MAPPING BETWEEN RDF AND XML WITH XSPARQL

Stefan Bischof Stefan Decker
Thomas Krennwallner Nuno Lopes
Axel Polleres

DERI TECHNICAL REPORT 2011-04-04
APRIL 2011

DERI Galway
IDA Business Park
Galway, Ireland
www.deri.ie

DERI TECHNICAL REPORT

DERI TECHNICAL REPORT 2011-04-04, APRIL 2011

MAPPING BETWEEN RDF AND XML WITH XSPARQL

Stefan Bischof [1] Stefan Decker [2] Thomas Krennwallner [3]
Nuno Lopes [4] Axel Polleres [5]

Abstract.
One of the promises of Semantic Web applications is to seamlessly deal with heterogeneous data. While 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 from one XML format into another, the more recent Resource Description Framework (RDF) has become another popular standard for data representation and exchange, supported by its own powerful query language SPARQL, that enables extraction and transformation of RDF data. Being able to work with these two languages using a common framework eliminates several unnecessary steps that are currently necessary when handling both formats side by side. In this report we present the XSPARQL language that, by combining XQuery and SPARQL, allows to query XML and RDF data using the same framework and, respectively transform 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.
Keywords: XML, RDF, SPARQL, XQuery, Transformation language

[1]Digital Enterprise Research Institute, National University of Ireland, Galway.
E-mail: stefan.bischof@deri.org
[2]Digital Enterprise Research Institute, National University of Ireland, Galway.
E-mail: stefan.decker@deri.org
[3]Institut für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Vienna, Austria.
E-mail: tkren@kr.tuwien.ac.at
[4]Digital Enterprise Research Institute, National University of Ireland, Galway.
E-mail: nuno.lopes@deri.org
[5]Digital Enterprise Research Institute, National University of Ireland, Galway .
E-mail: axel.polleres@deri.org

Acknowledgements: The work presented in this report has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Lion-2).

Contents

1 Introduction

XML [1] has become, after its first edition in 1998 [2], a well established and widely adopted interchange format for data on the Web. Accompanying standards, such as XSL Transformations (XSLT) [3] and, more recently, XQuery [4], both based on the XML Path Language (XPath) [5], 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) [6, 7] 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. As opposed to the tree-like structure of XML, RDF structures data in sets of triples, often interpreted as edges of a directed, labelled graph. Querying RDF graphs and converting between them can be performed using SPARQL [8], 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 Gleaning Resource Descriptions from Dialects of Languages [9] (GRDDL) and Semantic Annotations for WSDL [10] (SAWSDL) working groups. As we will show, approaches relying on the standard XML serialisation of RDF [11] and using, e.g., XSLT for such transformations have several disadvantages. In contrast, our proposed language XSPARQL provides an integration framework for handling XML and RDF side by side combining XQuery and SPARQL. While both XQuery and SPARQL languages operate on different data models, respectively XML and RDF, we show that the merge of both 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. An additional use for XSPARQL is the conversion between RDF graphs by extending SPARQL's expressiveness for such transformations, by allowing, for instance, nested XSPARQL queries in the graph construction step.

Since its first version [12], XSPARQL has gained quite some community interest and several practical use cases have been presented in a W3C member submission [13]. Based on these experiences, the present report makes the following main contributions:

- We present the syntax and formal semantics of XSPARQL based on the XQuery formal semantics [14]. In comparison to our initial publication [12], 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 from [12], in this report we present a new prototype implementing several new features.

- We discuss several strategies for evaluating XSPARQL queries according to our current implementation and present benchmark results for these.

In the remainder of this report, Section 2 will illustrate our main motivation to come up with a new language, by discussing some drawbacks of using existing technologies for transformations between RDF and XML. In Section 3 we will briefly review some main characteristics of XQuery and SPARQL, whereafter Section 4 presents the combination of both into XSPARQL, along with its formal semantics and our implementation. Section 5 discusses query optimisations in XSPARQL and experimental evaluation of these, before we wrap up with a discussion of related and future works in Sections 6 and 7.

2 Motivation: Lifting and Lowering

XML can be viewed as a tree-like data representation, 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 is – on an abstract level – just an unordered set of subject-predicate-object triples, as follows:

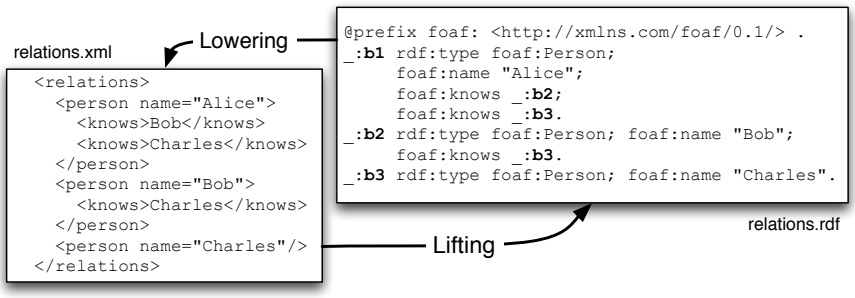


Figure 1: From XML to RDF and back: "lifting" and "lowering"

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

Besides the normative syntax to exchange RDF (RDF/XML [11]) there are various serialisation formats for RDF, such as RDFa [15], a format that allows one to embed RDF within (X)HTML, or non-XML representations such as the more readable Turtle [16] 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 XML and RDF formats has been acknowledged 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 [10, 17]. Both GRDDL and SAWSDL – although they acknowledge that other mechanisms could be used – use XSLT in their examples to perform lifting and lowering. In the following, let us illustrate drawbacks of this approach.

As a running example throughout this report we use a mapping between a customised XML format and RDF (using Turtle syntax here for illustration) as shown in Fig. 1. The task – in either direction – is 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 [18]ontology in the namespace `http://xmlns.com/foaf/0.1/` using prefix `foaf:`.[1]

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

RDF/XML [11] is the recommended syntax for RDF, using XML as the underlying representation model. This enables the use of XSLT and other XML tools to translate between RDF/XML and other XML formats, although such transformation is greatly complicated by the flexibility the RDF/XML format offers in serialising RDF graphs. Thus, tools that handle RDF/XML as XML data (and not as a sets of triples) need to take different possible representations into account. Fig. 2 shows four versions of a subset of the RDF data from our running example that represent the same FOAF data. Version (a) uses Turtle [16], a simple and readable textual format for RDF, inaccessible to pure XML processing tools though; the other three versions are all RDF/XML, ranging from concise (b) to verbose (d). These three RDF/XML variants look very different to XML tools, yet exactly the same to RDF tools. 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.

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

(a)

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

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

(b)

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

(c)

```
<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>
```

(d)

```
<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>
```

Figure 2: Different representations of the same RDF graph

Lifting For instance, the XSLT in Fig. 3(a) could be used to generate RDF/XML (in the format presented in Fig. 2(b)) from the relations.xml file in Fig. 1. However, this first attempt does not yet accomplish the intended transformation, loosing correlations between blank nodes. This is easy to see in the Turtle version of the result of this transformation (presented in Fig. 3(b)): while in our example names should uniquely identify a person, in this transformation the same person is potentially given several blank nodes. A proper lifting transformation is possible in XSLT, but at this point we only state that the corresponding XSLT script would need to be far more involved.

Lowering The simple XSLT stylesheet lowering.xsl in Fig. 4 is an attempt to perform the lowering task directly from RDF/XML. However, this XSLT will break if the input RDF/XML is in any other variant than the version in Fig. 2(b). 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 [19]. RDF Schema (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,[2] which are all stated as subproperties of `foaf:knows`:

```
@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".
```

As XPath and XSLT do not support ontological inference, we literally would need to implement an RDFS inference engine within XSLT, 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 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

[2] `http://vocab.org/relationship/`

```
<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) lifting.xsl

```
<rdf:RDF xmlns:rdf="...rdf-syntax-ns#"
         xmlns:foaf="...foaf/0.1/">
  <foaf:Person>
    <foaf:name>Alice</foaf:name>
    <foaf:knows><foaf:Person>
        <foaf:name>Bob</foaf:name>
      </foaf:Person></foaf:knows>
    <foaf:knows><foaf:Person>
        <foaf:name>Charles</foaf:name>
      </foaf:Person></foaf:knows>
  </foaf:Person>
  <foaf:Person>
    <foaf:name>Bob</foaf:name>
    <foaf:knows><foaf:Person>
        <foaf:name>Charles</foaf:name>
      </foaf:Person></foaf:knows>
  </foaf:Person>
  <foaf:Person>
    <foaf:name>Charles</foaf:name>
  </foaf:Person>
</rdf:RDF>
```

```
@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 of the XSL transform in RDF/XML (up) and Turtle (down)

Figure 3: Lifting attempt by XSLT

results XML format [20]. The advantage of such a two-step approach is firstly 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 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 described FOAF file above, 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 staring point, we argue that it can still be improved on several points. Firstly, the detour through SPARQL's XML query results format seems to be an unnecessary burden. Secondly, 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/XQuery sequentially, directly amenable to query optimisations. XSPARQL, the language proposed in the present report, 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. 6. 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.

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. 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*). Our goal is to combine these components for both languages in a unified language, XSPARQL, where XQuery's and SPARQL's heads and bodies

```
<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>
```

Figure 4: Lowering attempt by XSLT (lowering.xsl)

may be used interchangeably and even nested (this is presented schematically in Fig. 7). 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 [4, 14] and to [8, 21].

3.1 XQuery

XQuery consists mainly of so-called **FLWOR** expressions, denoting the body (**FLWO**) and the head (**R**) of a query. **`for`** clauses (**F**) can be used to declare variables looping over the XML sequences returned, for example, 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 **`for`** can be specified via the **`where`** part (**W**) and the **`order by`** clause **(O)**, resp. 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 [22], XQuery offers a flexible instrument for arbitrary XML transformations.

The lifting task of Fig. 1 can be solved with XQuery as shown in Fig. 5(a). 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 all nodes containing person names from the original file, for which a blank node needs to be created in variable `$p` (line 3). Looping over these nodes, we extract the actual names from the value of the `name` attribute in variable `$n`. Finally, we compute the position in the original XML tree and use it as blank node identifier in variable `$id`. The **`where`** clause (lines 12–14) filters out all but the last occurrence of a name, in order to avoid duplicate occurrences of the same name. The nested **`for`** (lines 19–31) again loops over persons in order to create nested `foaf:knows` elements. The difference here is that we select only the nodes corresponding to persons which are known by the person from the outer **`for`** loop (line 25). 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. Both these drawbacks will be alleviated by adding SPARQL to XQuery.

3.1.1 Semantics

Next, let us give a short overview of the XQuery Formal Semantics [14], on which we will also base XSPARQL's semantics; it is defined essentially via three types of rules: *normalisation rules*, *static typing rules* and *dynamic evaluation rules*. *Normalisation rules* are used to reduce an XQuery into the XQuery Core language – a subset of XQuery that aims to be easier to define, implement and optimise [23]. *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 meanwhile guaranteeing that the input is coherent with the type of the expression.

The *normalisation rules* are presented using mapping rules and, as an example, we present the following rule

```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
let $persons := //*[@name or ../knows]
return
<rdf:RDF>{
 for $p in $persons
 let $n := if( $p[@name] )
           then $p/@name else $p
 let $id :=count($p/preceding::*)
           +count($p/ancestor::*)
 where
  not(exists($p/following::*[
      @name=$n or data(.)=$n]))
 return
 <foaf:Person rdf:nodeId="b{$id}">
  <foaf:name>{data($n)}</foaf:name>{
  for $k in $persons
  let $kn := if( $k[@name] )
             then $k/@name else $k
  let $kid :=count($k/preceding::*)
             +count($k/ancestor::*)
  where
   $kn = data(//*[@name=$n]/knows) and
   not(exists($kn/../following::*[
       @name=$kn or data(.)=$kn]))
  return
  <foaf:knows>
   <foaf:Person rdf:nodeID="b{$kid}"/>
  </foaf:knows>
  }</foaf:Person>
 }</rdf:RDF>
```

(a) XQuery

```
declare namespace foaf="...foaf/0.1/";
declare namespace rdf="...-syntax-ns#";
let $persons := //*[@name or ../knows]
return
 for $p in $persons
 let $n := if( $p[@name] )
           then $p/@name else $p
 let $id :=count($p/preceding::*)
           +count($p/ancestor::*)
 where
  not(exists($p/following::*[
      @name=$n or data(.)=$n]))
 construct {
 _:b{$id} a foaf:Person;
          foaf:name {data($n)}.
  { for $k in $persons
  let $kn := if( $k[@name] )
             then $k/@name else $k
  let $kid :=count($k/preceding::*)
             +count($k/ancestor::*)
  where
   $kn = data(//*[@name=$n]/knows) and
   not(exists($kn/../following::*[
       @name=$kn or data(.)=$kn]))
  construct {
   _:b{$id} foaf:knows _:b{$kid}.
   _:b{$kid} a foaf:Person.
  }
 }
}
```

(b) XSPARQL

Figure 5: Lifting using XQuery and XSPARQL

from [14] that illustrates the normalisation of consecutive **`for`** clauses into XQuery Core:

$$\begin{array}{l} \left[\!\!\left[\begin{array}{l} \texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1 \\ \qquad \mathit{OptPosVar}_1\ \texttt{in}\ \mathit{Expr}_1, \\ \quad \ldots, \\ \quad \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n \\ \qquad \mathit{OptPosVar}_n\ \texttt{in}\ \mathit{Expr}_n \\ \mathit{ReturnClause} \end{array} \right]\!\!\right]_{\mathit{Expr}} \\ \qquad == \\ \texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPosVar}_1 \\ \quad \texttt{in}\ [\![\mathit{Expr}_1]\!]_{\mathit{Expr}}\ \texttt{return} \\ \ldots \\ \texttt{for}\ \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPosVar}_n \\ \quad \texttt{in}\ [\![\mathit{Expr}_n]\!]_{\mathit{Expr}} \\ [\![\mathit{ReturnClause}]\!]_{\mathit{Expr}} \end{array} \tag{1}$$

In normalisation rules, fixed-width font – for instance `for` – refer to specific keywords and *italic* font refer to productions in the XQuery Core grammar [14, Appendix A].

The *static type rules* and *dynamic evaluation rules* are presented using inference rules. For instance, the following static typing rule from [14] ensures that no expression has the `empty` type except the empty sequence and functions in the *fs* namespace that are applied to empty parentheses ():

$$\frac{\begin{array}{c} \text{statEnv} \vdash \mathit{Expr} : \mathit{Type} \\ \text{statEnv} \vdash \mathit{Type} <: \texttt{empty} \\ \textbf{not}\left(\begin{array}{l} \text{Expr is the empty parentheses ()} \ \textbf{or}\ \texttt{fn:data()} \\ \textbf{or}\ \text{any } \textit{fs} \text{ function applied to empty parentheses ()} \end{array} \right) \end{array}}{\text{A static type error is raised for expression } \mathit{Expr}} \tag{2}$$

```
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>
```

Figure 6: Lowering using XSPARQL

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

Figure 7: An overview of XQuery and SPARQL

Here, statEnv $\vdash$ *Expr* : *Type* and statEnv $\vdash$ *Type* <: `empty` (called judgements) hold when, in the static environment statEnv, *Expr* has type *Type* and *Type* is a subtype of `empty`, respectively. For all details of the XQuery Semantics we refer the reader to [14].

