SeRQL: An RDF Query and Transformation Language

DRAFT

Jeen Broekstra
Aduna
jeen.broekstra@aduna.biz

Arjohn Kampman
Aduna
arjohn.kampman@aduna.biz

5th August 2004

Abstract

RDF Query Language proposals are more numerous than fish in the sea it seems. However, the most prominent proposals are query languages that were conceived as first generation tryouts of RDF querying, with little or no RDF-specific implementation and use experience to guide design, and based on an ever changing set of syntactical and semantic specifications.

In this paper, we introduce a set of general requirements for an RDF query language. This set is compiled from discussions between RDF implementors, our own experience and user feedback that we received on our work in Sesame, as well as general principles of query language design. We go on to show how we have compiled these requirements into designing the SeRQL query language, and conclude that SeRQL can be considered a real second generation RDF querying and transformation language.

1 Introduction

RDF Query Language proposals are more numerous than fish in the sea, it seems. However, the most prominent proposals are query languages that were conceived as first generation tryouts of RDF querying, with little or no RDF-specific implementation and use experience to guide design, and based on an ever-changing set of syntactical and semantic specifications.

Now that the RDF specifications have reached the status of Proposed Recommendation and are therefore less likely to change significantly, it is the right time to reevaluate the design of the current set of query languages.

In this paper, we introduce the new RDF query language SeRQL. SeRQL was designed using experiences gained from design and implementation of other query languages and from feedback received from users and developers of these query languages and the systems in which they were implemented, such as Sesame [Broekstra et al., 2002].

SeRQL's aim is to reconcile ideas from existing proposals (most prominently RQL [Karvounarakis et al., 2000], Squish/RDQL [Miller, 2001, Carrol and McBride, 2001, Seaborne, 2004], N-Triples [Grant and Beckett, 2003] and N3 [Berners-Lee, 1998]) into a proposal that satisfies a list of key requirements, and thus offer an RDF query language that is powerful, easy to use and adresses practical problems one encounters when querying RDF.

This paper is organized as follows: in section 2, we present a list of principles and requirements to which an RDF query language should conform. In section 3, we introduce the syntax and design of the SeRQL query language. In section 3.4, we briefly summarize how SeRQL addresses the list of requirements. In section 4, we define a formal interpretation of SeRQL. In section 5, we discuss related work. In section 6, we illustrate by example how SeRQL is used in practice, and finally we present our conclusions in section 8.

2 Query Language requirements

In the previous section, we have introduced the basic syntax of SeRQL. In this section, we will look at some requirements that an RDF query language should fullfill and we will give examples of how SeRQL

supports these requirements.

In [Reggiori and Seaborne, 2002], Alberto Reggiori and Andy Seaborne have collected a number of use cases and examples for RDF queries. From this report, we can distill several general requirements for RDF queries, most notably expressivity requirements. Apart from these requirements, several principles for query languages in general can be taken into account (see [Abiteboul et al., 1999]), such as compositionality, and data model awareness.

From these sources and our experience in implementing and using first generation RDF query languages such as RQL and RDQL, we have composed a list of key requirements for RDF querying. In the next sections, we briefly discuss these requirements and show how SeRQL aims to fullfill them.

In [Abiteboul et al., 1999], a list of requirements for query languages that deal with semistructured data is presented. We highlght a few of these requirements, and we briefly discuss each requirement and how it can be applied to query languages in general and RDF query languages in particular.

2.1 Expressive power

In [Abiteboul et al., 1999] it is noted that the notion of expressive power is ill-defined in the context of semistructured data models. However, we can write down an informal list of the kinds of operations that a query language should express.

Expressiveness requirements that have come up often in dialogue with RDF developers (see also [Reggiori and Seaborne, 2002]) and users of the Sesame system include:

1. A convenient yet powerful path expression syntax for navigating the RDF graph.
2. Functionality for navigating the class/property hierarchy.
3. Functionality for querying reified statements.
4. Value comparison and datatype support.
5. Functionality to deal with *optional* values; properties which may or may not be present in the data for a particular resource.

Of course, this list is far from exhaustive, but these requirements illustrate practical applications of an RDF query language.

2.2 Schema awareness

Query languages should be schema aware. When structure is defined or inferred, a query language should be capable of exploiting the schema for type checking, optimization, and entailment.

This requirement is closely tied with the requirement for formal semantics and for expressive power. In the case of RDF, it means that the query language should be aware of the semantics for RDF and RDF Schema as they are specified by the RDF model theory.

2.3 Program manipulation

It is important that the query language is simple enough to allow program-generated queries. This means that it is often preferable to use a query language syntax that is easy to parse and decompose, rather than try and make it as 'user-friendly' as possible (at the risk of making it ambiguous and thus harder to process). Nevertheless, there is a balance to be obtained here: a query language that is unintelligable to humans will not find acceptance, no matter how well it can be processed automatically.

Considerations to take into account with respect to this requirement include such things as simplicity of structure and avoiding redundancy, while keeping a balance with convenience and readability.

2.4 Compositionality

This requirement states that the output of a query can be used as the input of another query. This is useful in situations where one wants to decompose large queries into smaller ones, or when one wants to execute several queries in series, using the output of the first as the input for the second, etc. A query language with this property will also be able to facilitate view definitions.

In the case of an RDF query language, compositionality obviously means that the result of a query should be representable as an RDF graph. The effect of this is that the query language functions as a *transformation language* on RDF graphs.

2.5 Semantics

Precise formal semantics of a query language are important, because without these query transformations and optimizations are virtually impossible. Moreover, formal descriptions avoid ambiguity and thus help prevent misunderstanding and different implementations of the same language interpreting queries differently.

In the case of an RDF query language, such a formal description can be achieved by providing a mapping to the formal model of RDF itself, the *RDF model theory* as specified in [Hayes, 2003].

