tr}) \quad \text{(T2)}$$

.

– `for` expression of line (2)

$$\dfrac{\dfrac{\text{dynEnv}_1^{tr} \vdash \texttt{\$xsp:results//sr:result} \Rightarrow \mu_i}{\text{dynEnv}_2^{tr} \vdash \textit{Expr} \Rightarrow \textit{Res}_i} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv}_1^{tr} \vdash \begin{array}{l}\texttt{for \$xsp:result at \$}\textit{PosVar}\\ \texttt{in \$xsp:results//sr:result}\\ \texttt{return } \textit{Expr} \Rightarrow \textit{Res}_1, \cdots, \textit{Res}_n\end{array}}$$

where

$$\text{dynEnv}_2^{tr} = \text{dynEnv}_1^{tr} \ + \text{varValue}\begin{pmatrix}\texttt{xsp:result} \Rightarrow \mu_i;\\ \textit{PosVar} \Rightarrow i\end{pmatrix}$$

– `let` expressions of lines (3)–(4)

Here we consider all the `let` expressions represented by line (3), where $\$v \in \textit{Vars}$:

$$\dfrac{\dfrac{\text{dynEnv}_2^{tr} \vdash \texttt{\$xsp:result/sr:binding[@name} = v\texttt{]/*} \Rightarrow V}{\text{dynEnv}_3^{tr} \vdash \textit{ExprSingle} \Rightarrow \textit{Res}}}{\text{dynEnv}_2^{tr} \vdash \begin{array}{l}\texttt{let \$}v \texttt{ :=}\\ \quad \texttt{\$xsp:result/sr:binding[@name} = v\texttt{]/*}\\ \texttt{return } \textit{ExprSingle} \Rightarrow \textit{Res}\end{array}}$$

where $\text{dynEnv}_3^{tr} = \text{dynEnv}_2^{tr} + \text{varValue}(v \Rightarrow V)$

Consider the dynamic environment dynEnv_i^{xs} such that $\text{dynEnv}_i^{xs} \vdash \textit{ExprSingle} \Rightarrow \textit{Val}$ where, as we know from (T1), dynEnv_i^{xs} extends dynEnv by changing the activeDataset and varValue environment components.

Consider μ_C, Ω_{xs}, and Ω_{tr} as before. From Lemma 2 we get that $\Omega_{xs} = \Omega_{tr} \bowtie \{\mu_C\}$ and since μ_C is created based on dynEnv.varValue, all the variable bindings from μ_C are already included in dynEnv.

From the proof trees of $tr(Q)$ we can see that the `for` expression from line (2) iterates over the all the solution mappings included in Ω_{tr} and the `let` expressions from lines (3) and (4) ensure there exists a dynEnv_2^{tr} such that dynEnv_2^{tr}.varValue contains all the variable bindings from dynEnv_i^{xs}.varValue, and we have that $\text{dynEnv}_2^{tr} \vdash \textit{ExprSingle} \Rightarrow \textit{Val}$. □

C.6 Proof for Proposition 4

Proposition 4 *Let Q be a XSPARQL expression of form (Q2) or (Q3) and dynEnv the dynamic environment of Q, then dynEnv ⊢ Q ⇒ Val if and only if dynEnv ⊢ $opt_{nl}(Q)$ ⇒ Val.*

Proof We now present the proof of the opt_{nl} rewriting function for expressions of the form (Q3).

We start by showing the proof for the base case, where *ExprSingle* of (Q3) does not contain any occurrences of (Q3).

Base Case. (⇒) We start by showing that if $\text{dynEnv} \vdash Q \Rightarrow \textit{Val}$ then $\text{dynEnv} \vdash opt_{nl}(Q) \Rightarrow \textit{Val}$. Consider Ω_{xs}^{out} and Ω_{xs}^{in} the solution sequences returned, respectively, by the evaluation of the outer and inner *SparqlForClause*s of Q and the set of join variables $J = \textit{Vars}^{out} \cap \textit{vars}(\textit{WhereClause}^{in})$. Furthermore, consider $\mu_{xs}^{out} \in \Omega_{xs}^{out}$ and $\mu_{xs}^{in} \in \Omega_{xs}^{in}$ the solution mappings that agree on the value of each join variable $j \in J$ from where *Val* is generated, i.e., there exists some dynamic environment dynEnv^{xs} based on dynEnv and extended with the variable mappings from μ_{xs}^{out} and μ_{xs}^{in} such that $\text{dynEnv}^{xs} \vdash \textit{ExprSingle} \Rightarrow \textit{Val}$.

We show now the proof tree for each of the XQuery core expressions in the $opt_{nl}(Q)$ rewriting where, in each proof tree, *Expr* corresponds to the XQuery expressions of the following lines:

– `let` expression of line (1), considering $\textit{Vars} = \textit{Vars}^{in} \cup (\textit{Vars}^{out} \cap \textit{vars}(\textit{WhereClause}^{in}))$, we have that

$$\dfrac{\dfrac{\text{dynEnv} \vdash \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } \textit{Vars}\\ \textit{DatasetClause}^{in}\\ \textit{WhereClause}^{in}\\ \textit{SolutionModifier}^{in}\end{pmatrix} \Rightarrow \Omega^{in}}{\text{dynEnv}_1^{nl} \vdash \textit{Expr} \Rightarrow \textit{Res}}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{let \$xsp:res\_in :=}\\ \quad \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } \textit{Vars}\\ \textit{DatasetClause}^{in}\\ \textit{WhereClause}^{in}\\ \textit{SolutionModifier}^{in}\end{pmatrix}\\ \texttt{return } \textit{Expr} \Rightarrow \textit{Res}\end{array}}$$

where

$$\text{dynEnv}_1^{nl} = \text{dynEnv} \ + \text{varValue}(\texttt{xsp:res\_in} \Rightarrow \Omega^{in}) \ . \quad \text{(T3)}$$

The function $opt_{nl}(Q)$ translates the *SparqlForClause* from lines (4)–(6) of Q into the `xsp:sparqlCall` of line (1). The inner *SparqlForClause* of Q is evaluated considering some dynamic environment dynEnv_i^{xs} (and its expression context C_i). Since dynEnv_i^{xs} is an extension of dynEnv we have that $dom(\mu_C) \subseteq dom(\mu_{C_i})$. The rewritten `xsp:sparqlCall` function is evaluated over the dynamic environment dynEnv (included in expression context C). Consider μ_C the XSPARQL instance mapping of C and μ_{C_i} the XSPARQL instance mapping of C_i. Let $\Omega_{xs}^{in} = eval_{xs}(\textit{DatasetClause}^{in}, \textit{WhereClause}^{in}, \mu_{C_i})$ be the solution sequence resulting from the evaluation of the inner *SparqlForClause* of Q and the solution sequence resulting from evaluating the `xsp:sparqlCall` function be $\Omega_{nl}^{in} = eval(\textit{DatasetClause}^{in}, P^{in})$, where P^{in}

is the graph pattern obtained from replacing the variables in $WhereClause^{in}$ according to μ_C. As $dom(\mu_C) \subseteq dom(\mu_{C_i})$, i.e., μ_C contains less bindings for variables than μ_{C_i}, the rewritten graph pattern P_{in} contains more unbound variables, and we get that $\Omega^{in}_{xs} \preceq \Omega^{in}_{nl}$.

– `let` expression of line (2)

$$\frac{\dfrac{\text{dynEnv}^{nl}_1 \vdash \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } Vars^{out} \\ DatasetClause^{out} \\ WhereClause^{out} \\ SolutionModifier^{out}\end{pmatrix} \Rightarrow \Omega^{out}}{\text{dynEnv}^{nl}_2 \vdash Expr \Rightarrow Res}}{\text{dynEnv}^{nl}_1 \vdash \begin{array}{l}\texttt{let \$xsp:res\_out :=} \\ \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } Vars^{out} \\ DatasetClause^{out} \\ WhereClause^{out} \\ SolutionModifier^{out}\end{pmatrix} \\ \texttt{return } Expr \Rightarrow Res\end{array}}$$

where

$$\text{dynEnv}^{nl}_2 = \text{dynEnv}^{nl}_1 + \text{varValue}(\texttt{xsp:res\_out} \Rightarrow \Omega^{out}) \ .$$

Regarding the *SparqlForClause* of lines (1)–(3) of Q (evaluated considering dynEnv), the $opt_{nl}(Q)$ translates it into the `xsp:sparqlCall` from line (2), which is evaluated over dynEnv^{nl}_1.
Consider C_1 the expression context where dynEnv^{nl}_1 is included, μ_{C_1} the XSPARQL instance mapping of C_1 and P^{out} the graph pattern obtained from replacing the variables in $WhereClause^{out}$ according to μ_{C_1}. From (T3) we can see that $dom(\mu_{C_1}) = dom(\mu_C) \cup \{\texttt{\$xsp:res\_in}\}$, but `$xsp:res_in` belongs to the `$xsp:` reserved namespace so it will not be included in the variables of $WhereClause^{out}$, and we can observe that we obtain the same graph pattern P^{out} by replacing $WhereClause^{out}$ according to μ_C. Let $\Omega^{out}_{xs} = eval_{xs}(DatasetClause^{out}, WhereClause^{out}, \mu_C)$ be the solution sequence resulting from evaluating the outer *SparqlForClause* according to XSPARQL semantics and $\Omega^{out}_{nl} = eval(DatasetClause^{out}, P^{out})$ be the pattern solution resulting from evaluating the rewritten outer *SparqlForClause* according to SPARQL semantics. Following Lemma 2, we have that $\Omega^{out}_{xs} = \Omega^{out}_{nl} \bowtie \{\mu_C\}$ and, as we have seen from the proof of Proposition 3, since μ_C is already included in dynEnv, we have that $\Omega^{out}_{xs} = \Omega^{out}_{nl}$.