3.1.2 Typing

The XQuery and XPath Data Model (XDM) is described in [24] and is used to define the input to XQuery and the values of any XPath expression. For representing types throughout this report we will use the formal notation for types as described in [14, Section 2.4]. This representation of types is used for specification purposes only and is not exposed to the end user by XQuery. As an example (taken from [14, Section 2.4.4]), we present the following type:

```
<complexType name="UKAddress">
  <complexContent>
    <extension base="ipo:Address">
      <sequence>
        <element name="postcode" type="ipo:UKPostcode"/>
      </sequence>
      <attribute name="exportCode" type="positiveInteger" fixed="1"/>
    </extension>
  </complexContent>
</complexType>
```

which is represented as:

```
define type UKAddress extends ipo:Address {
  attribute exportCode of type positiveInteger,
  element postcode of type ipo:UKPostcode };
```

As described in [14, Section 8.3.1], it is possible to match a *Value* against a specific *Type* by using the judgement *Value* **matches** *Type*.

```
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 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 in XSPARQL

Figure 8: RDF-to-RDF mappings in SPARQL and in XSPARQL

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 to match parts of the dataset by specifying a *graph pattern*. Such patterns can be simple triple patterns possibly involving variables, URI references, and literals,[3] unions of graph patterns, optional patterns matching of parts of a graph, or patterns matching of named graphs, etc. More formally, graph patterns are defined recursively as follows:

Definition 2 (Graph Pattern)**.** *Let $\mathcal{V}$ be an infinite set of variables*

- *a set of triples $(s, p, o) \in (\mathcal{U} \cup \mathcal{L} \cup \mathcal{V})^3$ is called basic graph pattern (BGP)*
- *if P and P' are graph patterns then $(P\ P')$, $(P$ **`optional`** $P')$, $(P$ **`union`** $P')$ are graph patterns.*
- *if P is a graph pattern and $i(\mathcal{U} \cup \mathcal{V})$, then $($**`graph`** $i\ P)$ is a graph pattern.*
- *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.

The evaluation semantics of SPARQL consists of computing a sequence of *pattern solutions*, i.e., sets of bindings for the variables in these patterns, matching them against the graphs in the dataset.

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 the RDF merge [7] of several source graphs specified in consecutive **`from`** clauses, which could involve renaming of blank nodes at the pure XML level, comes for free in SPARQL. Pattern solutions can be ordered or sliced using solution modifiers (**M**) **`order by`**, **`limit`**, and **`offset`**.

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 [25] format to `foaf:name` is given by the SPARQL query in Fig. 8a. 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 pattern solution in the result graph.

Let us remark on that SPARQL does not offer the creation of new values by function calls in the head which on the contrary comes for free in XQuery by offering the full range of XPath/XQuery built-in functions [22]. Due to this, the query in Fig. 8b which attempts to merge family names and given names into a single `foaf:name` by 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.

[3] Note that we do not allow blank nodes in graph patterns, and thus don't consider them in our definitions. As well known, this restriction does not affect the expressivity of SPARQL (implicit in [21]), since blank nodes in query patterns can always be replaced equivalently with variables). See discussion in Section 4.2.1 below.

3.2.1 Semantics

The semantics of SPARQL is defined by means of evaluation rules which are presented in [8, Section 12.5]. Here we only give an overview of the notion of Basic Graph Pattern (BGP) matching, that we will later use to define the semantics of XSPARQL.

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

Definition 3 (Solution Mapping)**.** *A* solution mapping *[8, Section 12.1.6] is a partial function that assigns SPARQL variables to RDF terms. The domain of a solution mapping μ, $dom(\mu)$, is the set of variables for which μ is defined.*

Two solution mappings are considered *compatible* if, for all variables that are present in both mappings, the value each mapping assigns to the variable is the same.

Definition 4 (Compatible Mappings)**.** *Let μ_1 and μ_2 be solution mappings, μ_1 and μ_2 are* compatible *if $\forall v \in dom(\mu_1) \cap dom(\mu_2), \mu_1(v) = \mu_2(v)$.*

Finally, the definition of BGP matching from [8, Section 12.3], specifies the solutions to a query:

Definition 5 (Basic Graph Pattern Matching)**.** *Considering G an RDF graph and B a BGP, we say μ is a* solution *for B against the active graph G, if there exists a solution mapping P such that the substitution of variables in B according to P, denoted $P(B)$, is a subgraph of G and μ is the restriction of P to the query variables $Vars(B)$ in B.*

This definition of pattern solutions is extended to more complex SPARQL patterns (including **`union`**, **`optional`**, **`graph`**, **`filter`**, etc.) by the SPARQL algebra [8, Section 12.4], such that the **`where`** clause of every SPARQL query – i.e., any **DWM** body – returns a *list* of pattern solutions. For details, we refer the reader to the W3C SPARQL specification.

In a **`construct`** query, the pattern 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 pattern solution, blank nodes occurring in a **`construct`** template are replaced by new blank nodes with "fresh" 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 components 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 (slightly abusing nomenclature) **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 **`for`** clause, but instead of allowing a single variable(which is assigned to the results of an XPath expression), the new clausesupports a white space separated list of variables (*varlist*). Each variable in *varlist* 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`** statements (**C**) alternatively to XQuery's native **`return`** (**R**).

(iii) Different forms of nesting are allowed, such as subqueries constructing RDF graphs appearing in **`let`** assignments or within SPARQL style **`from`** clauses, or value construction within SPARQL-style **`construct`** clauses.

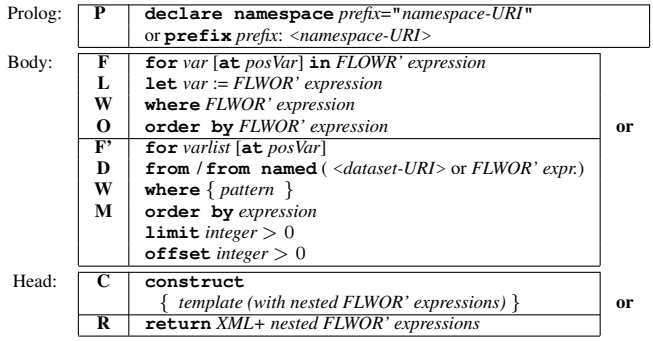


Figure 9: Schematic view of XSPARQL

(a)

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

(b)

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

(c)

```
[11] NamedGraphClause      ::=  "named" ( SourceSelector  | IRIref "(" "{" FLWORExpr' "}" ")"
                                          | "{" FLWORExpr' "}" "(" "{" FLWORExpr' "}" ")" )
[12] SourceSelector        ::=  IRIref | "{" FLWORExpr' "}"
```

Figure 10: (a) XSPARQL core syntax elements, extending [4, Appendix A] (b) modified ConstructTemplate syntax elements, extending [8, Appendix A] (c) modified DatasetClause syntax elements, extending [8, Appendix A]

These modifications allow us to reformulate the lifting query of Fig. 5(a) into its slightly more concise XSPARQL version of Fig. 5(b). 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. 6.

4.1 Syntax

In more detail, the XSPARQL syntax is an extension of the grammar rules in [4]. Fig. 9 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 [8] and XQuery [4] and mark any 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. 10.[4] The newly introduced `SparqlForClause` (rule 33a) is similar to an XQuery **`for`**-loop that can be used to iterate over SPARQL results. 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 [8], 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. 10(b).

The rules for `DatasetClause` from the SPARQL syntax are also extended (as presented in Fig. 10(c)), i.e., we allow for graphs in a SPARQL dataset to be specified by a nested `FLWORExpr'` expression, that must evaluate to the URI location of the RDF graph or to the RDF graph itself. Furthermore, in order to use constructed graphs as named graphs, we allow the name of the graph to be specified in the following manner: `uriOrExpr(graphExpr)`, where `uriOrExpr` represents the URI to be used as the name of the graph (or an expression that evaluates to a URI) and `graphExpr` is an expression that evaluates to an RDF graph.

[4] For the interested reader, the full XSPARQL grammar can be found at `http://xsparql.deri.org/doc/grammar.html`.

Note that – in analogy to SPARQL's "**select** *" shortcut – we allow to write "**for** *" in place of "**for** [*list of all unbound variables appearing in the* **where** *clause*]" for SparqlForClauses; as syntactic sugar this is also the default value for the **F'** clause whenever a SPARQL-style **where** clause is found and a corresponding **F'** clause is missing. For example, the query from Fig. 8b, corresponds to an implicit **for** $P $N $F clause. Please note that for SparqlForClauses we do not allow XQuery *QNames* as variable names (for further details the reader is referred to [14, 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 or any other SPARQL **construct** queries remain valid syntax for XSPARQL, i.e., we extend both the XQuery and SPARQL syntaxes conservatively. For more specific details on the XSPARQL syntax we refer the reader to the full grammar, cf. Footnote 4 on p. 12.

4.2 Semantics

Next, we will define the semantics of XSPARQL. After introducing some new types used in the semantics and an extension to the normalisation rules of XQuery **for** clauses, we will turn to extending the notion of Basic Graph Pattern matching (Section 4.2.1) to make SPARQL clauses aware of the bindings for variables from XQuery. Then, we present the semantics of the newly introduced expressions: SparqlForClause (Section 4.2.2) and Construct-Clause (Section 4.2.3), based on XQuery's formal semantics [14], 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 in [24]) with the following new typesthat 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;
2. the PatternSolution type consists of a set of pairs (variableName,RDFTerm) representing SPARQL variable bindings;
3. the RDFGraph will be the type of **construct** expressions; and
4. the RDFDataset as the type for DatasetClauses.

Throughout this report we will use the formal notation for types as described in [14, Section 2.4]. 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 XML Schema of the SPARQL Query Results XML Format, available at http://www.w3.org/2007/SPARQL/result.xsd.

```
define type URI-reference restricts xs:anyURI;

define element uri of type xs:string;

define element bnode of type xs:string;

define type literal {
  attribute datatype of type URI-reference?,
  attribute lang of type xml:lang? };
```

```
define type RDFTerm {
 uri | bnode | literal };

define type Binding {
  element variable of type xs:string,
  element binding of type RDFTerm };

define type Bindings {
  element result of type Binding* };

define type PatternSolution {
  element results of type Bindings };
```

The `RDFGraph` type corresponds to a sequence of `RDFTriples` which are in turn a complex type composed of `subject`, `predicate` and `object`.

```
define element graph of type RDFGraph;

define type RDFGraph {
  element triples of type RDFTriples };

define type RDFTriples {
  element triple of type RDFTriple* };

define type RDFTriple {
  element subject of type RDFTerm,
  element predicate of type RDFTerm,
  element object of type RDFTerm };
```

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`.

