On the Semantics of Heterogeneous Querying of Relational, XML and RDF Data with XSPARQL

Nuno Lopes[1], Stefan Bischof[1], Stefan Decker[1], and Axel Polleres[1,2]

[1] Digital Enterprise Research Institute,
National University of Ireland Galway, Ireland
Email: `firstname.lastname@deri.org`
[2] Siemens AG Österreich, Siemensstrasse 90, 1210 Vienna, Austria

Abstract XSPARQL is a transformation and query language that caters for heterogenous sources: in its present status it is possible to transform data between XML and RDF formats due to the integration of the XQuery and SPARQL query languages. In this paper we propose an extension of the XSPARQL language to incorporate data contained in relational databases by integrating a subset of SQL in the syntax of XSPARQL. Exposing data contained in relational databases as RDF is a necessary step towards the realisation of the Semantic Web and Web of Data. We present the syntax of an extension of the XSPARQL language catering for the inclusion of the SQL query language along with the semantics based on the XQuery formal semantics and sketch how this extended XSPARQL language can be used to expose RDB2RDF mappings, as currently being discussed in the W3C RDB2RDF Working Group.

1 Introduction

Our aim is to provide a transformation language that is capable of integrating, querying, transforming and exposing data from heterogeneous sources such as relational databases, XML [9], and RDF [23]. This language, called XSPARQL [2], is based on existing standards, currently XQuery and SPARQL, for querying XML and RDF input sources. In this paper we propose to extend XSPARQL with the integration of a subset of the SQL query language, catering for input from relational databases. The final result is an XQuery-flavoured language whose semantics is also defined as an extension of the XQuery semantics [14].

Data integration is a long standing and not easily surpassable problem in information systems [27]. The process commonly known as ETL (Extract, Transform and Load) can be used to provide data integration for sources that share the same underlying format, *e.g.*, relational databases. The data integration problem is further aggravated if the data resides in sources adhering to different representation formats. In this case, an extended ETL is required, for instance where the Extract process "wraps" different formats into a common representation model. Data integration on the Web is inherently an integration task involving sources from different formats and such scenarios require flexible infrastructures to aggregate data from the different information sources, *e.g.*, relational databases, XML documents and databases and recently RDF data exported from Web content. The relational model (and data adhering to this model) is still the predominant paradigm in current enterprise application scenarios, and many works have investigated either distributed querying or integration of relational sources [19,17,27]. On the Web, semi-structured XML is also widely used as an exchange format [20] and the integration of the relational and XML data formats has been published with the SQL language standard since 2003. The RDF data-model has recently come into the picture, supported by efforts like the Linked Open Data initiatives and is gaining importance on the Web and the Semantic Web. RDF has clear advantages over the RDB and XML formats in terms of providing a common integration model, by the fact that RDF is per se schema-unaware.

Transforming data between the relational model and RDF is a necessary step to further move forward the Semantic Web, as acknowledged by the W3C in the ongoing RDB2RDF Working Group. On the other hand, transforming between XML and RDF is already a much required – but not so simple – task in the Semantic Web, *e.g.*, in the realm of Semantic Web Services.

Another issue is that the current version of SPARQL [26,24], the W3C recommended query language for RDF, is still preliminary in terms of expressivity compared with SQL or XQuery. The SPARQL 1.1 query language, currently under development by the W3C SPARQL Working Group, gives a leap forward in terms of expressivity, however, providing mechanisms to convert data back into the native legacy data models of XML or SQL databases is beyond the scope of the Working Group.

The XSPARQL language considering only the integration of XQuery and SPARQL, was presented in [2] and published as a W3C Member Submission [25]. An updated version of the semantics of XSPARQL along with some possible optimisations was presented in [7]. With the addition of SQL and data from relational databases, the language caters for a much broader set of use cases. As a first trivial example we can think of the use case of exposing data in relational databases as RDF, in a similar approach to the RDB2RDF proposals. But furthermore a common language including SQL, XQuery and SPARQL can support more involved transformations, for instance, enabling to integrate enterprise legacy data into Linked Data.

1.1 Related work

In [17] the notion of *dataspaces* is introduced, which can be described as a "topic specific data wrapper" *i.e.*, provides a common view of a certain (general or specific) topic across heterogeneous sources. The paper also describes the functionalities that a dataspace support platform should provide, such as support accessing all the data in the original sources and assuming there are other ways (than those provided by the dataspace) to modify the original sources *i.e.*, do not assume we have full control of the data sources. Although these properties are slightly different and a more involved approach, they share common properties with our language. Furthermore we see our language as the transformation language that is used within a dataspace to access the heterogeneous sources.

Another work that focuses on integration of sources from different data models is Information Manifold [22]. Here each source is declaratively described in terms of the information it contains and its query interface. The declarative model is indeed an interesting part of this work, allowing to define query plans, query equivalence and to ensure query correctness.