3 The Syntax of SeRQL

SeRQL (Sesame RDF Query Language, proncounced ”circle”) is a new RDF/RDFS query language that was developed to address practical requirements from the Sesame user community[1] that were not sufficiently met by other query languages. SeRQL combines the best features of other languages and adds some of its own.

In the rest of this section, we will give an overview of the basic syntax of SeRQL. The overview of the SeRQL language that is given here only covers enough for the purposes of this paper; it is not intended to be complete. A full manual for writing SeRQL queries that covers the complete language is available on the Web [Broekstra and Kampman, 2003].

3.1 URIs, Literals and Variables

URIs and literal values are the basic building blocks of RDF. In SeRQL, URIs are denoted using a syntax derived from N-Triples and N3 notation. Full URIs must be surrounded with `<!` and `>`:

```
<!http://www.openrdf.org/index.jsp>
```

Notice that this is a slight modification of the standard N-Triples notation, where no exclamation mark is used. The exclamation mark is introduced to distinguish full URIs from abbreviated URIs.

As URIs tend to be long strings with the first part being shared by several of them (i.e. the namespace), SeRQL allows one to use abbreviated URIs by defining (short) names for these namespaces which are called ”prefixes”. An abbreviated URI always starts with a `<`, followed by one of the defined prefixes and a colon (`:`). After this colon, the part of the URI that is not part of the namespace follows and the abbreviated URI ends with a `>`. The first part, consisting of the prefix and the colon, is replaced by the full namespace by the query engine. An example abbreviated URI is:

```
<sesame:index.jsp>
```

RDF literals consist of three parts: a label, an optional language tag, and an optional datatype. The notation of literals in SeRQL has been modelled after their notation in N-Triples; literals start with the label, which is surrounded by double quotes, optionally followed by a language tag with a ”@” prefix, or by a datatype URI with a ”b prefix. Example literals are:

- `"foo"`

[1] See `http://www.openrdf.org/`

- `"foo"@en`
- `"foo"^^<!http://some/datatype>`

Note that the SeRQL notation for full URIs is used for the datatype URI (this is slightly different from N-Triples). The SeRQL notation for abbreviated URIs can also be used.

In SeRQL, variables are identified by names. These names must start with a letter or an underscore ('_') and can be followed by zero or more letters, numbers, underscores, dashes ('-') or dots ('.'). Variable names are case-sensitive. SeRQL keywords are not allowed to be used as variable names.

3.2 Path Expressions

One of the most prominent parts of SeRQL are path expressions. Path expressions are expressions that match specific paths through an RDF graph. Most current RDF query languages allow you to define path expressions of length 1, which can be used to find (combinations of) triples in an RDF graph. SeRQL, like RQL, allows to define path expressions of arbitrary length.

SeRQL uses a path expression syntax that is similar to the syntax used in RQL, and is based on the graph nature of RDF: the path is expressed as a collection of nodes and edges, where each node is denoted by surrounding curly brackets:

```
{node} edge {node} edge {node}
```

As an example, suppose we want to query an RDF graph for persons that work for an IT Company. A path expression to express this could look like:

```
{Person} <foo:worksFor> {Company} <rdf:type> {<foo:ITCompany>}
```

Notice that resource URIs and variables are intermixed to provide a template which is matched against the RDF graph.

Multiple path expressions can be comma-seperated. For example, we can split up the above path expression into two simpler ones:

```
{Person} <foo:worksFor> {Company},
{Company} <rdf:type> {<foo:ITCompany>}
```

Notice that SeRQL allows variable repetition for node (or edge) unification.

3.2.1 Extended Path Expressions

As we have just seen, SeRQL has a convenient syntax for basic path expressions, which can be composed into path expressions of arbitrary length.

Every path in an RDF graph can be expressed using these basic path expressions. However, several extended constructions are supported to allow for more convenient expressions of paths.

In situations where one wants to query for two or more triples with identical subject and predicate, the subject and predicate do not have to be repeated. Instead, a *multi-value node* can be used:

```
{subj1} pred1 {obj1, obj2, obj3}
```

This path expression is equivalent to:

```
{subj1} pred1 {obj1},
{subj1} pred1 {obj2},
{subj1} pred1 {obj3}
```

SeRQL also introduces the notion of *branched path expressions*. This is a construction that is useful when multiple properties that emenate from a single node are queried. The semi-column is used to denote a branch:

```
 {subj1} pred1 {obj1};
         pred2 {obj2}
```

which is equivalent to:

```
{subj1} pred1 {obj1},
{subj1} pred2 {obj2}
```

A slightly more complicated example of a branched path expression:

```
{subj1} pred {} pred1 {obj1};
                pred2 {obj2} pred3 {obj3}
```

The empty curly brackets represent a node in the graph that we have no interest in save as a connection point between `pred` and `pred1`. Thus, the node is not assigned a variable.

3.2.2 Reification

RDF allows for a syntactic construction known as *reification*, where the subject or object of a statement is itself a statement. Since it is a syntactic construction it can be expressed using basic path expression syntax, as follows:

```
{statement1} <rdf:type> {<rdf:Statement},
{statement1} <rdf:subject> {subj1},
{statement1} <rdf:predicate> {pred1},
{statement1} <rdf:object> {obj1},
{statement1} pred2 {obj2}
```

However, this is a cumbersome way of dealing with reification. SeRQL introduces a shorthand notation for reified statements that allows one to treat reified statements as actual statements instead of the complex syntactic structure shown above. In this notation, the above reified statement would become:

```
{{subj1} pred1 {obj1}} pred2 {obj2}
```

3.2.3 Class and Property Hierarchies

In the previous section we have shown how the RDF graph can be navigated through path expressions. The same principle can be applied to navigation of class and property hierarchies, since these are, of course, also graphs.