```
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 };
```

XQuery `for` normalisation In accordance to the SPARQL semantics, blank nodes in **`construct`** expressions need to be distinctly instantiated for any solution binding matching the body, i.e., for every pattern solution for the **`where`** clause a new blank node identifier needs to be created in the resulting graph. To ensure this behaviour in XSPARQL **`construct`** clauses, we will use so-called position variables[5] from XQuery in **`for`** loops to generate these new blank node identifiers, i.e., we introduce position variables in any XQuery **`for`**-loops 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 core **`for`**-loops (cf. Section 3.1):

$$\left[\!\!\left[\begin{array}{l} \texttt{for}\ \$\mathit{VarName}_1\ \mathit{OptTypeDeclaration}_1\ \mathit{OptPositionalVar}_1\ \texttt{in}\ \mathit{Expr}_1, \\ \quad \cdots, \\ \quad \$\mathit{VarName}_n\ \mathit{OptTypeDeclaration}_n\ \mathit{OptPositionalVar}_n\ \texttt{in}\ \mathit{Expr}_n \\ \mathit{ReturnClause} \end{array} \right]\!\!\right]_{Expr} \quad (3)$$

$$==$$

[5] Position variables are variables that appear in an XQuery **`for`** clauses after the optional **`at`** keyword – cf. Figure 7a – and bind to an integer indicating the current "position" in the for-loop.

$$==$$

$$\texttt{for}\ \$VarName_1\ OptTypeDeclaration_1\ [\![OptPositionalVar_1]\!]_{PosVar}\ \texttt{in}\ [\![Expr_1]\!]_{Expr}$$
$$\texttt{return}$$
$$\quad \ldots$$
$$\quad \texttt{for}\ \$VarName_n\ OptTypeDeclaration_n\ [\![OptPositionalVar_n]\!]_{PosVar}\ \texttt{in}\ [\![Expr_n]\!]_{Expr}$$
$$\quad [\![ReturnClause]\!]_{Expr}$$

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.* [14, Section 4.12.6]):[6]

$$[\![]\!]_{PosVar} == \texttt{at}\ \$fs{:}new \tag{4}$$

In case a positional variable is already present it is not changed:

$$[\![\texttt{at}\ \$PosVar]\!]_{PosVar} == \texttt{at}\ \$PosVar \tag{5}$$

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`** loops. The static type rules for the **`for`** expression (cf. [14, Section 4.8.2]) need to be extended accordingly to store these positional variables, similar to the rules for `SparqlForClause`s in Section 4.2.2 below.

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 namespace declarations to XQuery `declare` expressions by the following normalisation rules:

$$\begin{gathered}[\![\texttt{prefix}\ NCName\texttt{:}\quad \texttt{<}URILiteral\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare namespace}\ NCNAME\ \texttt{=}\ URILiteral\ \texttt{;}]\!]_{Expr}\end{gathered} \tag{6}$$

$$\begin{gathered}[\![\texttt{base}\ \texttt{<}URILiteral\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare base-uri}\ URILiteral\ \texttt{;}]\!]_{Expr}\end{gathered} \tag{7}$$

$$\begin{gathered}[\![\texttt{base}\ \texttt{<}URILiteral\texttt{>}]\!]_{Expr} \\ == \\ [\![\texttt{declare base-uri}\ URILiteral\ \texttt{;}]\!]_{Expr}\end{gathered} \tag{8}$$

4.2.1 XSPARQL BGP Matching

In this section we extend the notion of Basic Graph Pattern matching (described in [8, Section 12.3]) in order 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. Obviously, we can consider the `varValue` component as just defining a set of bindings in the spirit of SPARQL solution mappings. Along these lines, we will also call the `varValue` component of the *dynamic environment* in which a SPARQL graph pattern $pattern$ occurring within XSPARQL is executed the *XML instance mapping* of $pattern$.

Recapitulating the definition of Basic Graph Pattern Matching presented in Section 3.2.1, μ is a solution for a BGP B from G if there is a solution mapping P such that the substitution of variables in B according to P, denoted $P(B)$, is a subgraph of G and μ is the restriction of P to the query variables $Vars(B)$ in B.

[6] The $\$fs{:}new$ variable exhibits the same behaviour as in the semantics of XQuery, where it is – strictly speaking – not clear how such new variable can be generated without having to process the complete query. We asked the W3C XQuery WG for clarification on this matter, cf. `http://lists.w3.org/Archives/Public/public-qt-comments/2011Mar/0024.html`

Definition 6 (Extended solution mapping)**.** *An* extended solution mapping *is a solution mapping* compatible *with the* XML instance mapping *of the graph pattern.*

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

Definition 7 (XSPARQL BGP matching)**.** *Considering G an RDF graph and B a BGP, we say μ is a* solution *for B against the active graph G, if there exists an extended solution mapping P such that $P(B)$ is a subgraph of G and μ is the restriction of P to the query variables in B.*

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 B will not only match the input graph G but also respect any bindings for variables in the dynamic environment.

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 (cf. [8, Section 12]): the 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 XML 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. 6 blank nodes bound in the outer **`for`** loop to the variable $\$Person$ will be injected into the inner loop. 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 behavior, 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 Section 3.2 and detailed in [8]. The static type signature of this function is:

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

Any nested `FLWORExpr'` expression within a `DatasetClause` must evaluate to 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 report, we assume obtaining the RDF graph just by dereferencing the URI via HTTP.

4.2.2 `SparqlForClause`

The semantics of `SparqlForClause` (Rule 33a, Figure 10(a)) is defined by the following normalisation rules, static type analysis rules and dynamic evaluation rules. We start with normalisation rules for `SparqlForClause`s with implicit variable selection (by means of "**`for`** *") and with explicitly stated variables:

$$\left[\!\!\left[\begin{array}{l} \texttt{for} * \ OptDatasetClause \ SparqlWhereClause \\ SolutionModifier \ ReturnClause \end{array} \right]\!\!\right]_{Expr} \quad == \quad \left[\!\!\left[\begin{array}{l} \texttt{for} \ [\![SparqlWhereClause]\!]_{vars} \ OptDatasetClause \\ SparqlWhereClause \ SolutionModifier \ ReturnClause \end{array} \right]\!\!\right]_{Expr} \tag{9}$$

The normalisation rule $[\![\cdot]\!]_{vars}$ determines all statically *unbound variables* present in the $SparqlWhereClause$, i.e., returns a whitespace separated list of all variables in the $SparqlWhereClause$ 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:

$$\begin{array}{l} \left[\!\!\left[\begin{array}{l} \texttt{for}\ \$VarName_1 \ldots \$VarName_n \\ OptDatasetClause\ SparqlWhereClause \\ SolutionModifier\ ReturnClause \end{array} \right]\!\!\right]_{Expr} \\ \qquad == \\ \begin{array}{l} \texttt{for}\ \$VarName_1 \ldots \$VarName_n\ [\![]\!]_{PosVar} \\ \qquad OptDatasetClause\ SparqlWhereClause \\ [\![ReturnClause]\!]_{Expr} \end{array} \end{array} \tag{10}$$

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

$$\frac{\begin{array}{r} \text{statEnv.posVars} = (PosVar_1, \cdots, PosVar_n) \\ \text{statEnv + posVars}(PosVar_1, \cdots, PosVar_n, Var_{pos}) \\ \text{+ varType}(Var_{pos} \Rightarrow \texttt{xs:integer} \\ Var_1 \Rightarrow RDFTerm; \\ \cdots; \\ Var_n \Rightarrow RDFTerm \\ ) \vdash ReturnExpr : Type \end{array}}{\begin{array}{r} \texttt{for}\ \$Var_1 \cdots \$Var_n\ \textbf{at}\ \$Var_{pos} \\ \text{statEnv} \vdash DatasetClause\ SparqlWhereClause \\ SolutionModifier\ \texttt{return}\ ReturnExpr : Type* \end{array}} \tag{11}$$

The static type rule for a SparqlForClause without an explicit *DatasetClause* is analogous.

Dynamic Evaluation For the dynamic evaluation we have to introduce a new dynamic environment component called *activeDataset*, that will be used to evaluate *SparqlWhereClause*s. Initially, this component is empty (or set to a system default) and can be changed by a *DatasetClause* appearing within a SparqlForClause, as defined in the following rules. We further introduce 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 PatternSolution PS. If var is not bound in PS, the empty sequence is returned.

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

fs:sparql The *fs:sparql* function corresponds to an adapted version of the *eval* function (as described in the SPARQL specification [8, Section 12.5]), that evaluates graph patterns implementing our extended notion of Basic Graph Pattern Matching (described in Section 4.2.1). The static type signature of this function is:

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

Here, the graph pattern parameter from the SPARQL *eval* function corresponds to the pattern from the $SparqlWhereClause after conversion of $solutionModifiers (cf. [8, Section 12.2.3]) from the *fs:sparql* function. The result of this $eval$ function consists of a solution sequence, which can be translated directly into an XQuery sequence of XML elements of type PatternSolution.

We can now define the dynamic evaluation rules for a SparqlForClause. Intuitively these rules state that the return expression $ReturnExpr$ will be executed for each PatternSolution that is returned. The following two

dynamic rules specify the evaluation of a `SparqlForClause` with an explicit *DatasetClause*:

$$\frac{\begin{array}{l}\text{dynEnv} \vdash \mathit{fs{:}dataset}(\mathit{DatasetClause}) \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs{:}sparql}(\mathit{Dataset}, \\ \qquad \mathit{SparqlWhereClause}, \\ \qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv + activeDataset}(\mathit{Dataset}) \\ \qquad \text{+ varValue}(\mathit{Var}_{pos} \Rightarrow 1 \\ \qquad\qquad \mathit{Var}_1 \Rightarrow \mathit{fs{:}value}(\mathit{PS}_1, \mathit{Var}_1); \\ \qquad\qquad \cdots; \\ \qquad\qquad \mathit{Var}_n \Rightarrow \mathit{fs{:}value}(\mathit{PS}_1, \mathit{Var}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \qquad\qquad \vdots \\ \text{dynEnv + activeDataset}(\mathit{Dataset}) \\ \qquad \text{+ varValue}(\mathit{Var}_{pos} \Rightarrow n \\ \qquad\qquad \mathit{Var}_1 \Rightarrow \mathit{fs{:}value}(\mathit{PS}_m, \mathit{Var}_1); \\ \qquad\qquad \cdots; \\ \qquad\qquad \mathit{Var}_n \Rightarrow \mathit{fs{:}value}(\mathit{PS}_m, \mathit{Var}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\begin{array}{ll} & \texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \texttt{ at } \$\mathit{Var}_{pos}\ \mathit{DatasetClause} \\ \text{dynEnv} \vdash & \mathit{SparqlWhereClause}\ \mathit{SolutionModifier} \\ & \texttt{return } \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}} \tag{12}$$

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 SPARQL expression does not yield any solutions, i.e., evaluates to an empty sequence, the overall result will also be the empty sequence:

$$\frac{\begin{array}{l}\text{dynEnv.activeDataset} \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs{:}sparql}(\mathit{Dataset}, \mathit{SparqlWhereClause}, \\ \qquad \mathit{SolutionModifier}) \Rightarrow () \end{array}}{\begin{array}{ll} & \texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \texttt{ at } \$\mathit{Var}_{pos} \\ \text{dynEnv} \vdash & \mathit{DatasetClause}\ \mathit{SparqlWhereClause} \\ & \mathit{SolutionModifier}\ \texttt{return}\ \mathit{ReturnExpr} \Rightarrow () \end{array}} \tag{13}$$

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

$$\frac{\begin{array}{l}\text{dynEnv.activeDataset} \Rightarrow \mathit{Dataset} \\ \text{dynEnv} \vdash \mathit{fs{:}sparql}(\mathit{Dataset} \\ \qquad \mathit{SparqlWhereClause}, \\ \qquad \mathit{SolutionModifier}) \Rightarrow \mathit{PS}_1, \ldots, \mathit{PS}_m \\ \text{dynEnv + varValue}(\mathit{Var}_{pos} \Rightarrow 1 \\ \qquad\qquad \mathit{Var}_1 \Rightarrow \mathit{fs{:}value}(\mathit{PS}_1, \mathit{Var}_1); \\ \qquad\qquad \ldots; \\ \qquad\qquad \mathit{Var}_n \Rightarrow \mathit{fs{:}value}(\mathit{PS}_1, \mathit{Var}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \qquad\qquad \vdots \\ \text{dynEnv + varValue}(\mathit{Var}_{pos} \Rightarrow n \\ \qquad\qquad \mathit{Var}_1 \Rightarrow \mathit{fs{:}value}(\mathit{PS}_m, \mathit{Var}_1); \\ \qquad\qquad \ldots; \\ \qquad\qquad \mathit{Var}_n \Rightarrow \mathit{fs{:}value}(\mathit{PS}_m, \mathit{Var}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m \end{array}}{\begin{array}{ll} & \texttt{for } \$\mathit{Var}_1 \cdots \$\mathit{Var}_n \texttt{ at } \$\mathit{Var}_{pos} \\ \text{dynEnv} \vdash & \mathit{SparqlWhereClause}\ \mathit{SolutionModifier} \\ & \texttt{return}\ \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m \end{array}} \tag{14}$$

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

Figure 11: Example result of $[\![\cdot]\!]_{normaliseTemplate}$ applied to the `ConstructTemplate'` in Fig. 8b

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.

4.2.3 `ConstructClause`

We now define the semantics of the `ConstructClause` (Rule 33b, Figure 10(a)) by means of normalisation rules. SPARQL stand-alone **`construct`** queries (as described in Section 4.1) are normalised into **`construct`** queries with a surrounding **`for`**-clause:

$$\begin{array}{c} \left[\!\!\left[\begin{array}{l} \texttt{construct}\ \mathit{ConstructTemplate'}\ \mathit{DatasetClause} \\ \quad \mathit{SparqlWhereClause}\ \mathit{SolutionModifier} \end{array} \right]\!\!\right]_{Expr} \\ == \\ \left[\!\!\left[\begin{array}{l} \texttt{for}\ *\ \mathit{DatasetClause} \\ \mathit{SparqlWhereClause}\ \mathit{SolutionModifier} \\ \texttt{construct}\ \mathit{ConstructTemplate'} \end{array} \right]\!\!\right]_{Expr} \end{array} \tag{15}$$

The resulting query will be further rewritten according to normalisation rule (9) above. As introduced in Section 4.1, we allow nested XSPARQL expressions in subject, predicate and object positions of `ConstructTemplate'`. These nested expressions are identified by the following shortcuts: `{Expr}`, `<{Expr}>` and `_:{Expr}`, that construct, respectively, elements of type `literal`, `uri` and `bnode`.

Similar to the normalisation rule for stand-alone `ReturnClause`s presented in [14, Section 4.8.1], the following normalisation rule transforms **`construct`** clauses into XQuery `ReturnClause`s.

$$\begin{array}{c} [\![\ \texttt{construct}\ \mathit{ConstructTemplate'}\ ]\!]_{Expr} \\ == \\ \texttt{return} \\ fs{:}\mathit{evalTemplate}\left([\![\mathit{ConstructTemplate'}]\!]_{normaliseTemplate}\right) \end{array} \tag{16}$$

In the following we assume that *ConstructTemplate'* is a simple "`.`" separated list of *Subject*, *Predicate* and *Object*. The $[\![\cdot]\!]_{normaliseTemplate}$ 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. [16, Section 2.3]):

$$\begin{array}{c} \left[\!\!\left[\begin{array}{l} \mathit{Subject}\ \mathit{Pred}_1\ \mathit{Obj}_1; \\ \quad \ldots; \mathit{Pred}_n\ \mathit{Obj}_n \end{array} \right]\!\!\right]_{normaliseTemplate} \\ == \\ \mathit{Subject}\ \mathit{Pred}_1\ \mathit{Obj}_1\ \ \ldots\ \ \mathit{Subject}\ \mathit{Pred}_n\ \mathit{Obj}_n \end{array} \tag{17}$$

More normalisation rules for other Turtle shortcuts allowed in the SPARQL `ConstructTemplate'` syntax similar to this one are not presented here. Since anonymous blank nodes can be written in numerous ways in Turtle, our $[\![\cdot]\!]_{normaliseTemplate}$ 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'`.

As an example we show the normalisation of the `ConstructTemplate` in Fig. 8b in Figure 11.

fs:evalTemplate The *fs:evalTemplate* function is a new built-in function that ensures the created RDF graph is valid and rewrites any blank nodes inside of `ConstructTemplate`s to comply with the SPARQL semantics (as described in Section 4.2.1). The auxiliary *fs*:*validTriple* function checks if each triple is valid according to the RDF semantics and is defined by rules (19) and (20). The static type signatures of these functions are defined as:

```
fs:evalTemplate($template as RDFTerm*) as RDFGraph

fs:validTriple($subject as RDFTerm, $predicate as RDFTerm, $object as RDFTerm)
    as RDFTriple
```

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

$$\frac{\begin{array}{c}\text{dynEnv} \vdash fs{:}validTriple\,(Subj_1, Pred_1, Obj_1) \Rightarrow Triple_1 \\ \ldots \\ \text{dynEnv} \vdash fs{:}validTriple\,(Subj_n, Pred_n, Obj_n) \Rightarrow Triple_n\end{array}}{\begin{array}{rl}\text{dynEnv} \vdash & fs{:}evalTemplate \begin{pmatrix} Subj_1\ Pred_1\ Obj_1 \\ \ldots \\ Subj_n\ Pred_n\ Obj_n \end{pmatrix} \\ \Rightarrow & \begin{array}{l}\texttt{<triples>} \\ \quad Triple_1 \ldots Triple_n \\ \texttt{</triples>}\end{array}\end{array}} \tag{18}$$

The following dynamic evaluation rule for the $fs{:}validTriple$ function checks if a triple is valid according to the RDF semantics. As described in [14, Section 8.3.1], the judgement **matches**, e.g. "*Value* **matches** *Type*" matches *Value* against specific *Type*.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash fs{:}bnode(Subject) \Rightarrow ValueS \\ \text{statEnv} \vdash ValueS\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode}) \\ \text{dynEnv} \vdash Predicate \Rightarrow ValueP \\ \text{statEnv} \vdash ValueP\ \textbf{matches}\ \texttt{uri} \\ \text{dynEnv} \vdash fs{:}bnode(Object) \Rightarrow ValueO \\ \text{dynEnv} \vdash ValueO\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode} \mid \texttt{literal})\end{array}}{\begin{array}{rl}\text{dynEnv} \vdash & fs{:}validTriple(Subject,\ Predicate,\ Object) \\ \Rightarrow & \begin{array}{l}\texttt{<triple>} \\ \quad \texttt{<subject>}\, ValueS \,\texttt{</subject>} \\ \quad \texttt{<predicate>}\, ValueP \,\texttt{</predicate>} \\ \quad \texttt{<object>}\, ValueO \,\texttt{</object>} \\ \texttt{</triple>}\end{array}\end{array}} \tag{19}$$

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.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash fs{:}bnode(Subject) \Rightarrow ValueS \\ \text{dynEnv} \vdash Predicate \Rightarrow ValueP \\ \text{dynEnv} \vdash fs{:}bnode(Object) \Rightarrow ValueO \\ \text{dynEnv} \vdash \textbf{not} \begin{pmatrix} ValueS\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode})\ \textbf{and} \\ ValueP\ \textbf{matches}\ \texttt{uri}\ \textbf{and} \\ ValueO\ \textbf{matches}\ (\ \texttt{uri} \mid \texttt{bnode} \mid \texttt{literal}\ ) \end{pmatrix}\end{array}}{\text{dynEnv} \vdash fs{:}validTriple(Subject,\ Predicate,\ Object) \Rightarrow ()} \tag{20}$$

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 **for**-loop (XQuery and `SparqlForClauses`) by assigning them position variables (as described in Section 4.2), 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 $fs{:}bnode$