There are several projects that enable exposing RDB data as RDF data. For instance, D2R Server [8] and D2R Map or Triplify [5] allow to specify the conversions between RDB data and RDF. Large commercial database companies are also providing solutions for RDF triplestores, such as Oracle [12] and Virtuoso [16]. Most of these projects assume a fixed translation schema where, for instance, database tables are translated into RDFS classes and table attributes are represented as properties.

An analysis of the current RDB2RDF tools is presented in [18], also aiming at studying the expressivity of SPARQL to represent scientific queries, namely in the astronomy domain. Although, as stated by the authors, data and queries were mostly numeric, thus being biased towards relational data and SQL, the comparison gives a good overview of how the tested tools perform in comparison to relational databases. Some of the conclusions indicate that these tools are still not able to compete with relational databases in terms of performance and that SPARQL is also not yet expressive enough to pose the necessary queries.

In comparison to XSPARQL, all of these approaches focus on mapping relational data to RDF but do not consider the integration of XML. Providing an integrated language to query and transform between the three data models is XSPARQL's strong point.

In the next section we present notions of the XSPARQL language, considering the integration of XQuery and SPARQL, and briefly present the syntax and semantics of the subset of SQL considered in this paper. Section 3 then proceeds to presenting the integration of SQL in XSPARQL, the new syntax and the semantics of the new expressions. In Section 4 we introduce the R2RML mapping language and present an algorithm to process it in XSPARQL using the new SQL expressions and Section 5 wraps up the paper while discussing future work.

1.2 Running example: Band Members

As a running example throughout this paper we will use the a database modelling *Bands* and *Persons* that are members of bands. A possible schema for such a database, along with data adhering to the schema, is presented in Figure 1.[3] In this figure the names of relation attributes represented as **bold face** indicate primary keys of a relation, while *italic* corresponds to a foreign key constraint (in

[3] The relation attributes described in figure are of type "character string" for the data which is represented between "quotes" and numeric for the rest. Furthermore, attribute **ssn** is short for Social Security Number.

id	name	origin
1	"U2"	"Dublin"

(a) representation of relational table `band`

ssn	name	*memberOf*
123	"Bono"	1

(b) representation of relational table `person`

Figure 1. Example database schema

this case the *memberOf* attribute of relation `person` references the **id** attribute of relation `band`).

2 Preliminaries

This section presents a short overview of the current status of the XSPARQL language and the subset of SQL considered for the integration in XSPARQL.

2.1 XSPARQL: merging XQuery and SPARQL

In its current iteration, XSPARQL can facilitate the process of transforming data between the XML and RDF formats by merging the XQuery and SPARQL query languages. 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.

Most approaches to transform data from RDF to XML rely on performing a SPARQL `SELECT` query to gather the desired data and then performing a transformation over the XML serialisation of the SPARQL `SELECT` results. XSPARQL allows to perform such transformations within a single language, relying on the full power of XQuery for processing results but also improves transformations from XML to RDF, for example by performing automatic validation of the constructed RDF. Syntactically, the XSPARQL language integrates SPARQL `SELECT` queries as a new expression in XQuery. A schematic overview of the XSPARQL syntax is presented in Figure 5 (already including the SQL integration) and for further details the reader is referred to [7]. As an example of an XSPARQL query, consider a trivial representation of our running example data (Figure 1) in XML: the XSPARQL query presented in Figure 2 converts this XML input into RDF.[4]

Semantics The semantics of XSPARQL is based on the XQuery Formal Semantics and thus we next provide a small overview of the semantics of XQuery (for a more detailed description the reader is referred to [14]). The semantics of XQuery is defined by the following types of rules: *normalisation rules*, *static typing rules* and *dynamic evaluation rules*. *Normalisation rules* reduce XQuery to XQuery Core: a subset of XQuery over which the language is defined. These rules are represented using mapping rules of the following notation:

[4] The complete XML input file for this query is available at `http://xsparql.deri.org/data/bands.xml`.

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix band: <http://example.org/bands#>