For example, to retrieve the subclasses of a particular class `<my:class1>`:

```
{subclass} <rdfs:subClassOf> <my:class1>
```

Or, to retrieve all instances of class `<my:class1>`:

```
{instance} <rdf:type> <my:class1>
```

However, an RDF class/property hierarchy encapsulates notions such as inheritance, which must be taken into account. Therefore, SeRQL applies the RDF Schema semantics when this is required. In the case of the property `<rdfs:subClassOf>`, for example, SeRQL will return all such relations, including the ones that are entailed according to the model theory.

Additionally, SeRQL support a numer of *built-ins* for expressing queries about the class hierarchy. These built-ins are ”virtual properties”, that is, they are used as normal properties in path expressions, but this property is not expected to actually occur in the RDF graph. Instead, the meaning of the property is pre-defined in terms of other properties.

SeRQL supports four built-ins: `<serql:directSubClassOf>`, `<serql:directSubPropertyOf>` and `<serql:directType>`.

As an example, the built-in `<serql:directSubClassOf>` maps to `<rdfs:subClassOf>` edges in the graph, but only those for which the following conditions hold:

1. the nodes connected by the edge are not the same node.
2. the path between the two nodes formed by this edge is the *only* path between these nodes consisting exclusively of `<rdfs:subClassOf>` edges.

In other words: a class A is a direct subclass of a class B if A and B are not equal and there is no class C that is a subclass of B and a superclass of A.

It is important to note that these built-ins are not merely syntax shortcuts, but actually provide additional expressivity: the notion of direct subclass/property/instance can not be expressed using normal path expressions and boolean constraints only.

3.2.4 Optional Matches

The path expressions and boolean constraints introduced sofar provide the means to specify a template that *must* match the RDF graph in order to return results. However, since the RDF data model is in its very nature weakly structured (or *semi-structured*), it is important that an RDF query language has the means to deal with irregularities.

In contrast to query languages for strongly structured data models, such as SQL [ISO, 1999], RDF query languages must be able to cope with the possibility that a given value may or may not be present. In SeRQL, such values are called *optional matches*. The query language facilitates optional matches by introducing a square-bracket notation that encloses the optional part of a given path expression.

Consider an RDF graph that contains information about people that have names, ages, and optionally e-mail addresses, that is, for some people the e-mail address is known, but for others, it is not. This is a situation that is likely to be very common in RDF data. A logical query on this data is a query that yields all names, ages and, when available, e-mail addresses of people. A path expression to retrieve these values would look like this:

```
{Person} <person:name> {Name};
         <person:age>  {Age};
         <person:email> {EmailAddress}
```

However, using normal path expressions like in the query above, people without e-mail address will not be matched by the template specified by this path expression, and their names and ages will not be returned by the query.

With optional path expressions, one can indicate that a specific (part of a) path expression is optional. This is done using square brackets, i.e.:

```
{Person} <person:name> {Name};
         <person:age>  {Age};
        [<person:email> {EmailAddress}]
```

In contrast to the first path expression, this expression will also match with people without an e-mail address. For these people, the variable `EmailAddress` will not be assigned a value.

Optional path expressions can also be nested. This is useful in situations where the existence of a specific path is dependent on the existence of another path. For example, the following path expression queries for the titles of all known documents and, if the author of the document is known, the name of the author (if it is known) and his e-mail address (if it is known):

```
{Document} <foo:title> {Title};
          [<foo:author> {Author} [<foo:name> {Name}];
                                 [<foo:email> {Email}]]
```

There are a few restrictions on the use of variables in optional path expressions. Most importantly, two optional path expressions that are in parallel to each other (that is, one is not nested within the other) may only have a shared variable if that variable is constrained to a value *outside* either of the optional expressions. For example, the optional path expressions `<foo:name>` {`Name`} and `<foo:email>` {`Email`} share the subject-variable `Author`. This is allowed only because this variable is constrained by the path expression `<foo:author>` {`Author`}, that is, outside the two parallel optional path expressions.

The reason for this restriction becomes apparent when we consider the following example query[2]:

[2]Example by Andy Seaborne and Jeremy Carrol, see `http://lists.w3.org/Archives/Public/www-rdf-interest/2003Nov/0076.html`

```
select *
from [{<x>} <p> {a}], [{<x>} <q> {a}]
```

In this example, the variable `a` is shared between two parallel optional expressions, but it is not otherwise constrained. Now, we further assume that the RDF graph contains the following two RDF statements:

```
<x> <p> <y> .
<x> <q> <z> .
```

In this setting, the variable `a` can be unified with the value `<y>` or with `<z>`, but not both at the same time. The query causes an ambiguity: depending on the order in which the optional expressions are evaluated, the variable gets assigned a different value. Since such order dependency is an undesirable feature in a declarative language, we restrict the language to prevent this.

3.3 Filters and operators

In the preceding sections we have introduced several syntax components of SeRQL. Full queries are built using these components, and using an RQL-style `select-from-where` (or `construct-from-where`) filter. Both filters additionally support a `using namespace` clause.

Queries specified using the `select-from-where` filter return a table of values, or a set of variable-value bindings. Queries using the `construct-from-where` filter return a true RDF graph, which can be a subgraph of the graph being queried, or a graph containing information that is derived from it.

3.3.1 The select and construct clauses

The first clause (i.e. `select` or `construct`) determines what is done with the results that are found. In a `select` clause, one can specify which variable values should be returned and in what order, by means of a comma-seperated list of variables. Optionally, it is possible to use a `*` instead of such a list to indicate that all variables that are used should be returned, in the order in which they appear in the query.