```
prefix vc: <...vcard-rdf/3.0#>
prefix foaf: <...foaf/0.1/>
_:b vc:Given "Axel" .
_:b vc:Family "Polleres" .
```

Figure 12: Example input for query in Figure 8b

```
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)}.)
```

Figure 13: Query of Fig 8b after normalisation

function takes care of skolemising blank nodes: if the argument of this function is of type `bnode` a new blank node identifier is generated (rule (21)), otherwise the function returns its argument unchanged (represented by rule (22)). The generation of the new identifier is done by the *fs:skolemConstant* function based on the specified label of the blank node and on any positional variables in the dynamic environment.

$$\frac{\begin{array}{c}\text{dynEnv} \vdash \mathit{RDFTerm}\ \mathbf{matches}\ \texttt{bnode} \\ \text{dynEnv} \vdash \mathit{RDFTerm} \Rightarrow \mathit{ValueR} \\ \text{statEnv.posVars} = (\mathit{PosVar}_1, \cdots, \mathit{PosVar}_n) \\ \text{dynEnv.varValue}(\mathit{PosVar}_1) = \mathit{PosValue}_1 \\ \cdots \\ \text{dynEnv.varValue}(\mathit{PosVar}_n) = \mathit{PosValue}_n \\ \text{dynEnv} \vdash \mathit{fs{:}skolemConstant} \begin{pmatrix} \mathit{ValueR}, \\ \mathit{PosValue}_1, \\ \cdots, \\ \mathit{PosValue}_1, \end{pmatrix} \Rightarrow \mathit{ValueRS}\end{array}}{\text{dynEnv} \vdash \mathit{fs{:}bnode}(\mathit{RDFTerm}) \Rightarrow \texttt{<bnode>}\mathit{ValueRS}\texttt{</bnode>}} \quad (21)$$

$$\frac{\text{dynEnv} \vdash \mathit{RDFTerm}\ \mathbf{matches}\ (\texttt{uri} \mid \texttt{literal})}{\text{dynEnv} \vdash \mathit{fs{:}bnode}(\mathit{RDFTerm}) \Rightarrow \mathit{RDFTerm}} \quad (22)$$

4.2.4 Example of XSPARQL Semantics Evaluation

As an example we show the application of the presented evaluation semantics to the sample query from Fig. 8b. The example query features both, the new `SparqlForClause` as well as the new `ConstructClause`. We assume the input graph `vc.rdf` as given in Figure 12. 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 (7)). After that, the whole **`construct`** query is rewritten to a `SparqlForClause` by Rule (15). Then the `for *` is expanded according to Rule (9) and the resulting `SparqlForClause` is then handled by Rule (10). Rule (4) then adds a new positional variable (e.g. $\$pos$). Finally the `ReturnClause` is normalised by Rule (16). The whole normalisation phase results in the query given in Fig 13.

Static Type Analysis By Rule (11) the variables occurring after the `SparqlWhereClause`, 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 `ReturnClause`, which in turn inherits its type from the function *fs:evalTemplate* which is `RDFGraph`.

$P	$N	$F
_:gen	"Axel"	"Polleres"

Table 1: Result of *fs:sparql*

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

Figure 14: Result of example query using the `RDFGraph` type

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 (14) the `SparqlWhereClause` is evaluated using the *fs:sparql* function with the active dataset, as just initialised, the `SparqlWhereClause` as given in the query, and empty `SolutionModifier`s. The *fs:sparql* function call results in a sequence of `PatternSolution`s (in our case a singleton solution) as given in Table 1. Next, the same rule extracts the variable bindings for all variables, by using the *fs:value* function, and assignes them to the corresponding XQuery variables, typed as `RDFTerm`. After that the `ReturnExpression` is evaluated, using the just initialised variables. The *fs:evalTemplate* 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 (21)) 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:evalTemplate* function eventually returns the result of the single *fs:validTriple* function call, thus the result of the whole query being an "XML object" of type `RDFGraph` as shown in Figure 14. Serialised to Turtle the query result, including QName expansion, is the following: `_:gen_1 <http://...foaf/0.1/name> "Axel Polleres"`.

4.3 Implementation

In this section we present a prototype implementation of the XSPARQL language that consists of translating each XSPARQL expression into XQuery expressions with interleaved calls to a SPARQL engine. The architecture of our implementation (shown in Fig. 15) consists of three main components: (1) a query rewriter, which turns an XSPARQL query into an XQuery; (2) a SPARQL engine, for querying RDF from within the rewritten XQuery; and (3) an XQuery engine for computing the result document.

It is possible to use any XQuery and SPARQL engine to execute our rewritten query with the exception of the blank node handling presented in Section 4.2.1. For the implementation of this feature, we must rely on custom built code that uses the ARQ[7] Java API and thus needs to use an XQuery engine that allows us to call Java code from XQuery. We have tested two XQuery implementations that provide this feature: Saxon [26], which we also used for the experimental evaluation of this report and Qexo [27]. When calling the SPARQL engine, the current implementation relies on the SPARQL XML results format [28], to pass results back to XQuery. Note that by using the ARQ Java API and Saxon, we could actually carry the bindings of SPARQL results directly to XQuery in a more integrated implementation; however, for the moment we leave this as a potential further optimisation to future work.

For presenting our implementation we start by discussing the implementation of the types presented in Section 4.2

[7] `http://jena.sourceforge.net/ARQ/`

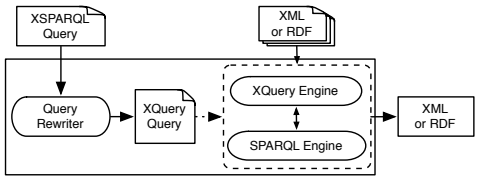


Figure 15: XSPARQL implementation architecture

and then we present – as opposed to the abstract, formal semantics – more concrete normalisation rules that realise `SparqlForClause`s and `ConstructClause`s in our implementation.

4.3.1 Type definitions

In our implementation, we do not need to implement all newly introduced types as custom types in XQuery, but we make direct use of the XML Schema of the SPARQL Query Results XML Format,[8] where the `sr:binding` type has a direct correspondence with our abstract `RDFTerm` type. Further, we serialise `RDFGraph`s in plain text as `xs:string`s using Turtle serialisation. The remaining types `RDFDataset` and `RDFNamedGraph` are adapted accordingly.

4.3.2 `SparqlForClause`

The normalisation rules presented in this section handle deferring SPARQL queries to an existing engine and extracting the bindings for the SPARQL variables from the returned SPARQL XML results document. We implement the $fs{:}sparql$ function by translating it to a SPARQL **`select`** query and, for the implementation of the *activeDataset* in nested `SparqlForClause`s (as described in Section 4.2.1) we rely on external Java code to call the ARQ API in such a way that blank node labels are preserved over consecutive calls on the same dataset. The custom function $sparqlCall$ – which replaces $fs{:}sparql$ in our implementation – implements this API call; it is called with a SPARQL **`select`** query, and returns the result in SPARQL's XML result format.

The normalisation of a single `SparqlForClause` is presented in Rule (23). Each `SparqlForClause` is translated into a SPARQL "**`select`** $\star$" query. Then, for each solution returned by the SPARQL engine, we assign the value in that solution to the corresponding XQuery variable.

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \texttt{for}\ \$VarName_1 \cdots \$VarName_n\ DatasetClause \\ SparqlWhereClause\ SolutionModifier\ ReturnClause \end{array} \right]\!\!\right]_{Expr} \\
\qquad == \\
\texttt{let}\ \$queryresult\ \texttt{:=} \\
\quad sparqlCall \left(\begin{array}{l} \texttt{select}\ \$VarName_1 \cdots \$VarName_n \\ \qquad DataSetClause \\ \qquad SparqlWhereClause \\ \qquad SolutionModifier \end{array} \right) \\
\texttt{for}\ \$result\ \text{at}\ \$result\_pos\ \texttt{in}\ \$queryresult\texttt{//sr:result} \\
\quad [\![VarName_1 \cdots VarName_n]\!]_{SparqlResult(\$result)} \\
[\![ReturnClause]\!]_{Expr}
\end{array}
\tag{23}
$$

Here, the auxiliary mapping rule $[\![\cdot]\!]_{SparqlResult(res)}$ extracts the solution variable bindings from the SPARQL XML

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

results format:

$$
\begin{array}{c}
[\![\$Var_1 \cdots \$Var_n]\!]_{SparqlResult(\$res)} \\
== \\
\texttt{let}\ \$Var_1 := \$res/\texttt{sr:binding}[@name = \$Var_1]/* \\
\cdots \\
\texttt{let}\ \$Var_n := \$res/\texttt{sr:binding}[@name = \$Var_n]/*
\end{array}
\tag{24}
$$

4.3.3 `ConstructClause`

As for the construction of RDF graphs (the return of a `ConstructClause`), we need to ensure that each element of an RDF triple is syntactically valid. Along with the rules presented in the previous section we define an auxiliary function, $rdfTerm\ (\$\,VarName)$ that, given a variable of a SPARQL result type, returns the correctly formatted RDF value of the variable in Turtle syntax. This is done by matching the type of the variable and adding the necessary syntactic elements for each type, as shown in Rule (25).

$$
\begin{array}{l}
[\![rdfTerm\ (\$\,VarName)]\!]_{Expr} \\
== \\
\texttt{typeswitch}(\$\,VarName) \\
\texttt{case}\ \$e\ \texttt{as literal} \\
\quad \texttt{let}\ \$DT := data(\$e/@datatype) \\
\quad \texttt{let}\ \$L := data(\$e/@xml{:}lang) \\
\quad \texttt{return fn:concat}\,(\texttt{"""},\$e, \\
\quad\quad \texttt{if}(\$L))\ \texttt{then fn:concat}(\text{"@"}, \$L) \\
\quad\quad \texttt{else}\ \text{""} \\
\quad\quad \texttt{if}(\$DT))\ \texttt{then} \\
\quad\quad\quad \texttt{fn:concat}(\texttt{"\^{}\^{}<"}, \$DT, \texttt{">"}) \\
\quad\quad \texttt{else}\ \text{""} \\
\quad ,\texttt{"""}) \\
\texttt{case}\ \$e\ \texttt{as bnode} \\
\quad \texttt{return fn:concat}(\texttt{"\_:"},\ \$e) \\
\texttt{case}\ \$e\ \texttt{as uri} \\
\quad \texttt{return fn:concat}(\texttt{"<"},\$e,\texttt{">"}) \\
\texttt{default} \\
\quad \texttt{return ""}
\end{array}
\tag{25}
$$

The $rdfTerm$ function is then used to serialise triples to text in Turtle syntax, that is, in our implementation rule (19) can be viewed as being replaced by:

$$
\frac{\begin{array}{c}
\text{dynEnv} \vdash \mathit{fs{:}bnode}(\mathit{Subject}) \Rightarrow \mathit{ValueS} \\
\text{statEnv} \vdash \mathit{ValueS}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode}) \\
\text{dynEnv} \vdash \mathit{Predicate} \Rightarrow \mathit{ValueP} \\
\text{statEnv} \vdash \mathit{ValueP}\ \textbf{matches}\ \texttt{uri} \\
\text{dynEnv} \vdash \mathit{fs{:}bnode}(\mathit{Object}) \Rightarrow \mathit{ValueO} \\
\text{dynEnv} \vdash \mathit{ValueO}\ \textbf{matches}\ (\texttt{uri} \mid \texttt{bnode} \mid \texttt{literal})
\end{array}}{\begin{array}{ll}
\text{dynEnv} \vdash & \mathit{fs{:}validTriple}(\mathit{Subject},\ \mathit{Predicate},\ \mathit{Object}) \\
& \Rightarrow\ \texttt{fn:concat}\,(\mathit{rdfTerm}(\mathit{Subject}),\ \text{`\ '},\ \mathit{rdfTerm}(\mathit{Predicate}),\ \text{`\ '},\ \mathit{rdfTerm}(\mathit{Object}), \text{`.'})
\end{array}}
\tag{26}
$$

fs:skolemConstant Our implementation of the $fs{:}skolemConstant$ function simply appends all the position variables from the static context (stored in the *statEnv.posVars* component) to the respective blank node identifier using "_" as a

separator, as shown in the following normalisation rule:[9]

$$\frac{\begin{array}{c}\text{statEnv} \vdash BNodeName : \texttt{bnode} \\ \text{statEnv.posVars} = (PosVar_1, \cdots, PosVar_n)\end{array}}{\begin{array}{c}[\![\texttt{"\_:"}\, BNodeName]\!]_{skolemConstant} \\ = \\ \text{statEnv} \vdash \texttt{fn:concat}\left(\begin{array}{l}\texttt{"\_:"}, BNodeName, \\ \texttt{"\_"}, \$PosVar_1, \cdots, \\ \texttt{"\_"}, \$PosVar_n\end{array}\right)\end{array}} \tag{27}$$

5 Optimisation and Experiments

In this section we present possible rewriting strategies for a specific type of queries, namely queries using nested loops. We are specifically interested in nested loops with an inner loop consisting of a `SparqlForClause`, as the number of interleaved calls to the SPARQL engine can be potentially minimised in these kind of queries.

Definition 8 (Dependent Join)**.** *We call two nested XSPARQL* **`for`** *expressions (`ForClause` or `SparqlForClause`), where the inner loop is a `SparqlForClause`) and at least one variable in the inner query is bound by the outer query, a* dependent join*. The shared variables between the* **`for`** *expressions are called* dependent variables*.*

Note that the strategies presented her are only applicable for dependent joins satisfying the following constraints:

1. The dataset clause of the inner query is statically determined i.e., it cannot be determined based on variables that might be bound from the outer loop;
2. the dependent variable in the inner query's graph pattern must be strictly bounded, where strict boundedness is defined as follows.

Definition 9 (Strict Boundedness)**.** *The set of strictly bound variables in a graph pattern P (as per Definition 2), denoted $bVars(P)$, is defined recursively as follows:*

- *if P is a BGP, then $bVars(P) = Vars(P)$*
- *if $P = (P'\ P'')$, then $bVars(P) = bVars(P') \cup bVars(P'')$*
- *if $P = (P'$ **`optional`** $P'')$, then $bVars(P) = bVars(P')$*
- *if $P = (P'$ **`union`** $P'')$, then $bVars(P) = bVars(P') \cap bVars(P'')$*
- *if $P = ($**`graph`** $i\ P')$, then $bVars(P) = bVars(P') \cup (\{i\} \cap \mathcal{V})$*
- *if $P = (P'$ **`filter`** $R)$, then $bVars(P) = bVars(P')$*

Strict boundedness essentially ensures that the join variable does not occur only in a FILTER expression, which would lead to problems in case you call the inner query unconstrained, see below.

Our introduced rewritings 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 join algorithms from relational databases, namely nested-loop joins or sort-merge joins. For performing the join in SPARQL, if the outer loop is a `SparqlForClause` we can implement the join by rewriting both the inner and the outer loop into a single SPARQL call. In case the outer query consists of an XQuery **`for`** clause, we can still consider this approach, but we need to convert the result of the outer XQuery **`for`** clause to an RDF graph, for instance relying on a SPARQL engine that supports SPARQL Update [29]to add this temporary graph to a triple store.

[9] Since this normalisation rule needs to access the static environment, it is written slightly different to the normalisation rules we have used so far, cf. [14, Section 3.2.2]

5.1 Dependent Join implementation in XQuery

The intuitive idea with these rewritings is, instead of using the naïve rewriting that performs one SPARQL query in the inner loop for each iteration of the outer loop, 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 outer loop results, using one of the following strategies.

5.1.1 Nested-loop join

The straightforward way to implement the join over dependent variables direclty in XQuery is by nesting two XQuery **`for`** loops, much like a regular nested-loop join [30] in standard relational databases. The "join" is done by restricting the values of variables from the inner loop to the values taken from the current iteration of the outer loop. We will describe the implementation of this nested-loop join by means of normalisation rules using two additional auxiliary functions: given the set of variables from the outer loop and the $SparqlWhereClause$ of the inner loop as parameters, the dep and $nondep$ functions return dependent and, respectively, non-dependent variables. In case the outer loop is an XQuery **`for`** clause, we rewrite the query by the following normalisation rule:

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l} \texttt{for } Var^{\text{outer}} \texttt{ in } Expr \texttt{ return} \\ \quad \texttt{for } Vars^{\text{inner}}\ DatasetClause \\ \quad SparqlWhereClause \\ \quad SolutionModifier\ ReturnClause \end{array} \right]\!\!\right]_{Expr} \\
== \\
\texttt{let } \$results := \\
sparqlCall \left(\begin{array}{l} \texttt{select } Var^{\text{outer}} \cup Vars^{\text{inner}} \\ \quad DatasetClause \\ \quad SparqlWhereClause \\ \quad SolutionModifier \end{array} \right) \\
\texttt{for } Var^{\text{outer}} \texttt{ in } [\![Expr]\!]_{Expr} \texttt{ return} \\
\quad \texttt{for } \$result \texttt{ at } \$result\_pos \texttt{ in } \$results//\texttt{sr:result} \\
\quad\quad \texttt{where } \left[\!\!\left[dep \left(\begin{array}{c} Var^{\text{outer}}, \\ SparqlWhereClause \end{array} \right) \right]\!\!\right]_{JoinDep(\$result)} \\
\quad\quad \texttt{return} \\
\quad\quad\quad \left[\!\!\left[nondep \left(\begin{array}{c} Var^{\text{outer}}, \\ SparqlWhereClause \end{array} \right) \right]\!\!\right]_{SparqlResult(\$result)} \\
\quad\quad\quad\quad ReturnClause
\end{array}
\tag{28}
$$

Here, we use '$\cup$' to denote the concatenation of two lists of variables. An additional normalisation rule $[\![\cdot]\!]_{JoinDep(\$result)}$ aggregates the actual "join-comparison" in an XPath expression:

$$
\begin{array}{c}
[\![\$Var_1 \cdots \$Var_n]\!]_{JoinDep(\$res)} \\
== \\
\begin{array}{l}
(fn{:}empty\,(\$res/\texttt{sr:binding}[@name = \$Var_1]/*) \ \texttt{or} \\
\quad (\$Var_1 \ \texttt{eq} \ \$res/\texttt{sr:binding}[@name = \$Var_1]/*)) \\
\qquad \texttt{and ... and} \\
(fn{:}empty\,(\$res/\texttt{sr:binding}[@name = \$Var_n]/*) \ \texttt{or} \\
\quad (\$Var_n \ \texttt{eq} \ \$res/\texttt{sr:binding}[@name = \$Var_n]/*))
\end{array}
\end{array}
\tag{29}
$$

Here, the two variables are considered compatible if their values are equal or the inner value ($\$VarRes_i$) is unbound. The normalisation rule for the rewriting corresponding to an outer SPARQL loop is as follows. The `SparqlForClause`s

use two (not necessarily different) datasets:

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l}
\texttt{for } Vars^{\text{outer}}\ DatasetClause^{\text{outer}} \\
SparqlWhereClause^{\text{outer}} \\
SolutionModifier^{\text{outer}}\ \texttt{return} \\
\quad \texttt{for } Vars^{\text{inner}}\ DatasetClause^{\text{inner}} \\
\quad SparqlWhereClause^{\text{inner}} \\
\quad SolutionModifier^{\text{inner}}\ ReturnClause
\end{array}\right]\!\!\right]_{Expr} \\
== \\
\texttt{let } \$results\_inner\ := \\
\quad sparqlCall\left(\begin{array}{l}
\texttt{select } Vars^{\text{inner}} \cup fs{:}dep\left(\begin{array}{c} Vars^{\text{outer}}, \\ SparqlWhereClause^{\text{inner}} \end{array}\right) \\
\quad DatasetClause^{\text{inner}} \\
\quad SparqlWhereClause^{\text{inner}} \\
\quad SolutionModifier^{\text{inner}}
\end{array}\right) \\
\texttt{let } \$results\_outer\ := \\
\quad sparqlCall\left(\begin{array}{l}
\texttt{select } Vars^{\text{outer}} \\
\quad DatasetClause^{\text{outer}} \\
\quad SparqlWhereClause^{\text{outer}} \\
\quad SolutionModifier^{\text{outer}}
\end{array}\right) \\
\texttt{return} \\
\texttt{for } \$result\_outer\ \texttt{at } \$result\_pos\_outer\ \texttt{in } \$results\_outer\texttt{//sr:result} \\
\quad [\![Vars^{\text{outer}}]\!]_{SparqlResult(\$result\_outer)} \\
\texttt{for } \$result\_inner\ \texttt{at } \$result\_pos\_inner\ \texttt{in } \$results\_inner\texttt{//sr:result} \\
\quad \texttt{where } \left[\!\!\left[fs{:}dep\left(\begin{array}{c} Vars^{\text{outer}}, \\ SparqlWhereClause^{\text{inner}} \end{array}\right)\right]\!\!\right]_{JoinDep(\$result\_inner)} \\
\quad \texttt{return} \\
\quad\quad \left[\!\!\left[fs{:}nondep\left(\begin{array}{c} Vars^{\text{outer}}, \\ SparqlWhereClause^{\text{inner}} \end{array}\right)\right]\!\!\right]_{SparqlResult(\$result\_inner)} \\
\quad\quad ReturnClause
\end{array}
\tag{30}
$$

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

5.1.2 Sort-merge join

Instead of a naive nested loop, we may try to re-implement more "clever" join algorithms known from relational databases within XQuery, such as a sort-merge join. Here, first the two join candidate sequences need to be sorted by the join key, whereafter a join over the two sorted lists can be done in linear time (when performing an inner join) [30]. In order to evaluate whether performance of dependent join queries could benefit from such an alternative join implementation, we have also experimented with an alternative rewriting that re-implements sort-merge joins in XQuery, by first ordering the two SPARQL solution mapping sets[10] and then calling a tail-recursive function to perform the actual merge. This function expects six arguments: the first argument holds the result built up so far, the second holds a group of temporary results needed for the left-outer join, the third and the forth hold the sorted data structures for the two join partners, respectivly, and the last two variables hold the current indices: `merge-join($result, $current, $left, $right, $indexLeft as xs:integer, $indexRight as xs:integer) as item()*` The `merge-join` function will call the two functions `local:inner` and `local:outer` which are responsible for creating the results for the inner **`for`** loop, and the outer **`for`** loop respectively. These two functions need to be defined in the rewritten query file since they contain query-dependent parts. The full optimised rewriting of the query in Figure 16 is shown in Figure 22.

[10] Note that we can only apply sort-merge, in case the outer loop is a `SparqlForClause`, since sorting affects ordering of results which is relevant for XML.

5.2 Dependent Join implementation in SPARQL

This form of rewriting of nested loops aims at improving the runtime of the query by delegating the execution of the join to the SPARQL engine – as opposed to retrieving the results and performing the join within XQuery.

5.2.1 `SparqlForClause` within a `SparqlForClause`

For nested loops where both loops consist of `SparqlForClause`s we can implement the join by rewriting the `SparqlForClause`s into a single SPARQL query. The idea here is that such a join encoded as a nested loop in XSPARQL can just be done within a SPARQL query that simply merges the **`where`** clauses of the outer and inner `SparqlForClause`. However, this rewriting is only applicable if it is possible to rewrite the query without any nesting or aggregators since such SPARQL queries are only possible in the not yet standardised SPARQL 1.1. For example, is it not possible to generate the nested XML structure returned by the query presented in Figure 16 using a single SPARQL query or alternatively further processing in XQuery. In these rewriting rules, in addition to the concatenation of $Vars$, we define the concatenation of $DatasetClause$, $SparqlWhereClause$ and $SolutionModifier$. Concatenating two $DatasetClause$s consists of a new $DatasetClause$ that contains all the **`from`** and **`from named`** clauses from the original $DatasetClause$s. For the concatenation of two $SparqlWhereClause$s, i.e. the concatenation of "`where` $GroupGraphPattern^{\text{inner}}$" and "`where` $GroupGraphPattern^{\text{outer}}$" is "`where` $\{GroupGraphPattern^{\text{inner}}$ $GroupGraphPattern^{\text{outer}}\}$". When concatenating two $SolutionModifier$s, while we can have several **`order by`** clauses, we need to assume some restrictions on the **`limit`** and **`offset`** clauses, namely that these clauses exist in at most one of the $SolutionModifier$s we are considering. If so, the concatenation is the concatenation of the corresponding components (**`order by`**, **`limit`** and **`offset`**). The rewriting rule for this strategy is presented next:

$$
\begin{array}{l}
\left[\!\!\left[\begin{array}{l}
\texttt{for } Vars^{\text{outer}}\ DatasetClause^{\text{outer}} \\
SparqlWhereClause^{\text{outer}} \\
SolutionModifier^{\text{outer}} \\
\texttt{return} \\
\quad \texttt{for } Vars^{\text{inner}}\ DatasetClause^{\text{inner}} \\
\quad SparqlWhereClause^{\text{inner}} \\
\quad SolutionModifier^{\text{inner}}\ ReturnClause
\end{array}\right]\!\!\right]_{Expr} \\
\qquad == \\
\texttt{let } \$queryresult := \\
\quad sparqlCall\left(\begin{array}{l}
\texttt{select } Vars^{\text{inner}} \cup Vars^{\text{outer}} \\
\qquad DatasetClause^{\text{inner}} \cup DatasetClause^{\text{outer}} \\
\qquad SparqlWhereClause^{\text{inner}} \cup SparqlWhereClause^{\text{outer}} \\
\qquad SolutionModifier^{\text{inner}} \cup SolutionModifier^{\text{outer}}
\end{array}\right) \\
\texttt{for } \$result \text{ at } \$result\_pos \texttt{ in } \$queryresult//\texttt{sr:result} \\
\quad \left[\!\!\left[Vars^{\text{inner}} \cup Vars^{\text{outer}} \right]\!\!\right]_{SparqlResult(\$result)} \\
\left[\!\!\left[ReturnClause \right]\!\!\right]_{Expr}
\end{array}
\tag{31}
$$

5.2.2 `SparqlForClause` within an XQuery `for`

In case the outer loop is an XQuery **`for`**, and result ordering is not important,[11] a similar strategy of deferring the join to a single SPARQL query is possible, by first transforming the outer loops' XML results into RDF that can be joined in a single SPARQL query with the inner `SparqlForClause`'s **`where`** pattern. To implement this, we can, for instance, rely on a triple store with support for named graphs in order to temporarily store the RDF data corresponding to the outer XQuery **`for`** loop'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, details again in [31]. The following normalisation rule implementing this rewriting assumes an auxiliary normalisation rule $[\![\cdot]\!]_{genTriplePattern}$ that, given the values of XQuery **`for`** variable, generates RDF triples containing the values. These triples are then collected into the variable $\$ds$ that corresponds to the RDF

[11] Note that the approach proposed here potentially looses ordering of XML results of the outer loop, whereas in general, order is relevant in XML.

graph to be inserted into the triple store. This operation is achieved by the auxiliary function $createNamedGraph$, while finally the function $deleteNamedGraph$ takes care of deleting the temporary graph. The normalisation rule is presented next:

$$\begin{array}{l}
\left[\!\!\left[\begin{array}{l}\texttt{for}\ \$\mathit{VarName}\ \mathit{OptTypeDeclaration}\ \mathit{OptPositionalVar}\ \texttt{in}\ \mathit{Expr}\\ \texttt{return}\\ \quad\texttt{for}\ \mathit{Vars}\ \mathit{DatasetClause}\ \mathit{SparqlWhereClause}\\ \quad\mathit{SolutionModifier}\ \mathit{ReturnClause}\end{array}\right]\!\!\right]_{Expr}\\
== \\
\texttt{let}\ \$ds\ :=\ \texttt{fn:concat}(\\
\quad\texttt{for}\ \$\mathit{VarName}\ \mathit{OptTypeDeclaration}\ \mathit{OptPositionalVar}\ \texttt{in}\ \mathit{Expr}\\
\quad\texttt{return}\ \mathit{fs{:}evalTemplate}\left(\left[\!\!\left[\$\mathit{VarName}\right]\!\!\right]_{genTriplePattern}\right]_{normaliseTemplate}\right)\\
)\ \texttt{return}\ \mathit{createNamedGraph}\ (\$ds)\,,\\
\texttt{let}\ \$queryresult\ :=\\
\quad\mathit{sparqlCall}\left(\begin{array}{l}\texttt{select}\ \ \mathit{Vars}\cup\{\$\mathit{VarName}\}\\ \quad\mathit{DatasetClause}\cup\{\texttt{from named}\ \$ds\}\\ \quad\mathit{SparqlWhereClause}\cup\texttt{where}\left\{\texttt{graph}\ \$ds\ [\![\$\mathit{VarName}]\!]_{genTriplePattern}\right\}\\ \quad\mathit{SolutionModifier}\end{array}\right)\\
\texttt{return}\\
\quad\texttt{for}\ \$result\ \text{at}\ \$result_pos\ \texttt{in}\ \$queryresult//\texttt{sr:result}\\
\quad\quad[\![\mathit{Vars}\cup\{\$\mathit{VarName}\}]\!]_{SparqlResult(\$result)}\\
\quad\texttt{return}\ \mathit{ReturnClause},\\
\mathit{deleteNamedGraph}\ (\$ds)
\end{array}\tag{32}$$

where the auxiliary normalisation rule $[\![\cdot]\!]_{genTriplePattern}$ is as follows:

$$\begin{array}{c}
[\![\$\mathit{VarName}]\!]_{genTriplePattern}\\
==\\
\{[]\ \texttt{:value}\ \$\mathit{VarName}\}
\end{array}\tag{33}$$

5.3 Experimental Evaluation

For the evaluation of our implementation we used the XMark [32] benchmark suite which is, according to [33], the most widely used benchmark suite for XQuery. This benchmark suite provides a data generator that produces XML data simulating an auction website and includes 20 XQuery queries over this generated data. Using the provided data generator, we created datasets with sizes of 1, 2, 5, 10, 20, 50 and 100MB, and – for our comparison with XSPARQL – converted the XML datasets into RDF triples (by converting XML element names straightforwardly to RDF predicates[12]); the provided queries were then converted to equivalent XSPARQL queries. Our evaluation 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.

5.3.1 Methodology

Queries with (dependent join) nested loops, where our different rewritings can be applied, were queries 8, 9, and 10 from the XMark suite; these queries are described informally as follows:

$Q8$**:** "List the names of persons and the number of items they bought"

$Q9$**:** "List the names of persons and the names of the items they bought in Europe"

[12]The converted XMark datasets and queries are available `http://xsparql.deri.org/data/xmark/`

	Number of persons	Number of categories
1MB	255	10
2MB	510	20
5MB	1275	50
10MB	2550	100
20MB	5100	200
50MB	12750	500
100MB	25500	1000

Table 2: Benchmark dataset description