– `for` expression of line (3)

$$\frac{\dfrac{\text{dynEnv}^{nl}_2 \vdash \texttt{\$xsp:res\_out//sr:result} \Rightarrow \mu_i}{\text{dynEnv}^{nl}_3 \vdash Expr \Rightarrow Res_i} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv}^{nl}_2 \vdash \begin{array}{l}\texttt{for \$xsp:rout at \$}PosVar^{out} \\ \texttt{in \$xsp:res\_out//sr:result} \\ \texttt{return } Expr \Rightarrow Res_1, \cdots, Res_n\end{array}}$$

where

$$\text{dynEnv}^{nl}_3 = \text{dynEnv}^{nl}_2 + \text{varValue}\begin{pmatrix}\texttt{xsp:rout} \Rightarrow \mu_i; \\ PosVar^{out} \Rightarrow i\end{pmatrix} \ .$$

– `let` expressions of line (4)
Here we consider all the `let` expressions represented by line (4), where $\$v \in Vars^{out}$:

$$\frac{\dfrac{\text{dynEnv}^{nl}_3 \vdash \texttt{\$xsp:rout/sr:binding[@name} = v\texttt{]/*} \Rightarrow V}{\text{dynEnv}^{nl}_4 \vdash Expr \Rightarrow Res}}{\text{dynEnv}^{nl}_3 \vdash \begin{array}{l}\texttt{let \$}v\texttt{ :=} \\ \quad \texttt{\$xsp:rout/sr:binding[@name} = v\texttt{]/*} \\ \texttt{return } Expr \Rightarrow Res\end{array}}$$

where

$$\text{dynEnv}^{nl}_4 = \text{dynEnv}^{nl}_3 + \text{varValue}(v \Rightarrow V) \ .$$

– `for` expression of line (5)

$$\frac{\dfrac{\text{dynEnv}^{nl}_4 \vdash \texttt{\$xsp:res\_in//sr:result} \Rightarrow S_i}{\text{dynEnv}^{nl}_5 \vdash Expr \Rightarrow Res_i} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv}^{nl}_4 \vdash \begin{array}{l}\texttt{for \$xsp:rin at \$}PosVar^{out} \\ \texttt{in \$xsp:res\_in//sr:result} \\ \texttt{return } Expr \Rightarrow Res_1, \cdots, Res_n\end{array}}$$

where

$$\text{dynEnv}^{nl}_5 = \text{dynEnv}^{nl}_4 + \text{varValue}\begin{pmatrix}\texttt{xsp:rin} \Rightarrow S_i; \\ PosVar^{in} \Rightarrow i\end{pmatrix} \ .$$

– `if` expression of lines (6)–(9)

$$\frac{\dfrac{\text{dynEnv}^{nl}_5 \vdash join_{sr}\begin{pmatrix}Vars^{out} \cap vars(WhereClause)\,, \\ \texttt{\$xsp:res\_out, \$xsp:res\_in}\end{pmatrix} \Rightarrow \texttt{true}}{\text{dynEnv}^{nl}_5 \vdash ExprSingle \Rightarrow Res_1}}{\text{dynEnv}^{nl}_5 \vdash \begin{array}{l}\texttt{if}\left(join_{sr}\begin{pmatrix}Vars^{out} \cap vars(WhereClause)\,, \\ \texttt{\$xsp:res\_out, \$xsp:res\_in}\end{pmatrix}\right) \\ \texttt{then } ExprSingle \texttt{ else () } \Rightarrow Res_1\end{array}}$$

– `let` expressions of line (7) and (8)
Here we consider all the `let` expressions represented by line (7), where $\$v \in Vars^{out} \triangle vars(WhereClause^{in})$.

$$\frac{\dfrac{\text{dynEnv}^{nl}_5 \vdash \texttt{\$xsp:res\_in/sr:binding[@name} = v\texttt{]/*} \Rightarrow V}{\text{dynEnv}^{nl}_6 \vdash ExprSingle \Rightarrow Res}}{\text{dynEnv}^{nl}_5 \vdash \begin{array}{l}\texttt{let \$}v\texttt{ :=} \\ \quad \texttt{\$xsp:res\_in/sr:binding[@name} = v\texttt{]/*} \\ \texttt{return } ExprSingle \Rightarrow Res\end{array}}$$

where

$$\text{dynEnv}^{nl}_6 = \text{dynEnv}^{nl}_5 + \text{varValue}(v \Rightarrow V) \ .$$

Since we know that $\Omega^{out}_{nl} = \Omega^{out}_{xs}$ and $\Omega^{in}_{xs} \preceq \Omega^{in}_{nl}$, we obtain that $\mu^{out}_{xs} \in \Omega^{out}_{nl}$ and $\mu^{in}_{xs} \in \Omega^{in}_{nl}$. Since $opt_{nl}(Q)$ performs a nested loop iteration over Ω^{out}_{nl} and Ω^{in}_{nl}, the $join_{sr}$ function will join the two solution mappings successfully since μ^{out}_{xs} and μ^{in}_{xs} share the same values for the join variables, and thus we have that $\text{dynEnv} \vdash opt_{nl}(Q) \Rightarrow Val$.

($\Leftarrow$) We now proceed by showing that if $\text{dynEnv} \vdash opt_{nl}(Q) \Rightarrow Val$, then $\text{dynEnv} \vdash Q \Rightarrow Val$. Let Ω^{out}_{nl} and Ω^{in}_{nl} be the pattern solutions returned by the outer and inner *SparqlForClauses*, respectively, and let $\mu^{out}_{nl} \in \Omega^{out}_{nl}$ and $\mu^{in}_{nl} \in \Omega^{in}_{nl}$ be the solution mappings, where *Val* is deduced from, i.e., μ^{out}_{nl} and μ^{in}_{nl} agree on their values for the join variables. We also know that there must exist a dynamic environment dynEnv^{nl}, based on dynEnv and extended with the variable mappings μ^{out}_{nl} and μ^{in}_{nl} such that $\text{dynEnv}^{nl} \vdash ExprSingle \Rightarrow Val$.
Let us turn to the evaluation of $\text{dynEnv} \vdash Q \Rightarrow Val$.

– *SparqlForClause* from lines (1)–(3), where *Expr* corresponds to the *SparqlForClause* from lines (4)–(6) of Q. The evaluation of this *SparqlForClause* consists of the application of Rule (D1):

$$\cfrac{\cfrac{\cfrac{\text{dynEnv} \vdash fs{:}dataset\big(DatasetClause^{out}\big) \Rightarrow DS^{out}}{\text{dynEnv} \vdash fs{:}sparql\begin{pmatrix} DS^{out},\ WhereClause, \\ SolutionModifier \end{pmatrix} \Rightarrow \mu_i}}{\text{dynEnv}^{xs}_1 \vdash Expr \Rightarrow Value_i} \quad \iddots}{\text{dynEnv} \vdash \begin{array}{l} \texttt{for } Vars^{out} \texttt{ at } \$PosVar^{out} \\ DatasetClause^{out}\ WhereClause^{out} \\ SolutionModifier^{out} \texttt{ return} \\ Expr \Rightarrow Value_1 \cdots Value_m \end{array}}$$

with $Vars^{out} = \$Var^{out}_1 \cdots \$Var^{out}_n$, we have for each μ_i

$$\text{dynEnv}^{xs}_1 = \begin{array}{l} \text{dynEnv} + \text{activeDataset}\big(DS^{out}\big) \\ + \text{varValue}\begin{pmatrix} PosVar^{out} \Rightarrow i; \\ Var^{out}_1 \Rightarrow fs{:}value\big(\mu_i, Var^{out}_1\big); \\ \ldots; \\ Var^{out}_n \Rightarrow fs{:}value\big(\mu_i, Var^{out}_n\big) \end{pmatrix} \end{array}. \quad \text{(T4)}$$