For example, the following query retrieves all classes:

```
select C
from  {C} <rdf:type> {<rdfs:Class>}
```

In a `construct` clause, one can specify which triples should be returned. Construct queries, in their simplest form, simply return the subgraph that is matched by the template specified in the `from` and `where` clauses. The result is returned as the set of triples that make up the subgraph. For example:

```
construct *
from {SUB} <rdfs:subClassOf> {SUPER}
```

This query extracts all triples with a `<rdfs:subClassOf>` predicate from an RDF graph.

However, construct queries can also be used to do `graph transformations` or to specify simple rules. Graph transformation is a powerful tool in application scenarios where mappings between different vocabularies need to be defined.

As an example, consider the following construct query:

```
construct {Parent} <foo:hasChild> {Child}
from {Child} <foo:hasParent> {Parent}
```

This query can be interpreted as a rule that specifies the inverse of the `hasParent` relation. More generally, it specifies a graph transformation: the original graph may not know the `hasChild` relation, but the result of the query is a graph that contains `hasChild` relations between parents and children. The construct clause allows the introduction of new vocabulary, so this query will succeed even if the relation `<foo:hasChild>` is not present in the original RDF graph.

3.3.2 The from clause

The `from` clause always contains path expressions. It defines the paths in an RDF graph that are relevant to the query and binds variables to values.

3.3.3 The where clause

The `where` clause is optional and can contain additional boolean constraints on the values in the path expressions. These are constraints on the nodes and edges of the paths, which cannot always be expressed in the path expressions themselves.

SeRQL contains a set of operators for comparing variables and values that can be used as boolean constraints, including (sub)string comparison, datatyped numerical comparison and a number of boolean functions.

As an example, the following query uses a datatyped comparison to select countries with a population of less than 1 million.

```
SELECT Country
FROM {Country} <foo:population> {Population}
WHERE Population < "1000000"^^<xsd:positiveInteger>
```

For a full overview of the available operators and functions, see the SeRQL user manual [Broekstra and Kampman, 2003].

3.3.4 The using namespace clause

The `using namespace` clause is also optional and it can contain namespace declarations; these are the mappings from prefixes to namespaces for use in combination with abbreviated URIs (see section 3.1).

3.4 Requirements revisited

In section 2, we identified a list of requirements that should hold for RDF query language. In this section, we will briefly revisit this list and show how each requirement is fulfilled by SeRQL.

- **Expressive power**
  In the preceeding sections, we have illustrated SeRQL's expressivity in some detail. Specifically, we have shown how SeRQL handles *optional matches*, *reification* and *schema queries*. SeRQL's path expression syntax has been shown to be very powerful, and we have briefly touched upon datatyping in section 3.3.
- **Schema awareness**
  In section 3.2.3, we have shown how SeRQL handles schema interpretation.
- **Compositionality**
  In section 3.3, we have shown how SeRQL queries can be used for transformation or composition using the `construct` clause.
- **Program manipulation**
  As has been shown in the previous sections, SeRQL has a syntax that is designed to be unambiguous and structured. These properties make it ideally suited to queries being formulated and analyzed through programmatic means.
- **Formal semantics**
  SeRQL is grounded in the RDF Model Theory. In section 4, we present a formal interpretation of SeRQL queries.

4 Formal Interpretation of SeRQL

4.1 Mapping Basic Path Expressions to Sets

The RDF Semantics W3C specification [Hayes, 2003] specifies a model theoretical semantics for RDF and RDF Schema. In this section, we will use this model theory to specify a formal interpretation of SeRQL query constructs.

Without repeating here the entire model theory, we briefly summarize a couple of its notions for reference:

- The sets IR, IP, IC are sets of resources, properties, and classes, respectively. LV is a distinguished subset of IR and is defined as the set of literals.
- $IEXT$ is defined as a mapping from IP to the powerset of $IR \times IR$. Given $p \in IP$, $IEXT(I(p))$ is the set of pairs $\langle x, y \rangle | x, y \in IR$ for which the relation p holds, that is, for which $\langle x, p, y \rangle$ is a statement in the RDF graph.

For an *RDF interpretation*, the following semantic condition holds[3]:

- $x \in IP$ if and only if $\langle x, I(\texttt{rdf:Property}) \rangle \in IEXT(I(\texttt{rdf:type}))$

Additionally, we define υ as a 'null' value, that is $I(x) = \upsilon$ if no value is assigned to x in the current interpretation. We will first characterize SeRQL in terms of RDF only, i.e. give an RDF interpretation. See table 1.

`{x} p {y}`	$\{\langle x, p, y \rangle | \langle x, y \rangle \in IEXT(I(p))\}$
`{x} p {y}; q {z}`	$\{\langle x, p, y \rangle | \langle x, y \rangle \in IEXT(I(p))\} \cup$ $\{\langle x', q, z \rangle | \langle x', z \rangle \in IEXT(I(q))\} \wedge x = x'$
`{x} p {y,z}`	$\{\langle x, p, y \rangle | \langle x, y \rangle \in IEXT(I(p))\} \cup$ $\{\langle x', p', z \rangle | \langle x', z \rangle \in IEXT(I(p'))\} \wedge x = x' \wedge p = p'$
`[{x} p {y}]`	$\{\langle x, p, y \rangle\}$ for which, depending on which variables are undefined, the following conditions hold: case 1: $I(x), I(p), I(y) \neq \upsilon$: $\langle x, y \rangle \in IEXT(I(p))$ case 2: $I(x) = \upsilon, I(p), I(y) \neq \upsilon$: $\nexists x' | \langle x', y \rangle \in IEXT(I(p))$ case 3: $I(x), I(p) = \upsilon, I(y) \neq \upsilon$: $\exists p' | \langle x', y \rangle \in IEXT(I(p'))$ case 4: $I(x), I(y) = \upsilon, I(p) \neq \upsilon$: $IEXT(I(p)) = \emptyset$ case 5: $I(p) = \upsilon, I(x), I(y) \neq \upsilon$: $\nexists p' | \langle x, y \rangle \in IEXT(I(p'))$ case 6: $I(p), I(y) = \upsilon, I(x) \neq \upsilon$: $\nexists p' | \langle x, y' \rangle \in IEXT(I(p'))$ case 7: $I(y) = \upsilon, I(x), I(p) \neq \upsilon$: $\nexists y' | \langle x, y' \rangle \in IEXT(I(p))$