for $person in doc("bands.xml")//person
for $band in doc("bands.xml")//band
where data($person/memberOf) eq data($band/@id)
construct { [] foaf:name {$person//name}; band:memberOf {$band//name} }
```

Figure 2. XSPARQL query example

Head:	**S**	`select` *AttrSpecList*
Body:	**D**	`from` *RelationList*
	W	`where` *WhereSpecList*

(a) Simplified schematic view on SQL

SQL datatype	**XML datatype**
character string	xs:string
numeric, decimal	xs:decimal
boolean	xs:boolean

(b) Mapping from SQL datatypes to XML datatypes (partial)

Figure 3. Overview of the SQL language and mapping from SQL to XML datatypes

$$\begin{gathered}[\![\mathit{Object}]\!]_{\mathit{subscript}} \\ == \\ \mathit{Mapped\ Object}\end{gathered}$$

This rule indicates that *Object* will be transformed into *Mapped Object* in XQuery Core. The *subscript* is used to identify sets of rules and allow to selectively apply them. *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. *Static typing* and *dynamic evaluation rules* are represented using logical inference rules containing a set of premises and a conclusion, represented as follows:

$$\frac{\mathit{premise}_1 \ \ldots \ \mathit{premise}_n}{\mathit{conclusion}}$$

In each of these steps two *environments* are available: statEnv and dynEnv. *Environments* consist of a set of key/value pairs that can be used to store extra information regarding the processing of the query. These are used for instance to store the types of variables (statEnv) or their values during evaluation (dynEnv) and can be accessed by the expressions statEnv.varType and dynEnv.varValue, respectively. Other used operators are: (i) $\mathit{env} \vdash \mathit{expr} \Rightarrow \mathit{value}$ indicates that given environment *env*, *expr* evaluates to *value*; (ii) $\mathit{env} \vdash \mathit{expr} : \mathit{type}$ in environment *env*, *expr* has type *type*; and (iii) environments can be changed (extended) using the + operator.

For further information about the XQuery and SPARQL languages the reader is referred, respectively, to [11,14] and [26,24].

2.2 SQL

For this paper we are considering a subset of SQL that consists of the commonly known *select-from-where* queries: a schematic overview of this subset is presented

```
<SQLResult>
  <result>
    <id>1</id>
    <name>U2</name>
    <origin>Dublin</origin>
  </result>
</SQLResult>
```

Figure 4. SQL XML results format

in Figure 3a. As an initial approach we chose to keep the supported set of SQL expressions small and relay any more complex processing to XSPARQL (which inherits all XQuery features). Thus, the main features of SQL that we are not considering in this integration are aggregate functions and nesting, however both of these features can be achieved by using XQuery functions and nesting.

Mapping SQL results to XML The mapping from SQL datatypes into XML Schema datatypes is defined in the SQL specification and presented in [15]. An example of this mapping is presented in Figure 3b. However XML datatypes generically allow a wider range of valid values and thus concrete mappings may impose further restrictions on XML datatypes. For the scope of our paper we rely on the more general mapping (not considering restrictions or inclusion of vendor specific datatypes) and define the result of an SQL query as a sequence of elements of type `SQLResult` which consists of a complex datatype element with the attribute name and the value of the relation as the text node. For instance the result of an SQL query retrieving all tuples from the "band" relation from the database in our running example (Figure 1) is represented in Figure 4.

Semantics The semantics of SQL is defined by a translation into relational algebra as presented in [10]. SQL is first translated into a simplified syntax over which the translation to relational algebra is defined. The result of evaluating an SQL *select-from-where* query consists of a multiset of tuples, *i.e.*, may contain repeated answers in the results. This multiset can be trivially translated into our datatype `SQLResult`.

3 RDB integration in XSPARQL

In this section we specify the extension of XSPARQL towards querying relational data by extending XSPARQL's syntax and semantics (detailed in [7]).

3.1 Syntax

This extension of XSPARQL consists of merging the subset of SQL presented in Figure 3a into the syntax of XSPARQL and is presented schematically in Figure 5. The XSPARQL syntax rules, although restricted to the new rules representing SQL *select-where-clauses*, are presented in Figure 6.[5] A query example, over the

[5] In these syntax rules `VarRef` represents an XQuery variable (`"$"` prefixed).

Prolog:	`declare namespace` *prefix*="*namespace-URI*"	**or**	
	`prefix` *prefix*: <*namespace-URI*>		
Body:	`for` *var* `in` *FLOWR' expression* `let` *var* := *FLWOR' expression* `where` *FLWOR' expression* `order by` *FLWOR' expression*	**or**	XQuery
	`for` *varlist* `from` / `from named` (<*dataset-URI*> or *FLWOR' expr*) `where` { *pattern* } `order by` *expression* `limit` *integer* > 0 `offset` *integer* > 0	**or**	SPARQL
	`for` *SelectSpec* `from` *RelationList* `where` *WhereSpecList*		SQL
Head:	`construct` { *template (with nested FLWOR' expressions)* }	**or**	
	`return` *XML+ nested FLWOR' expressions*		

Figure 5. Schematic view of XSPARQL