```
1  for $id $name from $graph
2  where { [] foaf:name $name ; :id $id. }
3  return <person name="{$name}">{
4    for * from $graph
5    where { $ca :buyer [:id $id] .
6            optional { $ca :itemRef $itemRef .
7                       $itemRef :locatedIn [:name "europe"].
8                       $itemRef :name $itemname } . }
9    return <item>{$itemname}</item>
10 }</person>
```

Figure 16: Query $Q9$ in XSPARQL

$Q10$**:** "List all persons according to their interest"

In our benchmark tests we aim at comparing, for the three queries where our dependent join rewriting is applicable, the query evaluation times of the different rewritings to determine any performance gains. Table 2 presents an overview of the generated data, in terms of the number of persons and item categories modelled, for each of the dataset sizes considered. The increase in number of persons is relevant for queries 8 and 9 since the queries will loop over all persons, while categories are used as the outer loop in query $Q10$.

Since queries $Q8$ and $Q9$ are quite similar, we explain next the slightly more complex query $Q9$ presented in Figure 16.[13] The inner for loop of the query (lines 4–9) is executed once for each person which means that one SPARQL call will be made for each person separately. The number of iterations and execution time of this loop directly depends on the size of the dataset (cf. Table 2 for details), producing an exponential growth in execution time. In the rewriting strategies presented in this section, only two SPARQL calls are performed: one to get all the people (similar to the direct rewriting version), and one additional SPARQL call is used for retrieving all the information about all the auctions in the dataset. Although the query remains exponential, the practical evaluation shows that reducing the number of calls to the SPARQL endpoint drastically improves query execution times.

5.3.2 Results and Interpretation

For the evaluation of the different types of rewritings as presented in Section 5, we ran each query of each rewriting 10 times in order to achieve an average time. In the presented results, we denote the nested loop join rewriting from Section 5.1.1 with the prefix **NL**. For this strategy we tested two variants: performing the join using an XPath expression (**NL-X**) and using an XQuery **`where`** clause to perform the join (denoted **NL-W**). The rewriting using the sort merge join (Section 5.1.2) is denoted as **SM**, while thestrategies of rewriting to SPARQL and using a Named Graph approach are respectively denoted as **SR** and **NG**.

For the evaluation of our SPARQL based rewritings (as presented in Section 5.2) we included slightly modified versions of the queries. Due to the fact that SPARQL does not assume any default ordering for the query results, we declared the XSPARQL queries as "unordered" which, following the XQuery conventions removes the restriction that the solutions need to follow document order. As mentioned in Section 5.2.1, we also want the query output to be

[13] In this query, the $\$graph$ variable is an external variable to the query that will have assigned to it the URI of the dataset to be queried.

	XSPARQL	NL-X	NL-W	SM	SR	NG
$Q8$	10709.48	12.77	43.82	14.15	–	–
$Q9$	10576.36	13.08	37.39	14.98	–	–
$Q10$	450.62	17.79	28.81	19.31	–	–
$Q8'$	10721.48	12.53	40.47	13.31	8.31	–
$Q9'$	10586.85	12.87	36.32	15.01	10.42	–
$Q10'$	452.04	17.23	28.69	21.50	14.78	–
$Q8''$	10621.25	10.27	37.94	11.51	–	14.95
$Q9''$	10513.25	10.34	34.80	12.31	–	6.28
$Q10''$	446.37	15.23	24.88	19.43	–	20.52

Table 3: Query evaluation times (20MB Dataset)

	XSPARQL	NL-X	NL-W	SM	SR	NG
$Q8$	t	46.10	807.12	47.01	–	–
$Q9$	t	45.90	691.22	49.04	–	–
$Q10$	12960.28	64.25	349.79	67.98	–	–
$Q8'$	t	45.52	767.83	47.08	27.04	–
$Q9'$	t	46.43	694.13	48.62	29.19	–
$Q10'$	12362.61	64.72	348.38	132.41	46.93	–
$Q8''$	t	35.49	741.49	36.87	–	54.53
$Q9''$	t	38.06	662.79	36.00	–	17.14
$Q10''$	16200.13	72.10	317.39	140.34	–	75.65

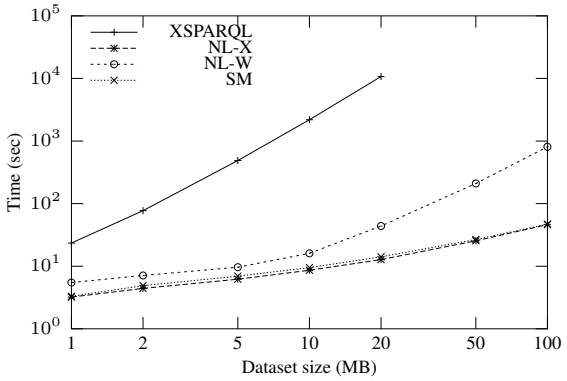
Table 4: Query evaluation times (100MB Dataset)

computable in directly in SPARQL without using aggregators or nesting, 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. Thus, the return format consists of a "flattened" sequence of results which removes the need to do any extra processing of the results returned by SPARQL in XQuery or using SPARQL 1.1 features. The queries that correspond to the strategies presented in Section 5.2.1 are named $Q8'$, $Q9'$ and $Q10'$, while the queries that consist of an outer **`for`** loop, to which the Named Graphs rewriting strategy from Section 5.2.2 can be applied, are called $Q8''$, $Q9''$ and $Q10''$.

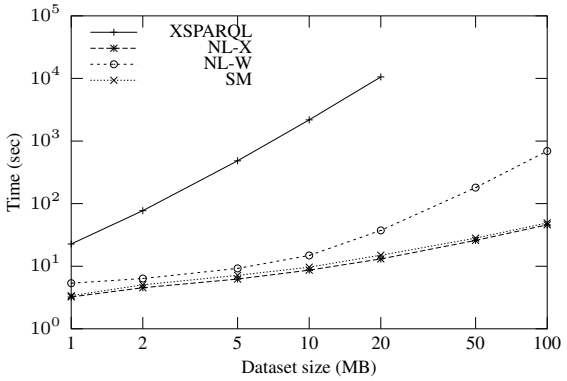
The average evaluation times of these queries for the 20MB and 100MB datasets are presented in Tables 3 and 4, respectively. The 20MB dataset is presented since it is the largest dataset that our direct rewriting approach is able to process. In these tables we can already notice the performance improvement our different rewritings provide. In Table 4, we can find the times for the evaluation of our 100MB dataset for each of our rewritings. Also in this table it is possible to notice a simliar improvement in our different strategies.

A graph depicting the evolution of the evaluation times over the different dataset sizes is presented in Figures 17a, and 17b for queries 8 and 9, respectively. Due to the fact that the original queries are quite similar (cf. Section 5.3.1), it is also expected that the evaluation times for these queries and their respective rewritings be quite similar, as can be seen in Figures 17a and 17b. As for query $Q10$, its behaviour is similar to the one presented for query $Q8$ but the improvement in the execution time is not as significant. This can be explained by the number of times the inner loop is executed, referring to Table 2, we can see that the number of the results of the outer loop (in this case of "categories") increases at a much smaller rate that the number of "persons" (that is used for queries $Q8$ and $Q9$). The comparison of the different versions of this query is presented in Figure 17c. For these queries, the nested loop with the join performed in XPath (**NL-X**) and the sort-merge join rewriting strategies (**SM**) consistently obtained better results when compared to the nested loop with **`where`** join (**NL-W**). As we could already see from Tables 3 and 4 the times for the **NL-X** and **SM** rewritings are quite similar.

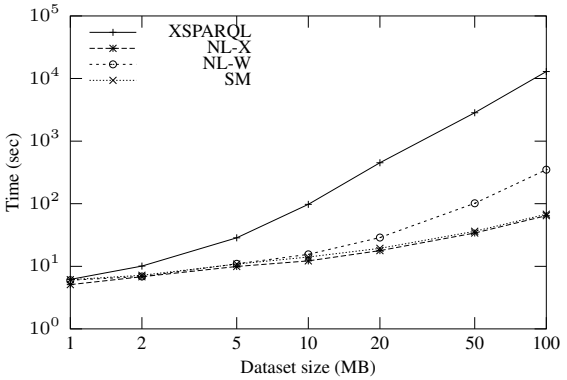
For our variants of queries $Q8$, $Q9$ and $Q10$ we also present graphs with the evolution of the evaluation times. The **SR** strategies graphs are presented in Figures 18a, 18b, and 18c, while the graphs for the **NG** approach are presented



(a) Evaluation times for the different rewriting strategies ($Q8$)



(b) Evaluation times for the different rewriting strategies ($Q9$)

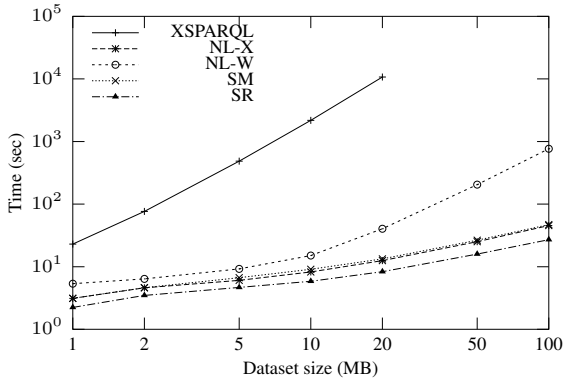


(c) Evaluation times for the different rewriting strategies ($Q10$)

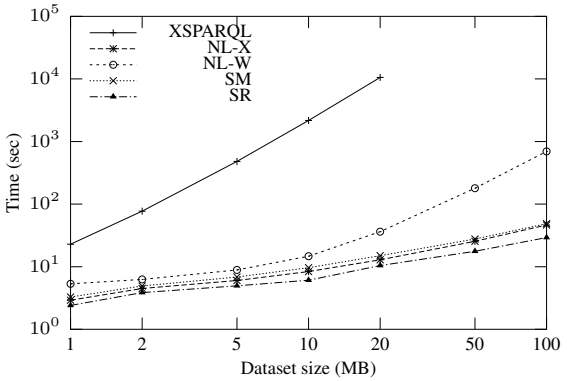
Figure 17: Evaluation times for queries $Q8$, $Q9$ and $Q10$

in Figures 18d, 18e, and 18f. Also here it is noticeable a significant reduction in evaluation times for all the queries when the different rewriting strategies are applied. It is possible to see that the SPARQL based rewritings (presented in Section 5.2) are generally more efficient in terms of evaluation times than the XQuery based. One possible explanation for this fact can be explained by the smaller amount of information that is necessary to pass from SPARQL to the XQuery engine. Only in queries $Q8''$ and $Q10''$ the XQuery-based rewritings outperform the SPARQL-based. This slight slowdown can be caused by the overhead of creating, inserting and deleting the RDF Named Graph.

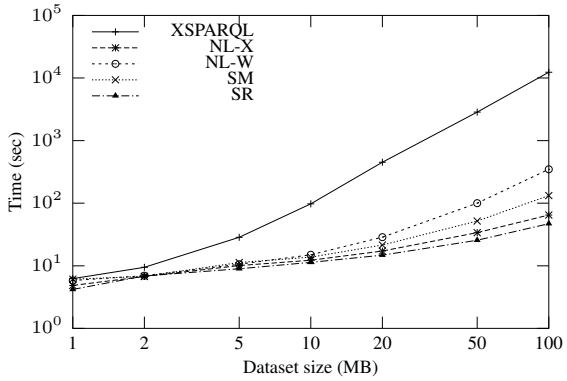
Calculating the *performance gain* allows us to quantitatively determine which rewriting strategies are better. This can be achieved by comparing the times of each rewriting to a baseline evaluation time. As the baseline for this comparison we took the evaluation times of our direct rewriting. The graphs depicting the evolution of the performance gain for the different dataset sizes are presented in Figures 19 and 20 and the average percentage gain for each rewriting strategy is presented in Table 5. In these figures, we can again notice the behaviour described above, where the SPARQL-based rewritings are the ones with higher performance gain. Another observation from this graphs is that the performance gain increases with the size of the dataset. These results also show that the nested loop rewriting with join in XPath (**NL-X**) behaves quite similarly to the sort merge rewriting (**SM**), this can be attributed to the XPath optimisations from the XQuery engine and/or the reduced number of elements the XQuery **`for`** loop iterates over.



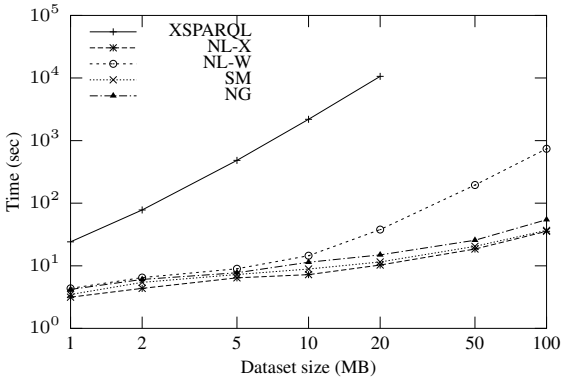
(a) Evaluation times for the different rewriting strategies ($Q8'$)



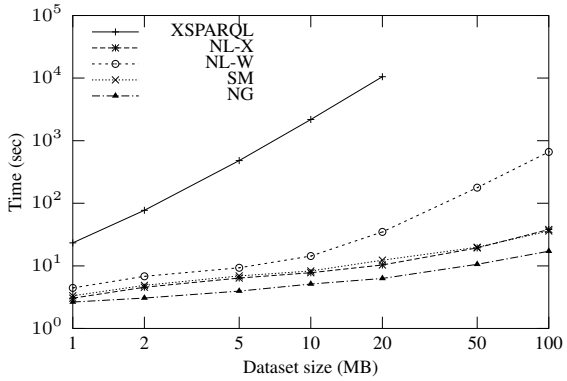
(b) Evaluation times for the different rewriting strategies ($Q9'$)



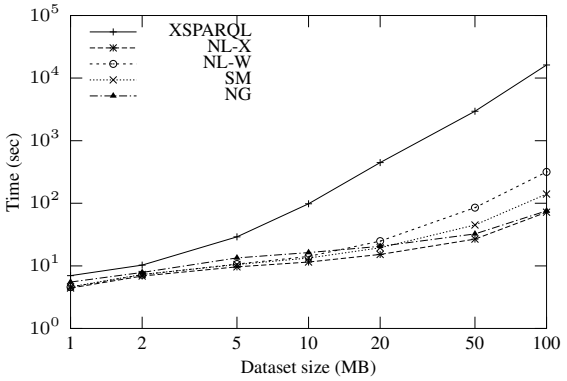
(c) Evaluation times for the different rewriting strategies ($Q10'$)



(d) Evaluation times for the different rewriting strategies ($Q8''$)



(e) Evaluation times for the different rewriting strategies ($Q9''$)



(f) Evaluation times for the different rewriting strategies ($Q10''$)

Figure 18: Evaluation times for queries $Q8'$, $Q9'$, $Q10'$, $Q8''$, $Q9''$ and $Q10''$

	Average Gain
NL-X	169.09
NL-W	63.89
SM	145.88
SR	218.72
NG	210.42

Table 5: Average performance gain for each rewriting

6 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 in 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.

6.1 Data translation

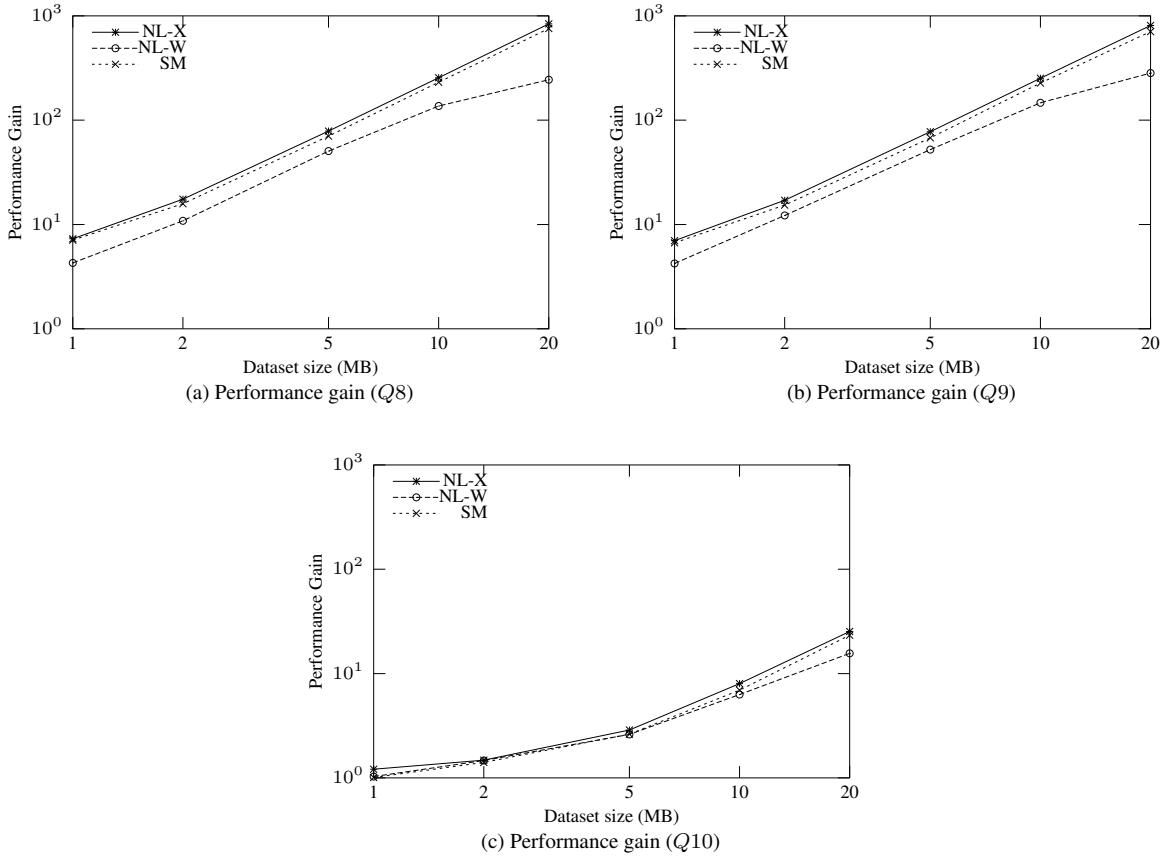
The TriX format [34] consists of an alternative serialisation for RDF in XML, with the aim of being compatible with standard XML tools. It uses XSLT as an extensibility mechanism, allowing to specify syntactic extensions and defining macros. *R3X*[14] uses an RDF processor and XSLT to transform RDF data into a predictable form of RDF/XML also catering for RSS. Similarly, Grit[15] is designed with to be a simplified normalisation for RDF, easier to process with XSLT than RDF/XML. Gloze [35] 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. [36] 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. In [37] they extend this work with the integration of XPath queries into SPARQL Basic Graph Patterns.

Deursen et al. [38] presents an approach for the transformation between XML and RDF in a 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 [39,40] 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 a 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 feel 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 [41] in this report, we should mention the forthcoming SPARQL1.1 [42] specification, that will add many 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 emphasize that XSPARQL – being a Turing-complete scripting language for RDF – will be able to encompass all features within SPARQL1.1 and more.

[14] http://wasab.dk/morten/blog/archives/2004/05/30/transforming-rdfxml-with-xslt
[15] http://code.google.com/p/oort/wiki/Grit



(a) Performance gain ($Q8$)

(b) Performance gain ($Q9$)

(c) Performance gain ($Q10$)

Figure 19: Performance gain for $Q8$, $Q9$ and $Q10$

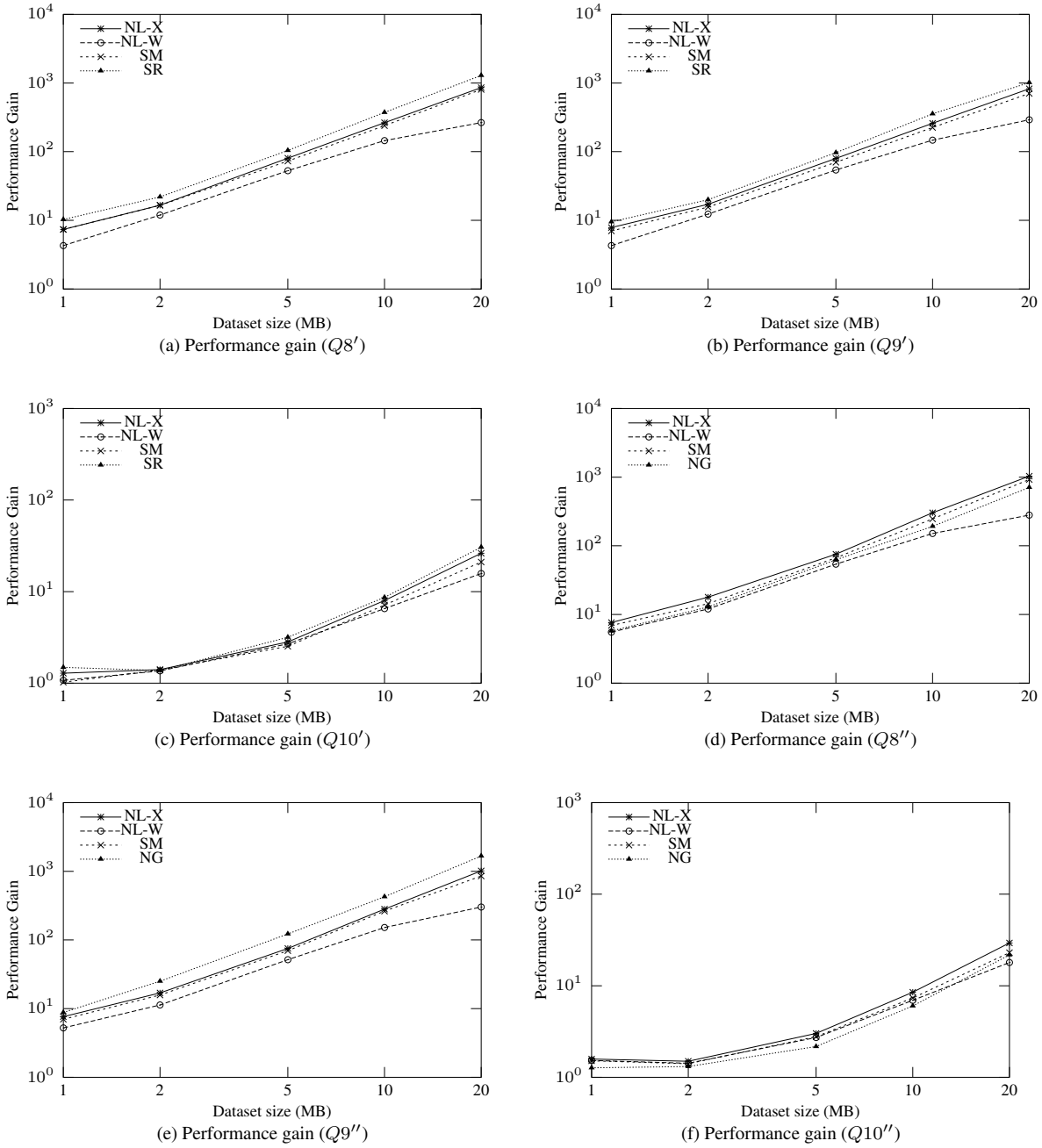
6.2 Language integration

In [43] is presented a framework that allows to perform SPARQL queries from XSLT: XSLT+SPARQL. It relies on adding functions to XSLT that provide the ability to query SPARQL endpoints and uses standard XSLT to process the SPARQL XML results format. Similarly to our current implementation, relies 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, SPARQL2XQuery [44] translates each SPARQL query into an XQuery using a previously defined mapping from OWL to XML Schema and [45] proposes to embed SPARQL into XSLT or XQuery, presenting extensions to these languages to enable SPARQL querying. The translation is done based on query rewriting equivalence for the embedded SPARQL. It does, however, require converting the RDF data to XML according to a predefined schema. On a similar approach, integrating XPath into SPARQL [46], also promises to bridge the gap between XML and RDF. This is approach is very similar to XSPARQL, although the choice where was to extend the SPARQL query language.