Table 1: RDF interpretation of basic path expressions

An extended interpretation takes into account RDF Schema semantics. For an *RDFS interpretation* the following semantic conditions hold in addition to those specified by an RDF interpretation (cf. [Hayes, 2003]):

- $x \in ICEXT(y)$ if and only if $\langle x, y \rangle \in IEXT(I(\texttt{rdf:type}))$
- $IC = ICEXT(I(\texttt{rdfs:Class}))$
- $IR = ICEXT(I(\texttt{rdfs:Resource}))$
- $LV = ICEXT(I(\texttt{rdfs:Literal}))$
- if $\langle x, y \rangle \in IEXT(I(\texttt{rdfs:domain}))$ and $\langle u, v \rangle \in IEXT(x)$ then $u \in ICEXT(y)$
- if $\langle x, y \rangle \in IEXT(I(\texttt{rdfs:range}))$ and $\langle u, v \rangle \in IEXT(x)$ then $v \in ICEXT(y)$

[3]Other conditions also hold, see [Hayes, 2003], but these are not relevant for this discussion

- $IEXT(I(\texttt{rdfs : subPropertyOf}))$ is transitive and reflexive on IP
- if $\langle x, y\rangle \in IEXT(I(\texttt{rdfs : subPropertyOf}))$ then $x, y \in IP$ and $IEXT(x) \subset IEXT(y)$
- if $x \in IC$ then $\langle x, IR\rangle \in IEXT(I\texttt{rdfs : subClassOf})$
- $IEXT(I(\texttt{rdfs : subClassOf}))$ is transitive and reflexive on IC
- if $\langle x, y\rangle \in IEXT(I(\texttt{rdfs : subClassOf}))$ then $x, y \in IC$ and $IEXT(x) \subset IEXT(y)$
- if $x \in ICEXT(I(\texttt{rdfs : ContainerMembershipProperty}))$
  then $\langle x, I(\texttt{rdfs : member})\rangle \in IEXT(I(\texttt{rdfs : subPropertyOf}))$
- if $x \in ICEXT(I(\texttt{rdfs : Datatype}))$ and $y \in ICEXT(x)$
  then $\langle y, I(\texttt{rdfs : Literal})\rangle \in IEXT(I(\texttt{rdf : type}))$

In table 2, the extensions of the interpretations of SeRQL path expressions and functions that the RDFS semantics add are shown.

{x} <rdf:type> {y}	$\{\langle x, y\rangle \mid x \in ICEXT(y)\}$
{x} <serql:directType> {y}	$\{\langle x, y\rangle \mid x \in ICEXT(y) \wedge$ $(\nexists z \mid z \neq y \wedge x \in ICEXT(z) \wedge$ $\langle z, y\rangle \in IEXT(I(\texttt{rdfs : subClassOf})))\}$
{x} <rdfs:subClassOf> {y}	$\{\langle x, y\rangle \mid \langle x, y\rangle \in IEXT(I(\texttt{rdfs : subClassOf}))\}$
{x} <serql:directSubClassOf> {y}	$\{\langle x, y\rangle \mid x \neq y \wedge \langle x, y\rangle \in IEXT(I(\texttt{rdfs : subClassOf})) \wedge$ $(\nexists z \mid x \neq z \neq y \wedge$ $\langle x, z\rangle, \langle z, y\rangle \in IEXT(I(\texttt{rdfs : subClassOf})))\}$
{p} <rdfs:subPropertyOf> {q}	$\{\langle p, q\rangle \mid \langle p, q\rangle \in IEXT(I(\texttt{rdfs : subPropertyOf}))\}$
{p} <serql:directSubPropertyOf> {q}	$\{\langle p, q\rangle \mid p \neq q \wedge \langle p, q\rangle \in IEXT(I(\texttt{rdfs : subPropertyOf})) \wedge$ $(\nexists r \mid p \neq r \neq q \wedge$ $\langle p, r\rangle, \langle r, q\rangle \in IEXT(I(\texttt{rdfs : subPropertyOf})))\}$

Table 2: RDFS interpretation of basic path expressions

At first glance, the added interpretations for properties such as `rdf:type` may seem redundant, in light of the fact that the case is already covered by the general path expression {x} p {y}. However, these mappings are added to make it explicit that these properties use an *RDFS* interpretation, that is, the semantic conditions regarding a.o. reflexivity and transitivity of these particular properties are observed.

4.2 Functions

Datatypes, operators and functions are strongly interdependent, and to interpret function behaviour in SeRQL formally, we need to summarize how RDF itself handles datatypes. The following is summarized from [Hayes, 2003].

RDF provides for the use of externally defined datatypes identified by a particular URI reference. In the interests of generality, RDF imposes minimal conditions on a datatype.

The semantics for datatypes as specified by the model theory is minimal. It makes no provision for associating a datatype with a property so that it applies to all values of the property, and does not provide any way of explicitly asserting that a blank node denotes a particular datatype value.

Formally, a datatype d is defined by three items:

1. a non-empty set of character strings called the lexical space of d;
2. a non-empty set called the value space of d;
3. a mapping from the lexical space of d to the value space of d, called the lexical-to-value mapping of d.

The lexical-to-value mapping of a datatype d is written as $L2V(d)$.

In stating the semantics we assume that interpretations are relativized to a particular set of datatypes each of which is identified by a URI reference.