```
XSPARQLExpr     ::=  (FLWORExpr | SPARQLForClause | SQLForClause)
                     (ReturnClause | ConstructClause)
SQLForClause    ::=  "for" SelectSpec RelationList SQLWhereClause?
SelectSpec      ::=  AttrSpecList | "*"  | "row" VarRef
AttrSpecList    ::=  AttrSpec AttrNameSpec? ("," AttrSpec AttrNameSpec?)*
AttrSpec        ::=  attrName | VarRef | relationName.attrName |
                     VarRef.attrName | relationName.VarRef | VarRef.VarRef
AttrNameSpec    ::=  "as" VarRef
RelationList    ::=  "from" TableSelector ("," TableSelector)*
TableSelector   ::=  TableName ("as" TableAlias)? | VarRef ("as" TableAlias)?
SQLWhereClause  ::=  "where" WhereSpecList
WhereSpecList   ::=  "(" WhereSpecList BooleanOp WhereSpecList ")" |
                     AttrSpec ComparisonOp AttrSpec | AttrSpec ComparisonOp Constant
BooleanOp       ::=  "and" | "or"
ComparisonOp    ::=  "=" | "!=" | "!=" | "<" | "<=" | ">" | "=>"
```

Figure 6. Extension of the XSPARQL syntax (partial)

relational schema described in Section 1.2, is presented in Figure 7. Intuitively, the newly introduced element `SQLForClause` represents an SQL `select` query that can be evaluated against the underlying database. Similarly to XQuery's `for` clause and XSPARQL's `SparqlForClause`, the `SQLForClause` expression iterates over the results returned by the execution of the SQL query and exposes the result values to other subsequent expressions in the query.

Variable names are assigned to the results of an `SQLForClause` in order for other XSPARQL expressions to reuse the expression results. We provide three ways of specifying variable names for the results of an `SQLForClause`: (i) by explicitly specifying a variable name for each attribute – represented by the syntax rule `AttrNameSpec` (from Figure 6), where `VarRef` is the variable name to which the attribute value is assigned; (ii) implicitly by omitting the variable name or using "`for *`"; and (iii) using the `row` keyword instantiates the specified variable with each *result row* the query produces. For (ii), each attribute in the result set is assigned a variable name automatically (with the same name as the attribute name). For example, in the query from Figure 7, if the `AttrNameSpec` (`as $name`

```
prefix foaf: <http://xmlns.com/foaf/0.1/>
prefix band: <http://example.org/bands#>