– *SparqlForClause* of lines (4)–(6):
The evaluation of $\text{dynEnv}^{xs}_1 \vdash Expr \Rightarrow Value_i$ is given by

$$\cfrac{\cfrac{\cfrac{\text{dynEnv}^{xs}_1 \vdash fs{:}dataset\big(DatasetClause^{in}\big) \Rightarrow DS^{in}}{\text{dynEnv}^{xs}_1 \vdash fs{:}sparql\begin{pmatrix} DS^{in}, WhereClause^{in}, \\ SolutionModifier^{in} \end{pmatrix} \Rightarrow \mu_j}}{\text{dynEnv}^{xs}_2 \vdash ExprSingle \Rightarrow Value_j} \quad \iddots}{\text{dynEnv}^{xs}_1 \vdash \begin{array}{l} \texttt{for } Vars^{in} \texttt{ at } \$PosVar^{in} \\ DatasetClause^{in}\ WhereClause^{in} \\ SolutionModifier^{in} \texttt{ return} \\ ExprSingle \Rightarrow Value_1 \cdots Value_m \end{array}}$$

where, considering $Vars^{in} = \$Var^{in}_1 \ldots \$Var^{in}_n$, we have for each μ_j

$$\text{dynEnv}^{xs}_2 = \begin{array}{l} \text{dynEnv}^{xs}_1 + \text{activeDataset}\big(DS^{in}\big) \\ + \text{varValue}\begin{pmatrix} PosVar^{in} \Rightarrow j; \\ Var^{in}_1 \Rightarrow fs{:}value\big(\mu_j, Var^{in}_1\big); \\ \ldots; \\ Var^{in}_n \Rightarrow fs{:}value\big(\mu_j, Var^{in}_n\big) \end{pmatrix} \end{array}.$$

As we know from the ($\Rightarrow$) direction of the proof, $\Omega^{out}_{nl} = \Omega^{out}_{xs}$ and so we have that $\mu^{out}_{nl} \in \Omega^{out}_{xs}$. Regarding the evaluation of the inner *SparqlForClause* we have that $\Omega^{in}_{xs} \preceq \Omega^{in}_{nl}$. We consider two cases: (i) $\mu^{in}_{nl} \in \Omega^{in}_{xs}$ or (ii) $\mu^{in}_{nl} \notin \Omega^{in}_{xs}$. In (i), we immediately get the desired result that $\text{dynEnv} \vdash Q \Rightarrow Val$. For (ii), consider $\mu^{xs}_{C_I}$ the XSPARQL instance of the inner *SparqlForClause* (created based on dynEnv^{xs}_1). As we can see from (T4), dynEnv^{xs}_1 (and thus also $\mu^{xs}_{C_I}$) includes the bindings for variables from each solution mapping $\mu_i \in \Omega^{out}_{xs}$. Thus, according to the XSPARQL BGP matching (cf. Definition 10), Ω^{in}_{xs} will contain all the solution mappings that are compatible with any solution mapping $\mu_i \in \Omega^{out}_{xs}$ and specifically those compatible with μ^{out}_{nl}. Since we know that μ^{in}_{nl} is compatible with μ^{out}_{nl}, we have that μ^{in}_{nl} must belong to Ω^{in}_{xs}; thus we can deduce that $\text{dynEnv} \vdash Q \Rightarrow Val$.

Inductive Step The proof follows from the recursive application of the base case, over a new dynamic environment determined by the opt_{nl} rewriting to $\text{dynEnv}_i \vdash opt_{nl}(ExprSingle)$.

The proof for nested queries with an XQuery `for` outer expression (Q2) is analogous where, in the preceding, the evaluation of the *SparqlForClause* from lines (1)–(3) of (Q3) is replaced by the evaluation of an XQuery *ForClause*, as presented by Draper et al. [27, Section 4.8.2]. □

C.7 Proof for Proposition 5

Proposition 5 *Let Q an XSPARQL expression of form* (Q4) *and dynEnv the dynamic environment of Q, then $\text{dynEnv} \vdash Q \Rightarrow Val$ if and only if $\text{dynEnv} \vdash opt_{sr}(Q) \Rightarrow Val$.*

Proof We start by showing the proof for the base case, where *ExprSingle* of (Q4) does not contain any occurrences of (Q4).

Base Case. ($\Rightarrow$) We start by showing that if $\text{dynEnv} \vdash Q \Rightarrow Val$ then $\text{dynEnv} \vdash opt_{sr}(Q) \Rightarrow Val$. Consider Ω^{out}_{xs} and Ω^{in}_{xs} the solution sequences returned, respectively, by the evaluation of the outer and inner *SparqlForClauses* of Q and $J = Vars^{out} \cap Vars\big(GGP^{in}\big)$ the set of join variables. Furthermore, consider dynEnv^{expr}_i the dynamic environment resulting from extending dynEnv with the variable mappings from the compatible solution mappings $\mu^{out}_{xs} \in \Omega^{out}_{xs}$ and $\mu^{in}_{xs} \in \Omega^{in}_{xs}$ such that $\text{dynEnv}^{expr}_{i} \vdash ExprSingle \Rightarrow Val$.

We now show the proof tree for each of the XQuery core expressions in each line of the opt_{sr} rewriting where, for each line, *Expr* represents the expressions of the following lines:

– `let` expression of line (1)

$$\begin{array}{c}
\text{dynEnv} \vdash \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } Vars^{out} \cup Vars^{in}\\ DatasetClause\\ \texttt{where } GGP^{out}\ GGP^{in}\\ \texttt{order by } OC^{out}\ OC^{in}\end{pmatrix} \Rightarrow \Omega_{sr}\\ \hline
\text{dynEnv}^{sr}_{1} \vdash Expr \Rightarrow Res\\ \hline
\begin{array}{ll} & \texttt{let \$xsp:results :=}\\ \text{dynEnv} \vdash & \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select } Vars^{out} \cup Vars^{in}\\ DatasetClause\\ \texttt{where } GGP^{out}\ GGP^{in}\\ \texttt{order by } OC^{out}\ OC^{in}\end{pmatrix}\\ & \texttt{return } Expr \Rightarrow Res\end{array}
\end{array}$$

where

$$\text{dynEnv}^{sr}_{1} = \text{dynEnv} + \text{varValue}\big(\texttt{xsp:results} \Rightarrow \Omega_{sr}\big)\ .$$

According to the SPARQL semantics, the solution sequence that results from evaluating the graph pattern $GGP^{out}\ GGP^{in}$, $\Omega_{sr} = \Omega^{out}_{sr} \bowtie \Omega^{in}_{sr}$ consists of all the solution mappings $\mu^{out}_{sr} \in \Omega^{out}_{sr}$ and $\mu^{in}_{sr} \in \Omega^{in}_{sr}$ such that μ^{out}_{sr} and μ^{in}_{sr} are *compatible*.
The following `for` expression iterates over all these compatible solution mappings:

– `for` expression of line (2)

$$\begin{array}{c}
\dfrac{\text{dynEnv}^{sr}_{1} \vdash \texttt{\$xsp:results//sr:result} \Rightarrow \mu_i}{\text{dynEnv}^{sr}_{2} \vdash ExprSingle \Rightarrow Res_i} \qquad \cdot^{\cdot^{\cdot}}\\ \hline
\text{dynEnv}^{sr}_{1} \vdash \begin{array}{l}\texttt{for \$xsp:result at \$}PosVar^{out}\\ \texttt{in \$xsp:results//sr:result}\\ \texttt{return } ExprSingle \Rightarrow Res_1, \cdots, Res_n\end{array}
\end{array}$$

where $\text{dynEnv}^{sr}_{2} = \text{dynEnv}^{sr}_{1} + \text{varValue}\begin{pmatrix}\texttt{xsp:result} \Rightarrow \mu_i;\\ PosVar^{out} \Rightarrow i\end{pmatrix}$

– `let` expressions of lines (3)–(4)

Here we consider all the `let` expressions represented by line (3), where $\$v \in Vars$:

$$\begin{array}{c}
\text{dynEnv}^{sr}_{2} \vdash \texttt{\$xsp:result/sr:binding[@name = \$}v\texttt{]/*} \Rightarrow V\\ \hline
\text{dynEnv}^{sr}_{3} \vdash ExprSingle \Rightarrow Res\\ \hline
\text{dynEnv}^{sr}_{2} \vdash \begin{array}{l}\texttt{let \$}v \texttt{ :=}\\ \quad\texttt{\$xsp:result/sr:binding[@name = }v\texttt{]/*}\\ \texttt{return } ExprSingle \Rightarrow Res\end{array}
\end{array}$$

where

$$\text{dynEnv}^{sr}_{3} = \text{dynEnv}^{sr}_{2} + \text{varValue}(v \Rightarrow V)\ .$$

Note that we are only considering `order by` solution modifiers; thus the number of results of each query is not changed. At most the ordering of the results is changed but this does not interfere with this proof and solution modifiers can be safely ignored in what follows.