Formally, let D be a set of pairs consisting of a URI reference and a datatype such that no URI reference appears twice in the set, so that D can be regarded as a function from a set of URI references to a set of datatypes: call this a datatype map. (The particular URI references must be mentioned explicitly in order to ensure that interpretations conform to any naming conventions imposed by the external authority responsible for defining the datatypes.) Every datatype map is understood to contain $\langle rdf : XMLLiteral, x\rangle$ where x is the built-in XML Literal datatype.

SeRQL supports a set of functions and operators. These functions and operators can be used as part of the boolean constraints in the `where`-clause. Since these functions and operators deal with literal values that can be typed, we use the notion of an *XSD-interpretation* of a vocabulary V as specified in the RDF Semantics. An XSD-interpretation of a vocabulary V is an RDFS-interpretation of V for which the following additional constraints hold (see [Hayes, 2003] for a detailed explanation):

- D contains contains the set of all pairs of the form $\langle http : //www.w3.org/2001/XMLSchema\#sss, sss\rangle$, where `sss` is a built-in datatype named `sss` in XML Schema Part 2: Datatypes [Biron and Malhotra, 2001], and listed in [Hayes, 2003], section 5.1.
- if $\langle a, x\rangle \in D$ then $I(a) = x$.
- if $\langle a, x\rangle \in D$ then $ICEXT(x)$ is the value space of x and is a subset of LV.
- if $\langle a, x\rangle \in D$ then for any typed literal `"sss"^^ddd` in V with $I(ddd) = x$, if sss is in the lexical space of x then $IL("sss"\hat{}\hat{}ddd) = L2V(x)(sss)$, otherwise $IL("sss"\hat{}\hat{}ddd) \notin V$
- if $\langle a, x\rangle \in D$ then $I(a) \in ICEXT(I(rdfs : Datatype))$

We provide a mapping for SeRQL functions in table 3.

`isResource(r)`	`true` if $I(r) \in IR$; `false` otherwise
`isLiteral(l)`	`true` if $I(l) \in LV$; `false` otherwise
`label("sss")`	$\{sss \mid I("\texttt{sss}") \in LV\}$
`label("sss"@lll)`	$\{sss \mid I("\texttt{sss}"@\texttt{lll}) \in LV\}$
`label("sss"^^ddd)`	$\{sss \mid I("\texttt{sss}"\hat{}\hat{}\texttt{ddd}) \in LV\}$
`datatype("sss"^^ddd)`	$\{ddd \mid I("\texttt{sss}"\hat{}\hat{}\texttt{ddd}) \in LV\}$
`language("sss"@lll)`	$\{lll \mid I("\texttt{sss}"@\texttt{lll}) \in LV\}$

Table 3: interpretation of SeRQL functions

4.3 Reducing Composed Expressions

In the previous sections we have seen how basic SeRQL expressions are formally interpreted. In this section, we show how composed path expressions can be reduced to semantically equivalent sets of basic path expressions and boolean constraints by means of a simple substitution.

Definition 1 *A* path expression *is of the form $\langle n_0, e_0, n_1, e_1, n_2, ...e_{i-1}, n_i\rangle$, where i is the length of the path expression, and where $n_0..n_i$ are nodes in the RDF graph and $e_0..e_{i-1}$ are directed edges. Each directed edge e_k has as source node n_k and as target node n_{k+1}.*

Definition 2 *A* basic path expression *is a path expression of length 1.*

composed expression	substituted expressions	constraints
$\langle n_0, e_0, n_1, ..., n_{i-1}, e_{i-1}, n_i \rangle$	$\langle n_0, e_0, n_1, ..., n_{i-2}, e-i-2, n_{i-1} \rangle$, $\langle n'_{i-1}, e_{i-1}, n_i \rangle$	$n_{i-1} = n'_{i-1}$
$\langle n_0, e_0, n_1, e_1, n_2 \rangle$	$\langle n_0, e_0, n_1 \rangle$, $\langle n'_1, e_1, n_2 \rangle$	$n_1 = n'_1$

Table 4: Breaking up composed path expressions

As an example, the SeRQL construction `{x} p {y}` corresponds to the general form $\langle n_0, e_0, n_1 \rangle$, and is a basic path expression.

A path expression of length $i > 1$ can be reduced to two path expression of, one of length $i - 1$ and one of length 1, as shown in table 4.

By recursively applying these substitutions to any path expression of length > 1 it is possible to reduce an arbitrary length composed path expression to a set of basic path expressions and boolean constraints. Thus, any complex SeRQL query can be normalized to a form consisting only of a set of basic path expressions and boolean constraints.

Branching path expressions, mult-valued node expressions and path expressions involving reification can also always be reduced to a set of basic expressions. We will prove this for branching path expressions, the proofs for the other two forms is analogous.

Theorem 1 *Any branching path expression p of the form* `{x} p {y}; q {z}` *can be reduced to a semantically equivalent set of basic path expressions.*

> **Proof:** By definition, the branching expression is syntactically equivalent to the two basic expressions `{x} p {y}, {x} q {z}` (see section 3.2). The first of these is defined as $\{\langle x, p, y \rangle | \langle x, y \rangle \in IEXT(I(p))\}$ (table 1). The second is defined as $\{\langle x, q, z \rangle | \langle x, z \rangle \in IEXT(I(q))\}$. The union of these two sets can be expressed as $\{\langle x, p, y \rangle | \langle x, y \rangle \in IEXT(I(p))\} \cup \{\langle x', q, z \rangle | \langle x', z \rangle \in IEXT(I(q))\} \wedge x = x'$, which is by definition (see table 1) equivalent to the definition of the branching path expression.