for person.name as $name, band.name as $bandName
from person, band
where { person.memberOf = band.id }
construct { [] foaf:name $name; band:memberOf $bandName }
```

Figure 7. XSPARQL DB query example

and `as $bandName`) had been omitted, the generated variable names would be `$person.name` and `$band.name`, respectively.

The third form of specifying variable names is intended to be used when the relation attributes are not known beforehand, *e.g.*, if the relation is specified by a variable. For example, if we use `"row $r"`, `$r` is instantiated with each `result` in the XML format (as represented in Figure 4) and the value of each attribute can be accessed with the XPath expression `$r/attributeName`, *e.g.*, `$r/id`.

So far we are not considering an update language for XSPARQL, thus it is not possible to change a relational database from XML or RDB sources.

3.2 Semantics

We define the semantics of the newly introduced `SQLForClause` by means of the normalisation rules, static type analysis rules, and dynamic evaluation rules presented in this section. The semantics of XSPARQL`ConstructClause`s are specified by rewriting them into `ReturnClause`s according to the rules presented in [7, Section 4.2.3]. This approach is still valid for the newly introduced expressions and thus we do not detail `ConstructClause`s in the rules presented in this paper.

Translating `SQLForClause` into relational algebra For the translation of XSPARQL `SQLForClause`s into relational algebra we rely on the translation from SQL to relational algebra as presented in [10,1]. However, we need to take into account our extended syntax (as presented in Figure 6) that allows for variables in `SQLWhereClauses`. For this translation we introduce a new formal semantics function – *fs:sql* – that extends the translation described in [10] by replacing any variables present in the `SQLWhereClause` with their values (taken from the dynEnv.varValue environment component). If the variable is unbound, *i.e.*, not present in the environment component, it is replaced by null. In our semantics this function also represents the evaluation of the `SQLForClause`, receiving two parameters: *RelationList* and *SQLWhereClause* which represent, respectively, the list of relations that the query involves and the *pattern* to be executed.

Following the XQuery datatypes for SQL results, briefly introduced in Section 2.2, we further introduce the auxiliary function $fs{:}value(SR, var)$ which returns the value of the specified variable *var* in an `SQLResult` *SR*. If *var* is not present in *SR* or its value is null, the empty sequence is returned. The result of the evaluation of the relational algebra expression is a solution sequence that can be translated directly into an XQuery sequence.

Normalisation rules First we specify the normalisation rule for `SQLForClause`s with no attribute selection specified ("`for *`"):

$$\begin{array}{c}
[\![\texttt{for}\ *\ \ \mathit{RelationList}\ \ \mathit{SQLWhereClause}\ \ \mathit{ReturnClause}]\!]_{\mathit{Expr}} \\
== \\
\left[\!\!\left[\begin{array}{l}\texttt{for}\ [\![\mathit{RelationList}\ \ \mathit{SQLWhereClause}]\!]_{\mathit{attrs}}\ \ \mathit{RelationList} \\ \mathit{SQLWhereClause}\ \ \mathit{ReturnClause}\end{array}\right]\!\!\right]_{\mathit{Expr}}
\end{array} \tag{1}$$

The normalisation rule $[\![\cdot]\!]_{\mathit{attrs}}$ returns a comma separated list of variables representing all the attributes from each relation from *RelationList*. These generated variables are of the form: $\$\mathit{relationName}.\mathit{attributeName}$.

Static type analysis The following static type rule defines the type of each variable in an `SQLForClause` as `xs:anySimpleType` and determines the static type of whole expression:

$$\frac{\begin{array}{l}\text{statEnv} + \text{varType}(\mathit{Var}_1 \Rightarrow \texttt{xs:anySimpleType}; \\ \qquad\qquad\qquad \ldots; \\ \qquad\qquad\qquad \mathit{Var}_n \Rightarrow \texttt{xs:anySimpleType} \\ \quad) \vdash \mathit{ReturnExpr} : \mathit{Type}\end{array}}{\text{statEnv} \vdash \begin{array}{l}\texttt{for}\ \mathit{AttrSpec}_1\ \texttt{as}\ \$\mathit{Var}_1 \ldots \mathit{AttrSpec}_n\ \texttt{as}\ \$\mathit{Var}_n \\ \mathit{RelationList}\ \mathit{SQLWhereClause}\ \texttt{return}\ \mathit{ReturnExpr} : \mathit{Type}*\end{array}} \tag{2}$$

This rule, given the static environment statEnv, takes care of creating a new environment with the added information that each of the variables in the `SQLFor-Clause` ($\$\mathit{Var}_1 \ldots \$\mathit{Var}_n$) are of type `xs:anySimpleType`. Given this new extended environment the type of *ReturnExpr* can be inferred to be *Type* (according to *ReturnExpr*), making the type of the overall `SQLForClause` a sequence of elements of type *Type*.

Dynamic Evaluation The dynamic evaluation rules for `SQLForClause`s intuitively define that the return expression *ReturnExpr* will be executed for each `SQLResult` that is returned:

$$\frac{\begin{array}{l}\text{dynEnv} \vdash \mathit{fs{:}sql}(\mathit{RelationList}, \mathit{SQLWhereClause}) \Rightarrow \mathit{SR}_1, \ldots, \mathit{SR}_m \\ \quad \text{dynEnv} + \text{varValue}(\mathit{Var}_1 \Rightarrow \mathit{fs{:}value}(\mathit{SR}_1, \mathit{Var}_1); \\ \qquad\qquad\qquad \ldots; \\ \qquad\qquad\qquad \mathit{Var}_n \Rightarrow \mathit{fs{:}value}(\mathit{SR}_1, \mathit{Var}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1 \\ \qquad\qquad\qquad \vdots \\ \quad \text{dynEnv} + \text{varValue}(\mathit{Var}_1 \Rightarrow \mathit{fs{:}value}(\mathit{SR}_m, \mathit{Var}_1); \\ \qquad\qquad\qquad \ldots; \\ \qquad\qquad\qquad \mathit{Var}_n \Rightarrow \mathit{fs{:}value}(\mathit{SR}_m, \mathit{Var}_n) \\ \qquad) \vdash \mathit{ReturnExpr} \Rightarrow \mathit{Value}_m\end{array}}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for}\ \mathit{AttrSpec}_1\ \texttt{as}\ \$\mathit{Var}_1 \ldots \mathit{AttrSpec}_n\ \texttt{as}\ \$\mathit{Var}_n\ \mathit{RelationList} \\ \mathit{SQLWhereClause}\ \texttt{return}\ \mathit{ReturnExpr} \Rightarrow \mathit{Value}_1, \ldots, \mathit{Value}_m\end{array}} \tag{3}$$

If the evaluation of the SQL expression does not yield any solutions, *i.e.*, evaluates to an empty sequence, the overall result will also be the empty sequence:

$$\frac{\text{dynEnv} \vdash \mathit{fs{:}sql}(\mathit{RelationList}, \mathit{SQLWhereClause}) \Rightarrow ()}{\text{dynEnv} \vdash \begin{array}{l}\texttt{for}\ \$\mathit{Var}_1 \ldots \$\mathit{Var}_n\ \mathit{RelationList}\ \mathit{SQLWhereClause}\\ \texttt{return}\ \mathit{ReturnExpr} \Rightarrow ()\end{array}} \tag{4}$$

4 Processing R2RML mappings in XSPARQL

The W3C RDB2RDF Working Group (WG) is currently in the process of defining a standard language to translate a relational database into RDF. The WG has defined 2 documents: the *Direct Mapping* [4] specifies the process of translating a relational database into RDF in an automated manner, *i.e.*, requiring minimal user input, and the *R2RML* language definition [13] corresponds to a user specified translation (in Turtle syntax) of the input relational database. The direct mapping provides a generic representation of the relational database while the R2RML provides more fine tunned control over the produced RDF.

In this paper we are focusing on the implementation of the R2RML language since the Direct Mapping approach requires access to the schema of the relational database in order to produce the RDF, a feature that is so far not provided within the XSPARQL language. Next we start by giving an overview of the R2RML language and then provide an algorithm for its implementation in XSPARQL.

4.1 The R2RML mapping language

The R2RML mapping is itself an RDF graph consisting of several `TriplesMap`, that specify how to map a *logical table* in the input relational database into RDF. The *logical table* can correspond to a table, a view existing in the database, or the result of an SQL query to be executed over the input relational database.[6]

Each `TriplesMap` consists of one `SubjectMap` and possibly multiple `Predicate-ObjectMaps`. Each row in the logical table produces a single *subject* in the target RDF which is specified by the `SubjectMap`. The multiple `PredicateObjectMaps` each specify how to generate a *predicate* and *objects* (by means of `PredicateMaps` and `ObjectMaps`, respectively) that are related to the generated *subject*.

Furthermore, each `SubjectMap`, `PredicateMap`, and `ObjectMap` may specify how the RDF term is generated by means of different predicates. For instance, using `column` predicate for the mapping rule (*e.g.*, stated as the predicate of the `ObjectMap` as per line 6 of Figure 8) indicates that the RDF object should be generated based on the value of the column in the input database. As another example the `template` predicate allows to specify how terms can be generated by a template based on values from the logical table, *e.g.*, the `subjectMap` from line 3 of Figure 8, states that the generated subject should be of the format

[6] So far we do not allow to execute arbitrary SQL queries in XSPARQL and thus we do not consider the case when a logical table is defined by an SQL query.

