The Syntactic Web

Syntax and Semantics on the Web

Jonathan **Robie**

Abstract

XQuery is a query language designed to allow queries across the many kinds of information that are represented in XML. Although topic maps and RDF can also be represented in XML, many have held that their many possible syntactic forms make them extremely difficult to query using an XML query language, and that they can only be queried using special-purpose query languages with built-in knowledge of their semantics, including the ability to exploit RDF schema information. This talk shows that XQuery can, in fact, be used to solve the kinds of queries for which RDF and topic map query languages were designed, though with a loss of type safety.

The approach taken is to transform instances of RDF and topic maps to a syntactic representation that closely models their underlying logical models, and to use function libraries written in XQuery to directly support operations specific to RDF or topic maps. Schema level information is also incorporated in this representation, and is supported in the library, so queries can exploit type hierarchies and perform joins across predicates.

Information from other XML sources can also be queried together with information from topic maps and RDF. For instance, a query on a topic maps that searches for Shakespeare plays mentioned in Italian operas can also query the plays themselves -- represented in XML -- to determine which Italian cities are mentioned in them.

Syntax is not the opposite of semantics, it is a medium for semantics.

1. Introduction

XML is an extremely versatile markup language, capable of labeling the information content of diverse data sources including structured and semi-structured documents, relational databases, and object repositories. Software systems that use the structure of XML intelligently can operate on all these kinds of data, whether physically stored in XML or viewed as XML via middleware. Data integration is fundamentally important to XML query language community, and many see XQuery as a universal query language for all the data sources that can be represented in a universal data format, XML.

Integration is just as important for the web per se. In the future, we expect the web to include vast amounts of XML data, including both data that is physically represented as XML and XML views of other data sources such as traditional databases. In addition, we expect the web to include descriptive information represented as RDF or topic maps, designed to facilitate searching, automated processing, and syndication. If the future web is likely to include XML, topic maps, and RDF, then we must build software architectures that are able to manipulate and correlate information from all of these sources. If we do not, we must reconcile ourselves to two or more separate webs, each containing only part of the information, and each requiring its own set of tools.

This paper uses one query language, XQuery, to access information from XML, topic map, and RDF, allowing information from all three sources to be correlated. Every code example shown in this paper is in pure XQuery, without any augmentation. The XQuery language has no knowledge of RDF or topic maps, but is based directly on XML. Although both RDF and topic maps may be serialized as XML, their underlying data models are quite different from that of XML, and bridging these differences is a challenge. We believe that this paper represents a first serious attempt to address this challenge, producing results that make us optimistic that this challenge can largely be met, and pointing the way for future efforts toward an integrated web.

Part of this challenge is purely syntactic - both topic maps and RDF allow many different syntactic forms for the same information, and searching for all the possible ways that a fact may be represented on the syntactic level is quite difficult. Since XQuery operates directly on the XML representation, the syntactic form of the data strongly affects the complexity and run-time cost of queries. Much of this paper is based a simple trick. To enable XQuery to query topic maps and RDF, we have defined syntactic structures that closely reflect their underlying models, and defined transformations that generate these normalized forms. Query languages designed specifically for RDF or topic maps generally transform the syntactic representation to an internal format that reflects the underlying model - the main difference is that we then give this underlying model a syntactic form by representing it in XML. Among the various syntactic representations RDF and topic maps can take, there should be one that is easy to query and easy for humans to understand. The normalized forms we use in this paper are very preliminary, and do not meet many important criteria. What we have gained from this effort, however, is a preliminary understanding of what kinds of normal forms may be most suitable, and a better understanding of criteria that should be taken into account.

A second part of this challenge is semantic. An XML representation of the class hierarchies and associations of RDF and topic maps contains no knowledge of the underlying model. If we wish to be able to support basic operations on the class hierarchies and associations in our data, we must design a function library for XQuery that incorporates this knowledge. At this point, we believe we have demonstrated representative functionality with this approach, and that our function library could be extended to provide more complete functionality.

A third part of this challenge involves strong typing of query results. In this paper, we have not yet begun to seriously address this issue. Our libraries and normal forms lose basic type information, such as the distinction between literal data and resources. We hope to address several obvious type information losses in future versions of our library and normal forms. However, some degree of type information loss may be inherent when we query information belonging to one type system with a query language based on a different type system. The extent to which we can compensate for this with mechanisms available in XML and XQuery is left for future work. For now, we merely point out that the ability to integrate information across sources is important enough to offset this loss of type information for many applications.

Although our results are preliminary, we show practical, working examples that can query all three kinds of data in a straightforward manner. We hope this will encourage the RDF, topic map, and XML communities to work more closely to produce more complete solutions.

2. Syntax and Semantics

The title of this paper is "The Syntactic Web: Syntax and Semantics on the Web". This title was not chosen as a criticism of the semantic web, but to emphasize that there should be a straightforward syntactic representation of logical models so that general purpose tools can leverage the content of the semantic web. Syntax and semantics are not separate enterprises - they go hand in hand. Every syntax should have a clear underlying semantics, and every semantics should have a syntactic form that clearly expresses its underlying model. This should include schema-level information as well as instance-level information; for instance, our RDF internal format represents the entire RDF schema with assertions, making it possible for schema-level constructs to be evaluated in XQuery.