5 Related work

RDQL [Seaborne, 2004] is an RDF query language that has been implemented in several system, including the Jena Toolkit [Carrol and McBride, 2001] and Sesame [Broekstra et al., 2002]. It offers many attractive qualities such as an easy to use syntax format. The main advantage of RDQL is that it is simple, allowing easy implementation across platforms and still offer a measure of expressivity that is enough in many practical applications that deal with RDF.

However, RDQL has been deliberately designed to be a minimal language. It lacks expressive power that is, in many more complex application scenarios, required. For example, it is not possible to express disjunctive patterns in RDQL, nor can it express optional matches. It also lacks a formal semantics, which may result in different implementations of the same language giving different results.

RQL [Karvounarakis et al., 2000] is an RDF Schema query language designed by ICS-FORTH. It has been implemented in ICS-FORTH's RDFSuite [Alexaki et al., 2000] tool and Sesame supports a partial implementation.

RQL is a very powerful and expressive language with a thorough formal grounding. It uses a path-expression syntax that closely resembles that of SeRQL (in fact, SeRQL's syntax was in a large part based on RQL) and has a functional approach to query language design.

However, partly because of its expressivity, RQL's syntax has proven to be difficult to use and understand for human users. Also, while it has a formal model, this formal model is not based directly on the RDF Semantics, but on the authors' own formal interpretation of RDF and RDF Schema, and this formal interpretation is incompatible with RDF and RDF Schema on some key points, placing additional restrictions on the structure of the RDF graph. While this is still suitable for a large range of applications, it is prohibitive in cases where interoperability is a key issue.

6 SeRQL in Practice

SeRQL has recently become the default query language of the Sesame system. As such, it is being used by numerous developers and researchers in a wide variety of settings and domains. In this section, we will give a few brief examples of such use cases.

6.1 SWAP

The Semantic Web and Peer to Peer (SWAP) project [Broekstra et al., 2003] is a European IST project that aims to combine technologies from the areas of Ontology and P2P. The SWAP system is a decentralized environment in which peer nodes communicate and share knowledge, using RDF as the basic language.

Each peer node in the SWAP system has a local repository, in which both local knowledge and knowledge obtained from other peers is stored, in RDF. A user interface allows users to edit, browse and query this knowledge. The de-facto standard query language in the SWAP system is SeRQL.

A problem in the SWAP system is that management metadata (sources of information, confidence ratings, etc.) are present in the same repository as the actual data. To allow convenient access to the knowledge to the user SWAP employs definable views on top of the repository. These views are defined using the management metadata, but they only contain the domain knowledge. SWAP defines views by using SeRQL construct-queries that retrieve and transform relevant subgraphs from the repository.

For example, the following SeRQL query is used to construct the view that describes the expertise of known peers. This knowledge is not directly present in the repository, but for every piece of knowledge an associated peer is known:

```
construct
     {P} <view:knowsAbout> {C}
from
     {{C} <rdf:type> {<rdfs:Class>}} <swap:hasSwabbi> {} <swap:hasPeer> {} <swap:hasLabel> {P}
```

(The 'hasSwabbi' property associates a particular domain knowledge statement with an object known as a 'Swabbi'. This Swabbi object then is a placeholder for all the relevant management metadata for this particular statement. See [Broekstra et al., 2003] for details.)

In general, the use of transformation queries in the SWAP context is invaluable, as the information shared between peers consists mainly of RDF models. Since query answers are RDF models themselves, this allows easy integration of knowledge from other peers in a particular peer's own knowledge repository.

6.2 DOPE

The aim of the DOPE project (Drug Ontology Project for Elsevier) [Stuckenschmidt et al., 2004] is to investigate the possibility of providing access to multiple information sources in the area of life sciences, through a single interface. The prototype system that was developed allows thesaurus-driven access to heterogeneous and distributed data, based on the RDF model.

In figure 1, we see the architecture of the DOPE system. Central to the architecture is a mediator, that functions as a central access point for queries posed by the user interface and distributes the query over the distributed data sources. In the prototype, this mediator has been realized using a Sesame's SAIL (Storage And Inference Layer) API, on top of which a SeRQL query engine functions as the entry point for the user interface.

The *Metadata Server* is a repository of information that is not equipped with RDF i/o. However, it does have a SOAP interface. Therefore, an extractor component is deployed which, through use of the SOAP interface, converts the available information in an RDF format that is a 1:1 mapping to the model as it is represented internally in the Metadata server. This model is referred to as the *source model*.

The data from this source is now in RDF, but not in the terminology that user queries are formulated in. Therefore, a transformation takes place from the physical model to a *document model*, using SeRQL construct-queries to define and perform the transformation.

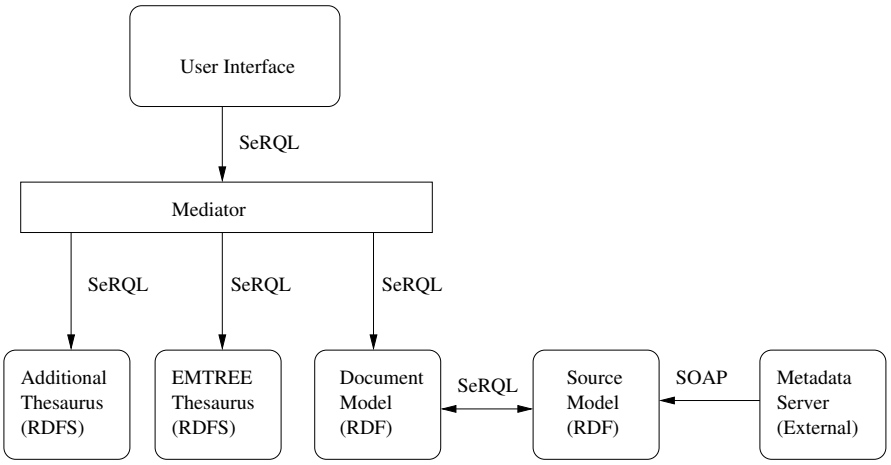

Figure 1: DOPE architecture

7 Future Work

SeRQL is an attempt to come to a basic query language that offers necessary expressivity on RDF(S) models. It is the intention to further develop this language, by adding new functions and operators. We list a few of the operations that we intend to add to the language definition in the near future:

- aggregation functions: `max`, `min`, `count`, `avg`.
- existential/universal quantifiers and set operations. Adding such features to the language will allow us to use the compositional nature of the language fully by having the option of specifying *nested queries*.
- advanced regular string pattern matching.