RDF Twig [47] 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*[16] 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 many platforms. The nSPARQL query language [48, 49] proposes to extend SPARQL with navigational capabilities using nested regular

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



(a) Performance gain ($Q8'$)

(b) Performance gain ($Q9'$)

(c) Performance gain ($Q10'$)

(d) Performance gain ($Q8''$)

(e) Performance gain ($Q9''$)

(f) Performance gain ($Q10''$)

Figure 20: Performance gain for queries $Q8'$, $Q9'$, $Q10'$, $Q8''$, $Q9''$ and $Q10''$

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 extension is to enable performing XQuery enriched SPARQL queries in XSPARQL.

7 Conclusion and Future Work

In this report we presented a novel query language, called XSPARQL, that combines XQuery and SPARQL in order to provide simplified transformations between the XML and RDF data models. We covered the semantics of XSPARQL, which is defined as an extension of the semantics of XQuery and presented our current implementation that is based on 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 loops involving SPARQL queries and presented a benchmark evaluation of these different rewritings, indicating potential performance gains. For these optimisations we detailed the normalisation rules of how they could be applied for our implementation of the XSPARQL language. The benchmarks that were carried out to determine the impact of our optimisations have shown encouraging results, hinting on a large potential for optimisations in XSPARQL. Among the rewriting strategies presented in this report and on our test data, pushing joins into a SPARQL engine appeared the most promising strategy.

Future Work

In this report we have shown that nested queries can be evaluated much more efficiently when particular rewritings are applied. Nonetheless all the tested rewriting strategies were created basically ad-hoc. To be able to systematically study optimisations, a declarative algebra model for XSPARQL is needed. As starting points, Grust et al. [50] have presented translations of XQuery to SQL, whereas in our own earlier works we have likewise translated SPARQL essentially to Relational Algebra [51]. These works seem to indicate valid starting points for further research on equivalences and optimisations in our language. 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 [29] and XQuery Update [52].

References

[1] T. Bray, J. Paoli, C. M. Sperberg-Mcqueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fifth Edition)," World Wide Web Consortium, W3C Recommendation, Nov. 2008, available at http://www.w3.org/TR/2008/REC-xml-20081126/.

[2] T. Bray, J. Paoli, and C. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0," World Wide Web Consortium, W3C Recommendation, Feb. 1998, available at http://www.w3.org/TR/1998/REC-xml-19980210.

[3] M. K. (ed.), "XSL Transformations (XSLT) Version 2.0 ," W3C, W3C Recommendation, Jan. 2007, available at http://http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[4] D. Chamberlin, J. Robie, S. Boag, M. F. Fernández, J. Siméon, and D. Florescu, "XQuery 1.0: An XML Query Language (Second Edition)," W3C, W3C Recommendation, Dec. 2010, available at http://www.w3.org/TR/2010/REC-xquery-20101214/.

[5] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon, "XML Path Language (XPath) 2.0 (Second Edition)," World Wide Web Consortium, W3C Recommendation, Dec. 2010, available at http://www.w3.org/TR/2010/REC-xpath20-20101214/.

[6] F. Manola and E. Miller, "RDF Primer," W3C, W3C Recommendation, Feb. 2004, available at http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[7] P. Hayes, "RDF Semantics," W3C, W3C Recommendation, Feb. 2004, available at http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

[8] E. Prud'hommeaux and A. S. (eds.), "SPARQL Query Language for RDF," W3C, W3C Recommendation, Jan. 2008, available at http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[9] D. C. (ed.), "Gleaning Resource Descriptions from Dialects of Languages (GRDDL)," W3C, W3C Recommendation, Sep. 2007, available at http://www.w3.org/TR/2007/REC-grddl-20070911/.

[10] J. Farrell and H. Lausen, "Semantic Annotations for WSDL and XML Schema," W3C, W3C Recommendation, Aug. 2007, available at http://www.w3.org/TR/2007/REC-sawsdl-20070828/.

[11] D. Beckett and B. M. (eds.), "RDF/XML Syntax Specification (Revised)," W3C, W3C Recommendation, Feb. 2004, available at http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[12] W. Akhtar, J. Kopecký, T. Krennwallner, and A. Polleres, "XSPARQL: Traveling between the XML and RDF Worlds - and Avoiding the XSLT Pilgrimage," in *ESWC 2008*. Springer, 2008, pp. 432–447.

[13] A. Passant, J. Kopecký, and Stéphane Corlosquet and Diego Berrueta and Davide Palmisano and Axel Polleres, "XSPARQL: Use cases," Jan. 2009, W3C member submission. [Online]. Available: http://www.w3.org/Submission/xsparql-use-cases/

[14] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, "XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)," W3C, W3C Recommendation, Dec. 2010, available at http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/.

[15] B. Adida, M. Birbeck, S. McCarron, and S. Pemberton, "RDFa in XHTML: Syntax and Processing," W3C, W3C Recommendation, Oct. 2008, available at http://www.w3.org/TR/2008/REC-rdfa-syntax-20081014.

[16] D. Beckett and T. Berners-Lee, "Turtle - Terse RDF Triple Language," Jan. 2008, available at http://www.w3.org/TeamSubmission/turtle/. [Online]. Available: http://www.w3.org/TeamSubmission/turtle/

[17] J. Kopecký, T. Vitvar, C. Bournez, and J. Farrell, "SAWSDL: Semantic Annotations for WSDL and XML Schema," *IEEE Internet Computing*, vol. 11, no. 6, pp. 60–67, 2007. [Online]. Available: http://dblp.uni-trier.de/db/journals/internet/internet11.html#KopeckyVBF07

[18] D. Brickley and L. Miller, "FOAF Vocabulary Specification," Nov. 2007, http://xmlns.com/foaf/spec/.

[19] D. Brickley and R. Guha, "RDF vocabulary description language 1.0: RDF Schema," W3C, W3C Recommendation, Feb. 2004, available at http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[20] K. G. Clark, L. Feigenbaum, and E. Torres, "SPARQL Protocol for RDF," W3C, W3C Recommendation, Jan. 2008, available at http://www.w3.org/TR/2008/REC-rdf-sparql-protocol-20080115/.

[21] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Trans. Database Syst.*, vol. 34, no. 3, pp. 1–45, 2009.

[22] A. Malhotra, J. Melton, and N. W. (eds.), "XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition)," W3C, W3C Recommendation, Dec. 2010, available at http://www.w3.org/TR/2010/REC-xpath-functions-20101214/.

[23] H. Katz, D. Chamberlin, M. Kay, P. Wadler, and D. Draper, *XQuery from the Experts: A Guide to the W3C XML Query Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] M. F. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh, "XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)," W3C, W3C Recommendation, Dec. 2010, available at http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/.

[25] R. Iannella, "Representing vCard Objects in RDF," Jan. 2010, W3C member submission. [Online]. Available: http://www.w3.org/Submission/vcard-rdf/

[26] Michael H. Kay. (2011, Feb.) SAXON The XSLT and XQuery Processor. [Online]. Available: http://saxon.sourceforge.net/

[27] P. Bothner, "Compiling XQuery to Java bytecodes," in *XIME-P*, I. Manolescu and Y. Papakonstantinou, Eds., 2004, pp. 31–36.

[28] D. Beckett and J. Broekstra, "SPARQL Query Results XML Format," W3C, W3C Recommendation, Jan. 2008, available at http://www.w3.org/TR/2008/REC-rdf-sparql-XMLres-20080115/.

[29] S. Schenk, P. Gearon, and A. Passant, "SPARQL 1.1 Update," W3C, W3C Working Draft, Oct. 2010, available at http://www.w3.org/TR/2010/WD-sparql11-update-20101014/.

[30] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[31] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres, "Mapping between RDF and XML with XSPARQL," DERI, Tech. Rep., Mar. 2011. [Online]. Available: http://www.deri.ie/fileadmin/documents/DERI-TR-2011-04-04.pdf

[32] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse, "XMark: A Benchmark for XML Data Management," in *VLDB*. Morgan Kaufmann, 2002, pp. 974–985.

[33] L. Afanasiev and M. Marx, "An analysis of XQuery benchmarks," *Inf. Syst.*, vol. 33, no. 2, pp. 155–181, 2008.

[34] J. J. Carroll and P. Stickler, "TriX, RDF triples in XML," HP Labs, Tech. Rep. HPL-2003-268, 2004, available at http://www.hpl.hp.com/techreports/2004/HPL-2004-56.html.

[35] S. Battle, "Gloze: XML to RDF and back again," in *In Proceedings of the First Jena User Conference*, 2006.

[36] M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler, and S. Zugal, "Translating XPath Queries into SPARQL Queries," in *OTM Workshops*, R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 4805. Springer, 2007, pp. 9–10.

[37] M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schopf, H. Staffler, and S. Zugal, "Embedding XPath Queries into SPARQL Queries," in *ICEIS*, J. Cordeiro and J. Filipe, Eds., 2008, pp. 5–14.

[38] D. V. Deursen, C. Poppe, G. Martens, E. Mannens, and R. V. d. Walle, "XML to RDF Conversion: A Generic Approach," in *AXMEDIS '08: Proceedings of the 2008 International Conference on Automated solutions for Cross Media Content and Multi-channel Distribution*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 138–144.

[39] H. Bohring and S. Auer, "Mapping xml to owl ontologies," in *Leipziger Informatik-Tage*, K. P. Jantke, K.-P. Fähnrich, and W. S. Wittig, Eds., vol. 72. GI, 2005, pp. 147–156.

[40] T. Rodrigues, P. Rosa, and J. Cardoso, "Moving from syntactic to semantic organizations using jxml2owl," *Computers in Industry*, vol. 59, no. 8, pp. 808–819, 2008.

[41] A. Polleres, F. Scharffe, and R. Schindlauer, "SPARQL++ for mapping between RDF vocabularies," in *OTM 2007, Part I : Proceedings of the 6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, ser. Lecture Notes in Computer Science, vol. 4803. Vilamoura, Algarve, Portugal: Springer, Nov. 2007, pp. 878–896. [Online]. Available: http://www.polleres.net/publications/poll-etal-2007.pdf

[42] S. Harris and A. Seaborne, "SPARQL 1.1 Query Language," W3C, W3C Working Draft, Oct. 2010, available at http://www.w3.org/TR/2010/WD-sparql11-query-20101014/.

[43] D. Berrueta, J. E. Labra, and I. Herman, "XSLT+SPARQL : Scripting the Semantic Web with SPARQL embedded into XSLT stylesheets," in *4th Workshop on Scripting for the Semantic Web*, C. Bizer, S. Auer, G. A. Grimmes, and T. Heath, Eds., Tenerife, Jun. 2008.

[44] N. Bikakis, N. Gioldasis, C. Tsinaraki, and S. Christodoulakis, "Querying XML Data with SPARQL," in *DEXA*, S. S. Bhowmick, J. Küng, and R. Wagner, Eds., vol. 5690. Springer, 2009, pp. 372–381.

[45] S. Groppe, J. Groppe, V. Linnemann, D. Kukulenz, N. Hoeller, and C. Reinke, "Embedding SPARQL into XQuery/XSLT," in *SAC*, R. L. Wainwright and H. Haddad, Eds. ACM, 2008, pp. 2271–2278.

[46] O. Corby, L. Kefi-Khelif, H. Cherfi, F. Gandon, and K. Khelif, "Querying the semantic web of data using sparql, rdf and xml," Institut National de Recherche en Informatique et en Automatique, Tech. Rep. 6847, February 2009.

[47] N. Walsh, "RDF Twig: accessing RDF graphs in XSLT," in *Extreme Markup Languages*, Montréal, Quebec, Canada, 2003.

[48] M. Arenas, C. Gutierrez, and J. Pérez, "An Extension of SPARQL for RDFS," in *SWDB-ODBIS*, V. Christophides, M. Collard, and C. Gutierrez, Eds., vol. 5005. Springer, 2007, pp. 1–20.

[49] J. Pérez, M. Arenas, and C. Gutierrez, "nSPARQL: A Navigational Language for RDF," in *International Semantic Web Conference*, A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, Eds., vol. 5318. Springer, 2008, pp. 66–81.

[50] T. Grust, M. Mayr, and J. Rittinger, "Let sql drive the xquery workhorse (xquery join graph isolation)," in *EDBT*, I. Manolescu, S. Spaccapietra, J. Teubner, M. Kitsuregawa, A. Léger, F. Naumann, A. Ailamaki, and F. Özcan, Eds., vol. 426. ACM, 2010, pp. 147–158.

[51] A. Polleres, "From SPARQL to Rules (and back)," in *Proceedings of the 16th World Wide Web Conference (WWW2007)*, Banff, Canada, 2007.

[52] J. Robie, D. Chamberlin, M. Dyck, D. Florescu, J. Melton, and J. Siméon, "XQuery Update Facility 1.0," W3C, W3C Recommendation, Mar. 2011, available at http://www.w3.org/TR/xquery-update-10/.

A Rewritten optimisation queries