Regarding the evaluation of the *SparqlForClause* from lines (1)–(4) of Q (evaluated considering dynEnv), the $opt_{sr}(Q)$ translates it into the `xsp:sparqlCall` from line (1), which is also evaluated over dynEnv. In this case, according to Lemma 2, we have that $\Omega^{out}_{sr} = \Omega^{out}_{xs}$ and then $\mu^{out}_{xs} \in \Omega^{out}_{sr}$.

Regarding the evaluation of the *SparqlForClause* from lines (5)–(8) of Q (evaluated considering some dynamic environment dynEnv^{expr}), the $opt_{sr}(Q)$ rewriting incorporates it into the `xsp:sparqlCall` from line (1), which is also evaluated over dynEnv. Considering that dynEnv is less restrictive than dynEnv^{expr}, i.e., dynEnv contains less bindings for variables than dynEnv^{expr}, and thus the evaluation of the inner *SparqlForClause* over dynEnv will contain all the solution mappings from Ω^{in}_{xs} and specifically μ^{in}_{xs}. As μ^{out}_{xs} and μ^{in}_{xs} are *compatible* we have that $\text{dynEnv} \vdash opt_{sr}(Q) \Rightarrow Val$.

($\Leftarrow$) Next we show that if $\text{dynEnv} \vdash opt_{sr}(Q) \Rightarrow Val$ then $\text{dynEnv} \vdash Q \Rightarrow Val$. Consider Ω^{out}_{sr} and Ω^{in}_{sr} as per the ($\Rightarrow$) direction of the proof and the set of join variables $J = Vars^{out} \cap vars\big(GGP^{in}\big)$. As we have seen Ω_{sr} contains all the solution mappings $\mu = \mu^{out}_{sr} \bowtie \mu^{in}_{sr}$ such that $\mu^{out}_{sr} \in \Omega^{out}_{sr}$ and $\mu^{in}_{sr} \in \Omega^{in}_{sr}$ and μ^{out}_{sr}, and μ^{in}_{sr} are *compatible*. Without loss of generality consider μ^{out}_{sr} and μ^{in}_{sr} the solution mappings where *Val* is deduced from.

Let us turn to the evaluation of $\text{dynEnv} \vdash Q \Rightarrow Val$.

– *SparqlForClause* from lines (1)–(4), where *Expr* corresponds to the *SparqlForClause* from lines (5)–(8) of Q. Again, the evaluation of this *SparqlForClause* consists of the application of Rule (D1):

$$\begin{array}{c}
\dfrac{\dfrac{\text{dynEnv} \vdash \mathit{fs{:}dataset}(DatasetClause) \Rightarrow DS}{\text{dynEnv} \vdash \mathit{fs{:}sparql}\begin{pmatrix}DS\texttt{, where } GGP^{out}\\ \texttt{order by } OC^{out}\end{pmatrix} \Rightarrow \mu_i}}{\text{dynEnv}^{xs}_{1} \vdash Expr \Rightarrow Value_i} \qquad \cdot^{\cdot^{\cdot}}\\ \hline
\text{dynEnv} \vdash \begin{array}{l}\texttt{for } Vars^{out} \texttt{ at \$}PosVar^{out}\ DatasetClause\\ \texttt{where } GGP^{out}\\ \texttt{order by } OC^{out}\\ \texttt{return } Expr \Rightarrow Value_1 \cdots Value_m\end{array}
\end{array}$$

where $Vars^{out} = \$Var^{out}_1 \cdots \$Var^{out}_n$, we have for each μ_i

$$\text{dynEnv}^{xs}_{1} = \begin{array}{l}\text{dynEnv} + \text{activeDataset}(DS)\\ + \text{varValue}\begin{pmatrix}PosVar^{out} \Rightarrow i;\\ Var^{out}_1 \Rightarrow \mathit{fs{:}value}\big(\mu_i, Var^{out}_1\big);\\ \ldots;\\ Var^{out}_n \Rightarrow \mathit{fs{:}value}\big(\mu_i, Var^{out}_n\big)\end{pmatrix}\end{array} \quad \text{(T5)}$$

– *SparqlForClause* of lines (4)–(6):
The evaluation of $\text{dynEnv}_i^{xs} \vdash \mathit{ExprSingle}^{out} \Rightarrow \mathit{Value}_i$ is shown next:

$$\dfrac{\dfrac{\dfrac{\text{dynEnv}_1^{xs} \vdash \mathit{fs{:}dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{DS}}{\text{dynEnv}_1^{xs} \vdash \mathit{fs{:}sparql}\begin{pmatrix}\mathit{DS},\\ \texttt{where}\ \mathit{GGP}^{in}\\ \texttt{order by}\ \mathit{OC}^{in}\end{pmatrix} \Rightarrow \mu_j}}{\text{dynEnv}_2^{xs} \vdash \mathit{ExprSingle} \Rightarrow \mathit{Value}_j} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv}_1^{xs} \vdash \begin{array}{l}\texttt{for}\ \mathit{Vars}^{in}\ \texttt{at}\ \$\mathit{PosVar}^{in}\ \mathit{DatasetClause}\\ \texttt{where}\ \mathit{GGP}^{in}\\ \texttt{order by}\ \mathit{OC}^{in}\\ \texttt{return}\ \mathit{ExprSingle}\ \Rightarrow \mathit{Value}_1 \cdots \mathit{Value}_m\end{array}}$$

where $\mathit{Vars}^{in} = \$\mathit{Var}_1^{in} \cdots \$\mathit{Var}_n^{in}$, we have for each μ_j

$$\text{dynEnv}_2^{xs} = \begin{array}{l}\text{dynEnv}_1^{xs} + \text{activeDataset}(\mathit{DS})\\ + \text{varValue}\begin{pmatrix}\mathit{PosVar}^{in} \Rightarrow j;\\ \mathit{Var}_1^{in} \Rightarrow \mathit{fs{:}value}\left(\mu_j, \mathit{Var}_1^{in}\right);\\ \ldots;\\ \mathit{Var}_n^{in} \Rightarrow \mathit{fs{:}value}\left(\mu_j, \mathit{Var}_n^{in}\right)\end{pmatrix}\end{array}.$$

As we have seen in the($\Rightarrow$) direction, we have that $\Omega_{sr}^{out} = \Omega_{xs}^{out}$ and so we have that $\mu_{sr}^{out} \in \Omega_{xs}^{out}$.

Consider C the expression context where dynEnv is included and μ_C the XSPARQL instance mapping of C. Further, consider P^{in} the graph pattern obtained from replacing the variables in GGP^{in} according to μ_C. Since $\mathit{vars}(\mathit{GGP}^{in}) \subseteq \mathit{vars}(P^{in})$ all solutions mappings returned by evaluating GGP^{in} under XSPARQL semantics are included in the solution sequence of evaluating P^{in} under SPARQL semantics, i.e., $\Omega_{xs}^{in} \preceq \Omega_{sr}^{in}$. We obtain two cases: (i) $\mu_{sr}^{in} \in \Omega_{xs}^{in}$ or (ii) $\mu_{sr}^{in} \notin \Omega_{xs}^{in}$. In (i) we immediately get that $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$. For (ii), consider $\mu_{C_1}^{xs}$ the XSPARQL instance of the inner *SparqlForClause*(created based on dynEnv_1^{xs}). As we can see from (T5), dynEnv_1^{xs} (and thus also $\mu_{C_1}^{xs}$) includes the bindings for variables from each solution mapping $\mu_i \in \Omega_{xs}^{out}$. Thus, according to the XSPARQL BGP matching (cf. Definition 10), Ω_{xs}^{in} will contain all the solution mappings that are compatible with any solution mapping $\mu_i \in \Omega_{xs}^{out}$ and specifically those compatible with μ_{sr}^{out}. Since we know that μ_{sr}^{in} is compatible with μ_{sr}^{out}, we have that μ_{sr}^{in} must belong to Ω_{xs}^{in}; thus we can deduce that $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$.

Inductive Step The proof follows from the recursive application of the base case, over a new dynamic environment determined by the opt_{sr} rewriting to $\text{dynEnv}_i \vdash \mathit{opt}_{sr}(\mathit{ExprSingle})$. □

C.8 Proof for Proposition 6

Proposition 6 *Let Q be an XSPARQL expression of form* (Q5) *and dynEnv the dynamic environment of Q, then* $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$ *if and only if* $\text{dynEnv} \vdash \mathit{opt}_{ng}(Q) \Rightarrow \mathit{Val}$.

Proof We start by showing the proof for the base case, where $\mathit{ExprSingle}_1$ and $\mathit{ExprSingle}_2$ of (Q5) do not contain any occurrences of (Q5).