8 Conclusions

In the previous sections, we have given an overview of the SeRQL query language, and we have demonstrated how SeRQL fulfills a set of key requirements for RDF query languages. We have provided the basic syntax and a formal model, and have illustrated use cases in which the different features of SeRQL are demonstrated.

SeRQL is an attempt to come to an RDF query language that satisfies necessary general requirements on such a language without adding unnecessary bloat. Specifically, SeRQL has been designed to be fully compatible with the RDF specifications, to be easy to read and write by humans while at the same being easy to process and produce in an automated fashion. Most of the features of SeRQL are not new, but we believe that SeRQL is the first proposal that combines all these requirements in a single language and the only such proposal that has been implemented succesfully and is being used succesfully.

Future work on the development of SeRQL as a language will focus on adding useful and necessary functions and operators demanded by the user community, as well as encouraging other developers to implement engines that support this language.

References

[Abiteboul et al., 1999] Abiteboul, S., Buneman, P., and Suciu, D. (1999). *Data on the Web: From Relations to Semistructural Data and XML*. Morgan Kaufman.

[Alexaki et al., 2000] Alexaki, S., Christophides, V., Karvounarakis, G., Plexousakis, D., and Tolle, K. (2000). The RDFSuite: Managing Voluminous RDF Description Bases. Technical report, Institute of Computer Science, FORTH, Heraklion, Greece. See http://www.ics.forth.gr/proj/isst/RDF/RSSDB/rdfsuite.pdf.

[Berners-Lee, 1998] Berners-Lee, T. (1998). Notation 3. See http://www.w3.org/DesignIssues/Notation3.html.

[Biron and Malhotra, 2001] Biron, P. V. and Malhotra, A. (2001). XML Schema Part 2: Datatypes. Recommendation, World Wide Web Consortium. See http://www.w3.org/TR/xmlschema-2/.

[Broekstra et al., 2003] Broekstra, J., Ehrig, M., Haase, P., v. Harmelen, F., Kampman, A., Sabou, M., Siebes, R., Staab, S., Stuckenschmidt, H., and Tempich, C. (2003). A Metadata Model for Semantics-Based Peer-to-Peer Systems. In *1st Workshop on Semantics in Peer-to-Peer and Grid Computing at the Twelfth International World Wide Web Conference*, Budapest, Hungary. See http://swap.semanticweb.org/.

[Broekstra and Kampman, 2003] Broekstra, J. and Kampman, A. (2003). The SeRQL Query Language. User manual, Aduna. See http://sesame.aduna.biz/publications/SeRQLmanual.html.

[Broekstra et al., 2002] Broekstra, J., Kampman, A., and van Harmelen, F. (2002). Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In Horrocks, I. and Hendler, J., editors, *Proceedings of the first International Semantic Web Conference (ISWC 2002)*, number 2342 in Lecture Notes in Computer Science, pages 54–68, Sardinia, Italy. Springer Verlag, Heidelberg Germany. See also http://sesame.aidministrator.nl/.

[Carrol and McBride, 2001] Carrol, J. and McBride, B. (2001). The Jena Semantic Web Toolkit. Public api, HP-Labs, Bristol. See http://www.hpl.hp.com/semweb/jena-top.html.

[Grant and Beckett, 2003] Grant, J. and Beckett, D. (2003). RDF Test Cases. Proposed recommendation, World Wide Web Consortium. See http://www.w3.org/TR/rdf-testcases/.

[Hayes, 2003] Hayes, P. (2003). RDF Semantics. Proposed Recommendation, World Wide Web Consortium. See http://www.w3.org/TR/2003/PR-rdf-mt-20031215/.

[ISO, 1999] ISO (1999). Information Technology-Database Language SQL. Standard No. ISO/IEC 9075:1999, International Organization for Standardization (ISO). (Available from American National Standards Institute, New York, NY 10036, (212) 642-4900.).

[Karvounarakis et al., 2000] Karvounarakis, G., Christophides, V., Plexousakis, D., and Alexaki, S. (2000). Querying community web portals. Technical report, Institute of Computer Science, FORTH, Heraklion, Greece. See http://www.ics.forth.gr/proj/isst/RDF/RQL/rql.pdf.

[Miller, 2001] Miller, L. (2001). RDF Squish query language and Java implementation. Public draft, Institute for Learning and Research Technology. See http://ilrt.org/discovery/2001/02/squish/.

[Reggiori and Seaborne, 2002] Reggiori, A. and Seaborne, A. (2002). See http://rdfstore.sourceforge.net/2002/06/24/rdf-query/query-use-cases.html.

[Seaborne, 2004] Seaborne, A. (2004). RDQL - A Query Language for RDF. W3c member submission, Hewlett Packard. See http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[Stuckenschmidt et al., 2004] Stuckenschmidt, H., de Waard, A., Bhogal, R., Fluit, C., Kampman, A., van Buel, J., van Mulligen, E., Broekstra, J., Crowlesmith, I., van Harmelen, F., and Scerri, T. (2004). Exploring Large Document Repositories with RDF Technology - The DOPE Project. *IEEE Intelligent Systems - Special Issue on the Semantic Web Challenge*. to appear.