```
1  import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/xsparql.xquery"
2                   at "http://xsparql.deri.org/demo/xquery/xsparql-types-noval.xquery";
3  declare namespace foaf = "http://xmlns.com/foaf/0.1/";
4  declare namespace _sr =  "http://www.w3.org/2005/sparql-results#";
5
6  declare variable $rdf external;
7
8    let $_aux_results4 := _xsparql:_sparqlQuery( fn:concat("PREFIX foaf: <http://xmlns.com/foaf/0.1/>
9    PREFIX : <http://xsparql.deri.org/data/>
10    SELECT $itemname $id from", _xsparql:_rdf_term( _xsparql:_binding_term( $rdf ) ),
11    " WHERE {  $ca :buyer [ :id $id ].
12               optional { $ca :itemRef $itemRef . $itemRef :locatedIn [ :name ""europe"" ] .
13                          $itemRef :name $itemname  }  }  ") )
14
15 let $_aux_results0 := _xsparql:_sparqlQuery( fn:concat("PREFIX foaf: <http://xmlns.com/foaf/0.1/>
16 PREFIX : <http://xsparql.deri.org/data/>
17  SELECT $id $name from", _xsparql:_rdf_term( _xsparql:_binding_term( $rdf ) ), " WHERE {
   { [] foaf:name $name ; :id $id .  }  } ") )
18 for $_aux_result0 at $_aux_result0_pos in _xsparql:_sparqlResultsFromNode( $_aux_results0 )
19
20 let $id := _xsparql:_resultNode( $_aux_result0, "id" )
21
22 let $name := _xsparql:_resultNode( $_aux_result0, "name" )
23 return
24   <person name="{$name}">
25     {
26   for $_aux_result4 at $_aux_result4_pos in
27     _xsparql:_sparqlResultsFromNode( $_aux_results4 )[./_sr:binding[@name="id"]/* = $id]
28
29   let $itemname := _xsparql:_resultNode( $_aux_result4, "itemname" )
30
31   return
32     <item>{fn:data( $itemname )}</item>}
33     </person>
```

Figure 21: Nested Loop Join of query in Figure 16