Base Case. ($\Rightarrow$) Let us start by showing that if $\text{dynEnv} \vdash Q \Rightarrow \mathit{Val}$ then $\text{dynEnv} \vdash \mathit{opt}_{ng}(Q) \Rightarrow \mathit{Val}$. Consider Ω_{xs}^{in} the solution sequence returned by the evaluation of the inner *SparqlForClause*s of Q. Furthermore, consider dynEnv_i^{expr} such that $\text{dynEnv}_i^{expr} \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Val}$. The dynamic environment dynEnv_i^{expr} results from extending dynEnv with bindings for the outer variable $\$\mathit{VarName}$ and with the variable bindings from a solution mapping $\mu_{xs}^{in} \in \Omega_{xs}^{in}$ where $\mu_{xs}^{in}(\mathit{VarName}) = \$\mathit{VarName}$, i.e., the value for the join variable in the solution mapping μ_{xs}^{in} is the same as assigned to $\$\mathit{VarName}$.

We now show the proof tree for each of the XQuery core expressions in the opt_{ng} rewriting.

– `let` expression of line (1)
Considering $\mathit{NGP} = \{[]\ \texttt{:value}\ \$\mathit{VarName}\}$, we have

$$\dfrac{\dfrac{\begin{array}{l}\text{dynEnv} \vdash \texttt{xsp:createNG}\begin{pmatrix}\texttt{for}\ \$\mathit{VarName}\\ \mathit{OptTypeDeclaration}\\ \mathit{OptPositionalVar}\\ \texttt{in}\ \mathit{ExprSingle}_1\ \texttt{return}\\ \texttt{xsp:evalTemplate}(\mathit{NGP})\end{pmatrix}\\ \quad\Rightarrow \mathit{DS}\end{array}}{\text{dynEnv}_1^{ng} \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Res}}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{let}\ \$\texttt{xsp:ds} :=\\ \quad\texttt{xsp:createNG}\begin{pmatrix}\texttt{for}\ \$\mathit{VarName}\\ \mathit{OptTypeDeclaration}\\ \mathit{OptPositionalVar}\\ \texttt{in}\ \mathit{ExprSingle}_1\ \texttt{return}\\ \texttt{xsp:evalTemplate}(\mathit{NGP})\end{pmatrix}\\ \texttt{return}\ \mathit{ExprSingle} \Rightarrow \mathit{Res}\end{array}}$$

where

$$\text{dynEnv}_1^{ng} = \text{dynEnv} + \text{varValue}(\texttt{xsp:ds} \Rightarrow \mathit{DS})\ . \qquad \text{(T6)}$$

– `let` expression of line (2)
Consider the dataset clause $\mathit{DatasetClause}^{ng} = \mathit{DatasetClause} \cup \{\texttt{from named}\ \$\texttt{xsp:ds}\}$ and the graph pattern $\mathit{WhereClause}^{ng} = \mathit{WhereClause} \cup \texttt{where}\,\{\texttt{graph}\ \$\texttt{xsp:ds}\ \{[]\ \texttt{:value}\ \$\mathit{VarName}\}\}$.

$$\frac{\dfrac{\text{dynEnv}_1^{ng} \vdash \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select}\\ \mathit{Vars} \cup \{\$\mathit{VarName}\}\\ \mathit{DatasetClause}^{ng}\\ \mathit{WhereClause}^{ng}\\ \mathit{SolutionModifier}\end{pmatrix} \Rightarrow \Omega}{\text{dynEnv}_2^{ng} \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Res}}}{\text{dynEnv}_1^{ng} \vdash \begin{array}{l}\texttt{let \$xsp:results :=}\\ \quad \texttt{xsp:sparqlCall}\begin{pmatrix}\texttt{select}\\ \mathit{Vars} \cup \{\$\mathit{VarName}\}\\ \mathit{DatasetClause}^{ng}\\ \mathit{WhereClause}^{ng}\\ \mathit{SolutionModifier}\end{pmatrix}\\ \texttt{return } \mathit{ExprSingle}_2 \Rightarrow \mathit{Res}\end{array}}$$

where

$$\text{dynEnv}_2^{ng} = \text{dynEnv}_1^{ng} + \text{varValue}\big(\texttt{xsp:results} \Rightarrow \Omega\big) \ .$$

The new merged dataset, *DatasetClause*^{ng}, is created based on *DatasetClause* and the newly created named graph *NG*. Since the URI that identifies the newly created named graph *NG* is distinct from any URI of named graphs present in *DatasetClause*, the triples included in *NG* will never be a solution for *WhereClause*, and will be matched only by the graph pattern `where` $\{$`graph` \$xsp:ds $\{$[] :value \$*VarName*$\}\}$.
Consider C the expression context where dynEnv is included, μ_C the XSPARQL instance mapping of C and P^{out} and P^{in} the graph patterns obtained from, respectively, replacing the variables in *WhereClause* and `where`$\{$`graph` \$xsp:ds $\{$ [] :value \$*VarName* $\}\}$ according to μ_C.
Furthermore, let $\Omega_{ng}^{out} = \mathit{eval}\big(\mathit{DatasetClause}^{ng}, P^{out}\big)$ and $\Omega_{ng}^{in} = \mathit{eval}\big(\mathit{DatasetClause}^{ng}, P^{in}\big)$. According to SPARQL semantics, the pattern solution that results from evaluating *WhereClause*, $\Omega_{ng} = \Omega_{ng}^{out} \bowtie \Omega_{ng}^{in}$ consists of all the solution mappings $\mu_{out} \in \Omega_{ng}^{out}$ and $\mu_{in} \in \Omega_{ng}^{in}$ such that μ_{out} and μ_{in} are *compatible*.

– `for` expression of line (3)

$$\frac{\dfrac{\text{dynEnv}_2^{ng} \vdash \texttt{\$xsp:results//sr:result} \Rightarrow \mu_i}{\text{dynEnv}_3^{ng} \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Res}_i} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv}_2^{ng} \vdash \begin{array}{l}\texttt{for \$xsp:result at \$xsp:result\_pos}\\ \quad \texttt{in \$xsp:results//sr:result}\\ \texttt{return } \mathit{ExprSingle} \Rightarrow \mathit{Res}_1, \cdots, \mathit{Res}_n\end{array}}$$

where

$$\text{dynEnv}_3^{ng} = \text{dynEnv}_2^{ng} + \text{varValue}\begin{pmatrix}\texttt{xsp:result} \Rightarrow \mu_i;\\ \texttt{xsp:result\_pos} \Rightarrow i\end{pmatrix} \ .$$

– `let` expressions of lines (4)–(5)
Here we consider all the `let` expressions represented by line (4), where $\$v \in \mathit{Vars}$:

$$\frac{\dfrac{\text{dynEnv}_3^{ng} \vdash \texttt{\$xsp:result/sr:binding[@name = \$}v\texttt{]/*} \Rightarrow V}{\text{dynEnv}_4^{ng} \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Res}}}{\text{dynEnv}_3^{ng} \vdash \begin{array}{l}\texttt{let \$}v \texttt{ :=}\\ \quad \texttt{\$xsp:result/sr:binding[@name = \$}v\texttt{]/*}\\ \texttt{return } \mathit{ExprSingle} \Rightarrow \mathit{Res}\end{array}}$$

where

$$\text{dynEnv}_4^{ng} = \text{dynEnv}_3^{ng} + \text{varValue}(v \Rightarrow V) \ .$$

Similarly to the proof of Proposition 5, we are only considering `order by` solution modifiers; these only change the order of the solution sequences and thus can be safely ignored for this proof.

Regarding the evaluation of the XQuery `for` clause from lines (1)–(2) of Q (evaluated considering dynEnv), the $\mathit{opt}_{ng}(Q)$ translates it into the `xsp:sparqlCall` from line (2), which is evaluated considering dynEnv$_1^{ng}$. As we can see from (T6), dynEnv$_1^{ng}$ is based on dynEnv by adding the binding for the `xsp:ds` variable. Since this variable belongs to the `xsp:` reserved namespace, it is not allowed in the *WhereClause* and so we have that the results of evaluating the `xsp:sparqlCall` function over dynEnv or dynEnv$_1^{ng}$ will be the same.

Regarding the evaluation of the *SparqlForClause*from lines (3)–(4) of Q (evaluated considering some dynamic environment dynEnv^{expr}), the $\mathit{opt}_{ng}(Q)$ also incorporates it into the `xsp:sparqlCall` from line (2), which is evaluated over dynEnv$_1^{ng}$. Considering that dynEnv$_1^{ng}$ is less restrictive than dynEnv^{expr}, i.e. dynEnv$_1^{ng}$ contains less bindings for variables than dynEnv^{expr}, the evaluation of the inner *SparqlForClause*over dynEnv$_1^{ng}$ will contain all the solution mappings from Ω_{xs}^{in} and specifically μ_{in}. As μ_{out} and μ_{in} are *compatible* we have that dynEnv $\vdash ng(expr) \Rightarrow \mathit{Val}$.