```
1 <#TriplesMapBand> a rr:TriplesMap;
2   rr:tableName "band";
3   rr:subjectMap [ rr:template "http://example.com/band/{id}" ];
4
5   rr:predicateObjectMap [
6     rr:predicateMap [ rr:predicate foaf:name ]; rr:objectMap [ rr:column "name" ] ];
7
8   r:predicateObjectMap [
9     rr:predicateMap [ rr:predicate foaf:based_near ]; rr:objectMap [ rr:column "origin" ] ]
```

Figure 8. RDB2RDF mapping for table “band”

Algorithm 1: rdb2rdf($\$m$)

```
  Input: RDB2RDF mapping $\$m$ (represented as RDF)
  Result: RDF Graph
1 let $\$mapSk$ := skolemise($\$m$)
2 for * from $\$mapSk$
3 where
4     $\$map$ rdf:type TriplesMap; rr:subjectMap $\$s$; rr:predicateObjectMap $\$po$
5 return
6     for row $\$tableRow$ in $\$map$ do
7         let $\$subject$ := createTerm($\$mapSk$, $\$tableRow$, $\$s$)
8         createPO($\$mapSk$, $\$tableRow$, $\$subject$, $\$po$)
```

`http://example.com/band/{id}`

where `{id}` is to be replaced by the value of the column `id` in the specific row. The other predicate used in the example from Figure 8 is `rr:predicate` which states that the value used for the predicate of the generated triples should be `foaf:name` for the first `predicateMap` (line 6) and `foaf:based_near` for the `predicateMap` from line 9.

An R2RML mapping produces an RDF dataset with all the generated triples belonging to the default graph unless otherwise stated. Since in XSPARQL we are only able to produce an RDF graph (as opposed to an RDF dataset with possibly several named graphs) we are assuming that all the generated triples belong to the default graph and ignore any rules that state the named graph in which to generate the triple.

For further details on the R2RML mapping language the reader is referred to the W3C specification [13].

4.2 R2RML implementation in XSPARQL

In this section we present an algorithm that has been developed for the implementation of an R2RML mapping in XSPARQL. The algorithm presented in Algorithm 1 assumes that the XSPARQL query will be executed with previously configured access to the underlying relational database. In this algorithm we rely on multiple queries to the R2RML input mapping file and since the R2RML

Algorithm 2: createTerm($\$mapSk, \$row, \$spec$)