```
import module namespace _xsparql = "http://xsparql.deri.org/demo/xquery/xsparql.xquery"
                   at "http://xsparql.deri.org/demo/xquery/xsparql-types-noval.xquery";

declare namespace foaf = "http://xmlns.com/foaf/0.1/";
import module namespace _sort_merge = "http://xsparql.deri.org/demo/xquery/sort_merge.xquery"
                                    at "http://xsparql.deri.org/demo/xquery/sort_merge.xquery";

declare namespace _sr =  "http://www.w3.org/2005/sparql-results#";

declare variable $rdf external;

declare function local:merge($result, $current, $left, $right,
                             $indexLeft as xs:integer, $indexRight as xs:integer) as item()*
{
  let $valueLeft := _xsparql:_resultNode($left[$indexLeft], "id")/text(),
      $valueRight := _xsparql:_resultNode($right[$indexRight], "id")/text(),
      $peekValueRight := $right[$indexRight+1]//_sr:binding[@name="id"]/*/text()
  return
    if ($indexLeft gt count($left) or $indexRight gt count($right)) then
      $result (: base case :)
      (: join :)
    else if ($valueLeft eq $valueRight) then
      (: if the next right hand value is the same as the current, then concatenate only to $current :)
      if($valueRight eq $peekValueRight) then
        local:merge($result,
                    ($current,<item>{_xsparql:_resultNode($right[$indexRight], "itemname")/text()}</item>),
                    $left, $right, $indexLeft, $indexRight+1)
      else (: create the group and put $current in the middle :)
        local:merge(($result,
                    <person name="{_xsparql:_resultNode($left[$indexLeft], "name")/text()}">
                      {$current}<item>{_xsparql:_resultNode($right[$indexRight], "itemname")/text()}</item>
                    </person>),
                   (), $left, $right, $indexLeft+1, $indexRight)
    else if ($valueLeft lt $valueRight) then
      local:merge(($result,<person name="{_xsparql:_resultNode($left[$indexLeft], "name")/text()}"></person>),
                  (), $left, $right, $indexLeft+1, $indexRight)  (: increment left + left join:)
    else if ($valueLeft gt $valueRight) then
      local:merge($result, (), $left, $right, $indexLeft, $indexRight+1) (: increment right :)
    else
      ()(: this is an error :)
};

declare function local:sort-merge($left, $right) as item()*
{
  local:merge((),
              (),
              for $item in $left  order by _xsparql:_resultNode($item, "id") ascending return $item,
              for $item in $right order by _xsparql:_resultNode($item, "id") ascending return $item,
              1,
              1)
};

  let $_inner := _xsparql:_sparqlQuery( fn:concat("PREFIX foaf: <http://xmlns.com/foaf/0.1/>
  PREFIX : <http://xsparql.deri.org/data/>
   SELECT $itemname $id from", _xsparql:_rdf_term( _xsparql:_binding_term( $rdf ) ),
   " WHERE { $ca :buyer [:id $id ] . optional { $ca :itemRef $itemRef .",
                  "$itemRef :locatedIn [ :name ""europe"" ] .",
                  "$itemRef :name $itemname  }  } ") )


(: XSPARQL FOR from 21:4 :)
let $_outer := _xsparql:_sparqlQuery( fn:concat("PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://xsparql.deri.org/data/>
 SELECT $id $name from", _xsparql:_rdf_term( _xsparql:_binding_term( $rdf ) ), " WHERE {
{ [] foaf:name $name ; :id $id .  }  } ") )

let $_outerR := _xsparql:_sparqlResultsFromNode( $_outer )
let $_innerR := _xsparql:_sparqlResultsFromNode( $_inner )

return
  local:sort-merge($_outerR, $_innerR)
```

Figure 22: Sort-Merge Join of query in Figure 16