($\Leftarrow$) Next we will show that if dynEnv $\vdash \mathit{opt}_{ng}(Q) \Rightarrow \mathit{Val}$ then dynEnv $\vdash Q \Rightarrow \mathit{Val}$. Consider Ω_{ng}^{out} and Ω_{ng}^{in} the solution sequences returned by, respectively, the evaluation of the new *WhereClause*^{ng} and *WhereClause*. As we have seen Ω_{ng} contains all the solution mappings $\mu = \mu_{ng}^{out} \bowtie \mu_{ng}^{in}$, where $\mu_{ng}^{out} \in \Omega_{ng}^{out}$ and $\mu_{ng}^{in} \in \Omega_{ng}^{in}$, such that μ_{ng}^{out} and μ_{ng}^{in} are *compatible*. Again, consider μ_{ng}^{out} and μ_{ng}^{in} the pattern solutions where *Val* is deduced from.

Let us turn to the evaluation of dynEnv $\vdash Q \Rightarrow \mathit{Val}$.

– XQuery `for` clause from lines (1)–(2):
Expr corresponds to the *SparqlForClause* from lines (3)–(4) of Q.

$$\dfrac{\dfrac{\text{dynEnv} \vdash \mathit{ExprSingle}_1 \Rightarrow V_i}{\text{dynEnv}_i^{xs} \vdash \mathit{ExprSingle}_1 \Rightarrow \mathit{Value}_i} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for } \$\mathit{VarName}\ \mathit{OptTypeDeclaration}\\ \quad \mathit{OptPositionalVar}\ \texttt{in}\ \mathit{ExprSingle}_1\\ \texttt{return}\ \mathit{Expr} \Rightarrow \mathit{Value}_i \ldots \mathit{Value}_n\end{array}}$$

we have for each V_i:

$$\text{dynEnv}_i^{xs} = \text{dynEnv} \ + \text{varValue}\big(\mathit{VarName} \Rightarrow V_i\big) \ . \qquad \text{(T7)}$$

– *SparqlForClause* of lines (2)–(4):

$$\dfrac{\dfrac{\dfrac{\text{dynEnv}_i^{xs} \vdash \mathit{fs{:}dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{DS}}{\text{dynEnv}_i^{xs} \vdash \mathit{fs{:}sparql}\begin{pmatrix}\mathit{DS}, \mathit{WhereClause},\\ \mathit{SolutionModifier}\end{pmatrix} \Rightarrow \mu_j}}{\text{dynEnv}_j^{xs} \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Value}_j} \quad \cdot^{\cdot^{\cdot}}}{\text{dynEnv}_i^{xs} \vdash \begin{array}{l}\texttt{for}\ \mathit{Vars}\ \texttt{at}\ \$\mathit{PosVar}\ \mathit{DatasetClause}\\ \mathit{WhereClause}\ \mathit{SolutionModifier}\\ \texttt{return}\ \mathit{ExprSingle}_2 \ \Rightarrow \mathit{Value}_1 \cdots \mathit{Value}_m\end{array}}$$

where, considering $\mathit{Vars} = \$\mathit{Var}_1 \ldots \$\mathit{Var}_n$, we have for each μ_j:

$$\text{dynEnv}_j^{xs} = \begin{array}{l}\text{dynEnv}_i^{xs} + \text{activeDataset}(\mathit{DS})\\ + \text{varValue}\begin{pmatrix}\mathit{PosVar} \Rightarrow j;\\ \mathit{Var}_1 \Rightarrow \mathit{fs{:}value}\big(\mu_j, \mathit{Var}_1\big);\\ \ldots;\\ \mathit{Var}_n \Rightarrow \mathit{fs{:}value}\big(\mu_j, \mathit{Var}_n\big)\end{pmatrix}\end{array} .$$

As we have seen in the ($\Rightarrow$) direction, we have that $\Omega_{ng}^{out} = \Omega_{xs}^{out}$ and so we have that $\mu_{ng}^{out} \in \Omega_{xs}^{out}$.

Consider C the expression context where dynEnv is included and μ_C the XSPARQL instance mapping of C. Further consider P^{in} the graph pattern obtained from replacing the variables in $\mathit{WhereClause}^{in}$ according to μ_C. Since we know that $\mathit{vars}\big(\mathit{WhereClause}^{in}\big) \subseteq \mathit{vars}\big(P^{in}\big)$, all solutions mappings returned by evaluating $\mathit{WhereClause}^{in}$ under XSPARQL semantics are included in the pattern solution of evaluating P^{in} under SPARQL semantics, i.e., $\Omega_{xs}^{in} \preceq \Omega_{ng}^{in}$. We obtain two cases: (i) $\mu_{ng}^{in} \in \Omega_{xs}^{in}$; or (ii) $\mu_{ng}^{in} \notin \Omega_{xs}^{in}$. In (i) we immediately get that dynEnv $\vdash Q \Rightarrow \mathit{Val}$. For (ii), consider $\mu_{C_1}^{xs}$ the XSPARQL instance of the inner *SparqlForClause*(created based on dynEnv_1^{xs}). As we can see from (T7), dynEnv_1^{xs} (and thus also $\mu_{C_1}^{xs}$) includes the bindings for variables from each solution mapping $\mu_i \in \Omega_{xs}^{out}$. Thus, according to the XSPARQL BGP matching (cf. Definition 10), Ω_{xs}^{in} will contain all the solution mappings that are compatible with any solution mapping $\mu_i \in \Omega_{xs}^{out}$ and specifically those compatible with μ_{ng}^{out}. Since we know that μ_{ng}^{in} is compatible with μ_{ng}^{out}, we have that μ_{ng}^{in} must belong to Ω_{xs}^{in}; thus we can deduce that dynEnv $\vdash Q \Rightarrow \mathit{Val}$.

Inductive Step. Let us assume that, for some arbitrary dynEnv_i, $\text{dynEnv}_i \vdash \mathit{ExprSingle}_1 \Rightarrow \mathit{Val}_i$ if and only if $\text{dynEnv}_i \vdash \mathit{opt}_{ng}\big(\mathit{ExprSingle}_1\big) \Rightarrow \mathit{Val}_i$. According to the opt_{ng} rewriting, there must exist a dynEnv_j that is the extension of dynEnv_i with Val_i and thus $\text{dynEnv}_j \vdash \mathit{ExprSingle}_2 \Rightarrow \mathit{Val}$ if and only if $\text{dynEnv}_j \vdash \mathit{opt}_{ng}\big(\mathit{ExprSingle}_2\big) \Rightarrow \mathit{Val}$. Consequently, we have that dynEnv $\vdash Q \Rightarrow \mathit{Val}$ if and only if dynEnv $\vdash \mathit{opt}_{ng}(Q) \Rightarrow \mathit{Val}$. □

D The XMarkRDF Benchmark

For the evaluation of our implementation we created a benchmark suite based on the XMark benchmark suite [57]. According to [3], the XMark suite is the most widely used benchmark suite for XQuery. It provides a data generator that produces XML data simulating an auction website (including information about persons and items they bid for) and includes 20 XQuery queries, referred to as q_1 to q_{20} henceforth, over this generated data.

In order to benchmark the XSPARQL language we also require data in RDF format; hence we provide transformations (in fact, using XSPARQL queries) from XML datasets generated by XMark into RDF triples. In this transformation we replicate all the data in the original XMark datasets as RDF triples. We start by generating IRIs for each XML element that represents concepts like "persons," "items," "bids," etc. Inner XML element names are then converted into RDF predicates and used to link the generated IRIs to the leaf element values which are converted into RDF literals. Next, we converted the XMark queries into corresponding XSPARQL queries using *SparqlForClause*s to access the RDF data. We call this new benchmark suite the XMarkRDF benchmark and is available for download at http://xsparql.deri.org/data/XMarkRDF/.

From the initial set of 20 queries there are 5 queries (q_8–q_{12}) which contain nested expressions. They are described informally in the XMark suite as follows:

(q_8) "List the names of persons and the number of items they bought;"
(q_9) "List the names of persons and the names of the items they bought in Europe;"
(q_{10}) "List all persons according to their interest;"
(q_{11}) "List the number of items currently on sale whose price does not exceed 0.02% of the seller's income;" and
(q_{12}) "For each richer-than-average person, list the number of items currently on sale whose price does not exceed 0.02% of the person's income."