```
Input: skolemised RDF2RDF mapping $\$mapSk$, Database data $\$row$, RDF term
       specification $\$spec$
Result: RDF Term
1 for * from $\$mapSk$
2 where
3     $\$spec$ $\$specType$ $\$specValue$
4 return
5     if $\$specType$ == $rr{:}predicate$ then
6         createURI($\$specValue$)
7     else if $\$specType$ == $rr{:}column$ then
8         createLiteral($\$row$/*[name() = $\$specValue$])
9     else ...
```

representation may use blank nodes for describing the mapping, we start by *skolemising* blank nodes in the input RDF graph, *i.e.*, any blank nodes used in the R2RML mapping are substituted with newly generated URIs that are distinct from any other URI in the graph. This transformation allows us to use these newly generated URIs to merge data across different queries and is represented in the algorithm by the *skolemise* function (line 1).

The `SparqlForClause` on lines 2-8 iterates over all the `TriplesMaps` present in the mapping file and, for each of these `TriplesMaps`, retrieves the specified data from the input relational database. This access to the (logical) table of the relational database is represented by the `SQLForClause` on line 6 which, as described in Section 3.1, instantiates $\$row$ with each result row the corresponding SQL query returns. In line 7 we generate the *subject* that is shared by all the triples derived from the same row of the relation and pass it to the auxiliary function (line 8) that takes care of generating the predicate-object pairs.

The auxiliary function `createTerm` is partially presented in Algorithm 2: the function produces an RDF term for a specific database table *$row*, according to the specification given in the RDB2RDF mapping. The `SparqlForClause` from lines 1-3 takes care of querying the RDB2RDF mapping to determine the type of term to be produced. Finally, the **return** clause (lines 4-9) presents the process of creating RDF terms for the `rr:predicate` and `rr:column` types of specifications. The `createURI` and `createLiteral` functions used in this algorithm are built in functions from XSPARQL that behave as constructors for URIs and Literals, respectively. The missing specifications are similar to the presented ones possibly requiring some extra processing, *e.g.*, the `rr:template` specification types needs to be parsed to extract the column names from the template and then access their values of the current row. Foreign key references involve performing an extra `SQLForClause` to access the referenced table from the input database and also the representing `TriplesMap` in the RDB2RDF input mapping.

Algorithm 3 retrieves all the `predicateMap` and `objectMaps` associated with the `TriplesMap` we are processing (lines 1-3), creates the respective *predicate*

Algorithm 3: createPO($\$mapSk, \$row, \$subject, \$po$)

```
Input: skolemised RDF2RDF mapping $\$mapSk$, Database data $\$row$, generated
       RDF term $\$subject$, input RDF term $\$po$
Result: RDF Graph
1 for * from $\$mapSk$
2 where
3     $\$po$ rr:predicateMap $\$p$; rr:objectMap $\$o$
4 return
5     let $\$predicate$ := createTerm($\$mapSk$, $\$row$, $\$p$)
6     let $\$object$ := createTerm($\$mapSk$, $\$row$, $\$o$)
7     construct
8         $\$subject$ $\$predicate$ $\$object$
```