Figure 20a, b presents XMark query q_9 and its translated XSPARQL version in XMarkRDF, respectively. We have made two changes to the XMark queries: (1) SPARQL que-

```
 1 declare ordering unordered;
 2 declare variable $xml external;
 3
 4 let $auction := doc($xml) return
 5 let $ca := $auction/site/closed_auctions/closed_auction
 6 return let $ei := $auction/site/regions/europe/item
 7 for $p in $auction/site/people/person
 8 let $a := for $t in $ca
 9           where $p/@id = $t/buyer/@person return
10           let $n := for $t2 in $ei
11                     where $t/itemref/@item = $t2/@id
12                     return $t2
13 return <item>{$n/name/text()}</item>
14 return <person name="{$p/name/text()}">{$a}</person>
```

(a) Query q_9 in XQuery (XMark)

```
 1 prefix : <http://xsparql.deri.org/data/>
 2 prefix foaf: <http://xmlns.com/foaf/0.1/>
 3 declare variable $rdf external;
 4
 5 for $person $name from $rdf
 6 where { $person foaf:name $name }
 7 return <person name="{$name}">{
 8   for * from $rdf where { $ca :buyer $person .
 9     optional { $ca :itemRef $itemRef .
10       $itemRef :locatedIn [ :name "europe" ].
11       $itemRef :name $itemname } }
12   return <item>{$itemname}</item>
13 }</person>
```

(b) Query q_9 in XSPARQL (XMarkRDF)

```
 1 declare namespace ac="http://xsparql.deri.org/data/";
 2 declare namespace foaf="http://xmlns.com/foaf/0.1/";
 3 declare variable $rdf external;
 4
 5 for ($n, $m) in
 6   SELECT $person $name FROM $rdf
 7   WHERE { $person foaf:name $name . }
 8 return
 9   <person name="{$n}">{ for ($item) in
10      SELECT $itemname WHERE { $ca ac:buyer $person .
11        optional { $ca ac:itemRef $itemRef .
12          $itemRef ac:locatedIn [ ac:name "europe" ] .
13          $itemRef ac:name $itemname } .
14     } return <item>{$itemname}</item>
15   }</person>
```

(c) Query q_9 in SPARQL2XQuery (XMarkRDF$_{S2XQ}$)

Fig. 20 Variants of benchmark query q_9

ries do not guarantee any default ordering; hence all original XMark queries were declared unordered—as a consequence the XQuery engine is not required to follow document order when executing the query; and (2) we added the external variables `$xml` and `$rdf` in the XQuery and XSPARQL query, respectively, as parameters used to specify the URL identifying the input benchmark instance.

We included the SPARQL2XQuery system, which is similar in spirit to XSPARQL, by [33] in our system comparison. While the language allows to perform similar queries to the XSPARQL language, the implementation follows a different approach to integrate the XML and RDF data. Rather than performing interleaved calls to a SPARQL engine, the SPARQL2XQuery system relies on translating the RDF data into a pre-defined XML format and transforming SPARQL queries into equivalent XQuery over the pre-defined XML format. The translated queries can be directly executed using a native XQuery engine. For further comparison between XSPARQL and the SPARQL2XQuery language, and other related works, we refer the reader to Sect. 8.

Table 4 XMarkRDF$_{S2XQ}$ dataset and translation times

Scaling factor	Dataset size (MB)	Translation times (s)
0.01	3.3	18.94
0.02	6.4	18.30
0.05	16.1	26.08
0.10	32.7	39.01
0.20	65.3	62.35
0.50	162.3	143.35
1.00	326.2	329.93

Query q_9, as presented in Fig. 20c, is ready to be evaluated by the SPARQL2XQuery system over the XMarkRDF$_{S2XQ}$ dataset. Please note that this query follows the syntax presented in [33], since we only had access to the implementation of the translation from SPARQL to XQuery, while the evaluation was done using the associated XQuery code. We focussed in our experimental evaluation on query response time rather than on data transformation time, and as SPARQL2XQuery requires an additional translation step from RDF to a custom RDF/XML format, we converted the XMarkRDF RDF data into the format required by the SPARQL2XQuery system. An overview of this translation process, including the translation times, is presented in Table 4. We denote these new datasets, containing the RDF/XML format required for the SPARQL2XQuery, by XMarkRDF$_{S2XQ}$.

Optimising XMarkRDF Nested Queries

The different rewritings presented in Sect. 6 can be applied to the four nested queries q_8–q_{11}. Query q_{12} also consists of a nested expression; however, the most accurate translation of this query into XSPARQL results in the dependent variable not being *strictly bound* since it occurs only in the `filter` of the inner query. As such, we cannot apply the different rewritings to this query.

XMarkRDF query q_9 is presented in Fig. 20 on the facing page. This query is close to queries q_8, q_{10}, and q_{11} and consists of a nested expression: the inner `for` expression of the query (lines 8–12) is executed once for each person matched by the outer expression (lines 5–6), which means that one SPARQL call will be made for each person separately. Thus, the number of SPARQL calls performed in the inner expression directly depends on the size of the dataset (cf. Table 5 for details). Queries q_8, q_9, and q_{11} evaluate the inner expression for each person, while q_{10} evaluates the inner expression for each category. Each dataset contains usually about 25 times

Table 5 Benchmark dataset description

Scaling factor	Persons	Categories	XMark (MB)	XMarkRDF (MB)
0.01	255	10	1.1	1.2
0.02	510	20	2.3	2.3
0.05	1,275	50	5.8	5.8
0.10	2,550	100	11.7	12.4
0.20	5,100	200	23.5	24.9
0.50	12,750	500	58.0	61.7
1.00	25,500	1,000	116.5	124.8

```
<person name="Alagu Nyrup">
 <item>monument </item>
 <item>herring hush </item>
</person>
```

(a) Query q_9 – bought items grouped by person

```
<item name="monument ">Alagu Nyrup</item>
<item name="herring hush ">Alagu Nyrup</item>
```

(b) Query q'_9 – flat list of items and buyer

Fig. 21 Example output excerpts of queries q_9 and q'_9

more persons than categories. The rewriting strategies presented in Sect. 6 reduce the number of SPARQL calls to two: one to get all the people (similar to the direct rewriting version), and one additional SPARQL call for retrieving all the information about all the auctions in the dataset. Although the query remains exponential, the practical evaluation will show that reducing the number of SPARQL calls drastically improves query execution times (Table 2).

As mentioned in Sect. 6.2, for the SPARQL-based rewritings, we want the query output to be computable directly in SPARQL without any further processing, i.e., we do not want to use XQuery for further processing of the SPARQL results, and the query should be expressible in SPARQL without features from SPARQL 1.1. Since the original nested queries q_8–q_{11} group the output results (while optionally applying some aggregation function), we need to include modified versions of these benchmark queries for the evaluation of the SPARQL based rewritings. In these modified queries, denoted q'_8–q'_{11}, we changed the return format of the queries to consist of a flattened representation of the output of the original query. An example of the output for queries q_9 and q'_9 is presented in Fig. 21. All queries q'_i and q''_i follow a similar strategy for reformatting the output: the queries resulting from applying opt_{sr} are named q'_8–q'_{11}, while the queries that consist of an outer `for` expression—to which opt_{ng} was applied—are q''_8–q''_{11}.

References

1. Abiteboul S, Hull R, Vianu V (1995) Foundations of databases. Addison-Wesley, New York
2. Adida B, Birbeck M, McCarron S, Pemberton S (2008) RDFa in XHTML: syntax and processing. W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014
3. Afanasiev L, Marx M (2008) An analysis of XQuery benchmarks. Inf Syst 33(2):155–181
4. Akhtar W, Kopecký J, Krennwallner T, Polleres A (2008) XSPARQL: traveling between the XML and RDF worlds—and avoiding the XSLT pilgrimage. In: ESWC'08. Springer, Berlin, pp 432–447
5. Angles R, Gutierrez C (2010) SQL nested queries in SPARQL. In: AMW'10, CEUR-WS.org, vol 619
6. Battle S (2006) Gloze: XML to RDF and back again. In: Jena user conference'06
7. Beckett D, Berners-Lee T (2008) Turtle—terse RDF triple language. http://www.w3.org/TeamSubmission/turtle/
8. Beckett D, Broekstra J (2008) SPARQL query results XML format. W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/
9. Beckett D, McBride B (eds) (2004) RDF/XML syntax specification (Revised). W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/
10. Berger S, Bry F, Furche T, Linse B, Schroeder A (2006) Beyond XML and RDF: the versatile Web query language Xcerpt. In: Carr L, Roure DD, Iyengar A, Goble CA, Dahlin M (eds) WWW. ACM, New York, pp 1053–1054
11. Berglund A, Boag S, Chamberlin D, Fernández MF, Kay M, Robie J, Siméon J (2010) XML path language (XPath) 2.0, 2nd edn. W3C recommendation, World Wide Web consortium. http://www.w3.org/TR/2010/REC-xpath20-20101214/
12. Berrueta D, Labra JE, Herman I (2008) XSLT+SPARQL: scripting the semantic web with SPARQL embedded into XSLT stylesheets. In: Workshop on scripting for the semantic web
13. Beyer KS, Cochrane R, Josifovski V, Kleewein J, Lapis G, Lohman GM, Lyle R, Özcan F, Pirahesh H, Seemann N, Truong TC, der Linden BV, Vickery B, Zhang C (2005) System RX: one part relational, one part XML. In: Özcan F (ed) SIGMOD conference. ACM, New York, pp 347–358
14. Bikakis N, Gioldasis N, Tsinaraki C, Christodoulakis S (2009) Querying XML Data with SPARQL. In: DEXA'09, vol 5690. Springer, Berlin, pp 372–381
15. Bischof S (2010) Full XSPARQL grammar. http://xsparql.deri.org/doc/grammar.html
16. Bischof S (2012) Optimising XML-RDF data integration. In: Simperl E (ed) ESWC 2012. LNCS, vol 7295. Springer, Heidelberg, pp 838–843
17. Bohring H, Auer S (2005) Mapping XML to OWL ontologies. In: Leipziger Informatik-Tage, GI, vol 72, pp 147–156
18. Bray T, Paoli J, Sperberg-Mcqueen CM, Maler E, Yergeau F (2008) Extensible markup language (XML) 1.0, 5th edn. W3C recommendation, World Wide Web consortium. http://www.w3.org/TR/2008/REC-xml-20081126/
19. Brickley D, Guha R (2004) RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/
20. Brickley D, Miller L (2007) FOAF vocabulary specification. http://xmlns.com/foaf/spec/
21. Carroll JJ, Stickler P (2004) TriX, RDF triples in XML. Tech. Rep. HPL-2003-268, HP Labs. http://www.hpl.hp.com/techreports/2004/HPL-2004-56.html
22. Chamberlin D, Robie J, Boag S, Fernández MF, Siméon J, Florescu D (2010) XQuery 1.0: an XML query language, 2nd edn.

W3C recommendation, W3C. http://www.w3.org/TR/2010/REC-xquery-20101214/
23. Clark KG, Feigenbaum L, Torres E (2008) SPARQL protocol for RDF. W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-rdf-sparqlprotocol-20080115/
24. Connolly D (2007) Gleaning resource descriptions from dialects of languages (GRDDL). W3C recommendation, W3C. http://www.w3.org/TR/2007/REC-grddl-20070911/
25. Corby O, Kefi-Khelif L, Cherfi H, Gandon F, Khelif K (2009) Querying the semantic web of data using SPARQL, RDF and XML. Tech. Rep. 6847, Institut National de Recherche en Informatique et en Automatique
26. Deursen DV, Poppe C, Martens G, Mannens E, Walle RVd (2008) XML to RDF conversion: a generic approach. In: AXMEDIS'08. IEEE, pp 138–144
27. Draper D, Fankhauser P, Fernández M, Malhotra A, Rose K, Rys M, Siméon J, Wadler P (2010) XQuery 1.0 and XPath 2.0 formal semantics, 2nd edn. W3C recommendation, W3C. http://www.w3.org/TR/2010/REC-xquerysemantics-20101214/
28. Droop M, Flarer M, Groppe J, Groppe S, Linnemann V, Pinggera J, Santner F, Schier M, Schopf F, Staffler H, Zugal S (2008) Embedding XPath queries into SPARQL queries. In: ICEIS'08, pp 5–14
29. Farrell J, Lausen H (2007) Semantic annotations for WSDL and XML schema. W3C recommendation, W3C. http://www.w3.org/TR/2007/REC-sawsdl-20070828/
30. Fernández MF, Malhotra A, Marsh J, Nagy M, Walsh N (2010) XQuery 1.0 and XPath 2.0 data model (XDM), 2nd edn. W3C recommendation, W3C. http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/
31. Fischer P, Florescu D, Kaufmann M, Kossmann D (2011) Translating SPARQL and SQL to XQuery. In: XMLPrague'11, pp 81–98
32. Gearon P, Passant A, Polleres A (2011) SPARQL 1.1 update. W3C working draft, W3C. http://www.w3.org/TR/2011/WD-sparql11update-20110512/
33. Groppe S, Groppe J, Linnemann V, Kukulenz D, Hoeller N, Reinke C (2008) Embedding SPARQL into XQuery/XSLT. In: SAC'08. ACM, New York, pp 2271–2278
34. Grust T, Sakr S, Teubner J (2004) XQuery on SQL hosts. In: Nascimento MA, Özsu MT, Kossmann D, Miller RJ, Blakeley JA, Schiefer KB (eds) VLDB. Morgan Kaufmann, pp 252–263
35. Grust T, Rittinger J, Teubner J (2007) eXrQuy: order indifference in XQuery. In: Chirkova R, Dogac A, Özsu MT, Sellis TK (eds) ICDE. IEEE, pp 226–235
36. Grust T, Mayr M, Rittinger J (2010) Let SQL drive the XQuery workhorse (XQuery join graph isolation). In: EDBT'10, vol 426. ACM, pp 147–158
37. Hartig O, Heese R (2007) The SPARQL query graph model for query optimization. In: Franconi E, Kifer M, May W (eds) ESWC, vol 4519. Springer, Berlin, pp 564–578
38. Harris S, Seaborne A (2011) SPARQL 1.1 query language. W3C working draft, W3C. http://www.w3.org/TR/2011/WD-sparql11query-20110512/
39. Hayes P (2004) RDF semantics. W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/
40. Iannella R (2010) Representing vCard Objects in RDF. http://www.w3.org/Submission/vcard-rdf/. W3C member submission
41. Katz H, Chamberlin D, Kay M, Wadler P, Draper D (2003) XQuery from the experts: a guide to the W3C XML query language. Addison-Wesley
42. Kay M (ed) (2007) XSL transformations (XSLT) version 2.0. W3C recommendation, W3C. http://http://www.w3.org/TR/2007/REC-xslt20-20070123/
43. Koch C (2006) On the complexity of nonrecursive XQuery and functional query languages on complex values. ACM Trans Database Syst 31(4):1215–1256
44. Kopecký J, Vitvar T, Bournez C, Farrell J (2007) SAWSDL: semantic annotations for WSDL and XML schema. IEEE Internet Comput 11(6):60–67
45. Malhotra A, Melton J, Walsh N (eds) (2010) XQuery 1.0 and XPath 2.0 functions and operators, 2nd edn. W3C recommendation, W3C. http://www.w3.org/TR/2010/REC-xpath-functions-20101214/
46. Manola F, Miller E (2004) RDF primer. W3C recommendation, W3C. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/
47. May N, Helmer S, Moerkotte G (2003) Three cases for query decorrelation in XQuery. In: Xsym'03, vol 2824. Springer, Berlin, pp 70–84
48. May N, Stuckenschmidt H (2007) Querying embedded RDF with XML technology: a feasibility study. In: XML Tage 2007. Freie University, Berlin
49. Passant A, Kopecký J, Corlosquet S, Berrueta D, Palmisano D, Polleres A (2009) XSPARQL: use cases. http://www.w3.org/Submission/xsparql-use-cases/. W3C member submission
50. Pérez J, Arenas M, Gutierrez C (2008) nSPARQL: a navigational language for RDF. In: ISWC'08, vol 5318. Springer, Berlin, pp 66–81
51. Pérez J, Arenas M, Gutierrez C (2009) Semantics and complexity of SPARQL. ACM Trans Database Syst 34(3):1–45
52. Polleres A (2007) From SPARQL to rules (and back). In: WWW'07
53. Polleres A, Scharffe F, Schindlauer R (2007) SPARQL++ for mapping between RDF vocabularies. In: ODBASE'07. Springer, Berlin
54. Prud'hommeaux E, Seaborne A (eds) (2008) SPARQL query language for RDF. W3C recommendation, W3C. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/
55. Robie J, Chamberlin D, Dyck M, Florescu D, Melton J, Siméon J (2011) XQuery update facility 1.0. W3C recommendation, W3C. http://www.w3.org/TR/xquery-update-10/
56. Rodrigues T, Rosa P, Cardoso J (2008) Moving from syntactic to semantic organizations using JXML2OWL. Comput Ind 59(8):808–819
57. Schmidt A, Waas F, Kersten ML, Carey MJ, Manolescu I, Busse R (2002) XMark: a benchmark for XML data management. In: VLDB'02. Morgan Kaufmann, pp 974–985
58. Walsh N (2003) RDF Twig: accessing RDF graphs in XSLT. In: Extreme markup languages'03
59. Zhou M, Wu Y (2010) XML-based RDF data management for efficient query processing. In: Dong XL, Naumann F (eds) Proceedings of the 13th international workshop on the web and databases 2010, WebDB 2010, Indianapolis, Indiana, USA, June 6, 2010