```
<http://example.com/band/1> <http://xmlns.com/foaf/0.1/name> "U2" .
<http://example.com/band/1> <http://xmlns.com/foaf/0.1/based_near> "Dublin" .
<http://example.com/person/123> <http://xmlns.com/foaf/0.1/name> "Bono" .
<http://example.com/person/123> <http://example.org/person#memberOf>
                                                    <http://example.com/band/1> .
```

Figure 9. Output of algorithm rdb2rdf (Algorithm 1)

(line 5) and *object* (line 6) and then generates an RDF triple using the XSPARQL built-in `construct` expression. The `construct` expression automatically takes care of discarding any non-valid RDF triples.

4.3 Mapping result

The RDF graph resulting from the applying Algorithm 1 to the RDF2RDF mapping partially described in Figure 8 is presented in Figure 9. Since the implementation of `SQLForClauses` in XSPARQL is still under development, this output was obtained by using as input an XML representation of the data contained in the relational database.

5 Conclusions

By merging two different query languages, XQuery and SPARQL, the XSPARQL language can be used to easily integrate XML and RDF data, providing a tool for transformation scenarios in several Semantic Web applications.

The extension described in this paper further considers the integration of a subset of the SQL language in XSPARQL thus also providing access to data stored in relational databases. We provided the semantics for this extension of XSPARQL, accounting for the new expression `SQLForClause`, which provides the access to the input relational databases.

A clear use case for the extended XSPARQL language is to expose data from relational databases as RDF. The W3C RDB2RDF Working Group is currently in the process of specifying a standard mapping language for exposing relational data as RDF and, we provided an algorithm describing how the current mapping language (R2RML) can be realised in an XSPARQL query.

The current implementation of the XSPARQL language consists of rewriting the original XSPARQL query into an equivalent XQuery query with interleaved calls to a SPARQL engine to provide facilities for RDF querying. The implementation of the integration of SQL is still ongoing but the rewritten XQuery follows this same approach to retrieve the data contained in relational databases.

Future work Regarding the integration of relational data, the next step for XSPARQL is the implementation of the RDB2RDF direct mapping language with the objective of making XSPARQL a fully compliant RDB2RDF engine and overcoming some of the limitations of the current algorithm such as lacking the possibility of creating named graphs. Another crucial next step is to provide a declarative model sustaining a representative subset of the language with known complexity bounds, while still allowing to perform queries over heterogeneous sources. Some complexity results for a non-recursive core fragment of XQuery have been investigated in [21]. The long standing mapping from relational algebra to Datalog, and the more recently, also the equivalence of SPARQL to relational algebra [3], provides another building block for defining the declarative model of our language. Using the declarative model it is also possible to check the equivalence between any proposed optimisations and also, in a similar approach to [22], allow to assign a cost to each source in order to be able to calculate optimal query plans.

Acknowledgements. The work presented in this report has been funded in part by Science Foundation Ireland under Grant No. SFI/08/CE/I1380 (Líon-2) and by an IRCSET grant co-funded by Alcatel-Lucent.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. W. Akhtar, J. Kopecky, T. Krennwallner, and A. Polleres. XSPARQL: Traveling between the XML and RDF worlds – and avoiding the XSLT pilgrimage. In *ESWC2008*, June 2008.
3. R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In *ISWC*, volume 5318, pages 114–129. Springer, 2008.
4. M. Arenas, E. Prud'hommeaux, and J. Sequeda. A Direct Mapping of Relational Data to RDF. Technical report, W3C Working Draft, Mar. 2011. `http://www.w3.org/TR/2011/WD-rdb-direct-mapping-20110324/`.
5. S. Auer, S. Dietzold, J. Lehmann, S. Hellmann, and D. Aumueller. Triplify: light-weight linked data publication from relational databases. In *WWW*, 2009.
6. P. V. Biron and A. M. (eds.). XML Schema Part 2: Datatypes Second Edition, Oct. 2004. W3C Recommendation, `http://www.w3.org/TR/xmlschema-2/`.

7. S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping between RDF and XML with XSPARQL. Technical report, DERI, Apr. 2011.
8. C. Bizer. D2R MAP - A Database to RDF Mapping Language. In *World Wide Web Conference 2003 (Posters)*, 2003.
9. T. Bray, J. Paoli, C. M. Sperberg-Mcqueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, Nov. 2008. `http://www.w3.org/TR/2008/REC-xml-20081126/`.
10. S. Ceri and G. Gottlob. Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries. *IEEE Trans. Software Eng.*, 11(4):324–345, 1985.
11. D. Chamberlin, J. Robie, S. Boag, M. F. Fernández, J. Siméon, and D. Florescu. XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation, Dec. 2010. `http://www.w3.org/TR/2010/REC-xquery-20101214/`.
12. S. Das and J. Srinivasan. Database technologies for rdf. In *Reasoning Web*, volume 5689, pages 205–221. Springer, 2009.
13. S. Das, S. Sundara, and R. Cyganiak. R2RML: RDB to RDF Mapping Language. W3C Working Draft, Mar. 2011. `http://www.w3.org/TR/2011/WD-r2rml-20110324/`.
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 Recommendation, Dec. 2010. `http://www.w3.org/TR/2010/REC-xquery-semantics-20101214/`.
15. A. Eisenberg and J. Melton. SQL/XML and the SQLX Informal Group of Companies. *SIGMOD Record*, 30(3):105–108, 2001.
16. O. Erling and I. Mikhailov. RDF Support in the Virtuoso DBMS. In *CSSW*, volume 113, pages 59–68. GI, 2007.
17. M. J. Franklin, A. Y. Halevy, and D. Maier. From databases to dataspaces: a new abstraction for information management. *SIGMOD Record*, 34(4):27–33, 2005.
18. A. J. G. Gray, N. Gray, and I. Ounis. Can RDB2RDF Tools Feasibily Expose Large Science Archives for Data Integration? In *ESWC*, vol. 5554. Springer, 2009.
19. A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.
20. I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One, Dec. 2004. W3C Recommendation, `http://www.w3.org/TR/2004/REC-webarch-20041215/`.
21. C. Koch. On the complexity of nonrecursive xquery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, 2006.
22. A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*. Morgan Kaufmann, 1996.
23. F. Manola and E. Miller. RDF Primer. W3C Recommendation, W3C, Feb. 2004. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.
24. J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45, 2009.
25. A. Polleres, T. Krennwallner, N. Lopes, J. Kopecký, and S. Decker. XSPARQL Language Specification, Jan. 2009. W3C member submission, `http://www.w3.org/Submission/xsparql-language-specification/`.
26. E. Prud'hommeaux and A. S. (eds.). SPARQL Query Language for RDF. W3C Recommendation, Jan. 2008. `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`.
27. J. D. Ullman. Information integration using logical views. *Theor. Comput. Sci.*, 239(2):189–210, 2000.