Terms such as "semantics", "meaning", and "understands" are understood differently in various technical communities and have often been a source of confusion. In general, we will avoid using these terms, in the hope that this paper will not add to the confusion. Explaining the various ways that the these terms have been used in the XML, semantic web, and XML query language communities would be a paper in its own right, and we prefer not to undertake that task in this paper. Instead, we will show the kinds of queries that the topic map and RDF communities have promoted for their models, demonstrate that an XML-based query language can perform these queries well, and show a scenario that uses queries to correlate information from these models with information from XML documents from other domains.

3. Querying Normalized RDF

RDF has many syntactic forms for representing the same logical content. Some of these forms are structurally quite distinct. For instance, the RDF specification contains the following three equivalent representations of the sentence "The individual referred to by employee id 85740 is named Ora Lassila and has the email address lassila@w3.org. The resource http://www.w3.org/Home/Lassila was created by this individual."

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator rdf:resource="http://www.w3.org/staffId/85740"/>
  </rdf:Description>

  <rdf:Description about="http://www.w3.org/staffId/85740">
    <v:Name>Ora Lassila</v:Name>
    <v:Email>lassila@w3.org</v:Email>
  </rdf:Description>
</rdf:RDF>
```

The representations we give below represents the same assertions with an XML structure that is quite different. The following representation is based on nested elements:

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator>
      <rdf:Description about="http://www.w3.org/staffId/85740">
 <v:Name>Ora Lassila</v:Name>
 <v:Email>lassila@w3.org</v:Email>
      </rdf:Description>
    </s:Creator>
  </rdf:Description>
</rdf:RDF>
```

And here is a representation that represents the same information using attributes:

```
<rdf:RDF>
  <rdf:Description about="http://www.w3.org/Home/Lassila">
    <s:Creator rdf:resource="http://www.w3.org/staffId/85740"
       v:Name="Ora Lassila"
       v:Email="lassila@w3.org" />
  </rdf:Description>
</rdf:RDF>
```

According to the RDF specification, a machine using RDF sees no difference between the above three forms. In order to make this true for XQuery, we first convert RDF assertions to a normal form that eliminates the syntactic and structural variation. In our normal form, every set of RDF assertions is presented as a list of statements, and each statement has one subject, one object, and one predicate. The following statements are equivalent to each of the RDF samples shown above:

```
<statement>
 <subject>http://www.w3.org/Home/Lassila</subject>
 <predicate>s:Creator</predicate>
 <object>http://www.w3.org/staffId/85740</object>
</statement>
<statement>
 <subject>http://www.w3.org/staffId/85740</subject>
 <predicate>v:Name</predicate>
 <object>v:Ora Lassila</object>
</statement>
<statement>
 <subject>http://www.w3.org/staffId/85740</subject>
 <predicate>v:Email</predicate>
 <object>lassila@w3.org</object>
</statement>
```

In this paper, we will call statements in the above normal form "triples". During the writing of this paper, we have produced such triples using our own XQuery library, using XSLT transformations written by Jonathan Borden, or by using SiRPAC. At times we have done the transformation during the query itself, using a call to the XQuery library. At other times we have pre-cooked the triples.

Whatever means is chosen to convert to triples, the main point is to create a common representation for each of the forms. For instance, an element name may be the predicate of an RDF statement, and its content may be the object. The following XQuery function converts an RDF assertion in that form to a triple:

```
DEFINE FUNCTION element-to-triple(charstring $about, ELEMENT $s)
 RETURNS statement
{
 <statement>
  <subject>{$about}</subject>
     <predicate>{name($s)}</predicate>
  <object>{string($s)}</object>
 </statement>
}
```

In RDF, the same information might be represented using attributes instead of elements. The following function converts the RDF assertions written as attributes to triples:

```
DEFINE FUNCTION attributes-to-triples(ELEMENT $about, ELEMENT $s)
 RETURNS ListOfStatement
{
 FOR $i IN $s/@*
 WHERE not(name($i) = "rdf:about" or name($i) = "rdf:ID")
 RETURN
  <statement>
   <subject>{ string($about) }</subject>
   <predicate>{ name($i) }</predicate>
   <object>{ string($i) }</object>
  </statement>
}
```

Because triples eliminate the many possible syntactic and structural forms in RDF, they make querying RDF much more straightforward, and independent of the syntactic representation chosen by the author. For instance, if the function assertions() returns the set of available assertions, then the following query returns the creator of the URI "http://www.w3.org/Home/Lassila":

```
LET $triples := assertions(),
    $creator := $triples[subject="http://www.w3.org/Home/Lassila" AND object="Cre-
ator"]
RETURN $creator/predicate
```

And this query returns the email address of the creator of the URI "http://www.w3.org/Home/Lassila":

```
LET $triples := assertions(),
    $creator := $triples[subject="http://www.w3.org/Home/Lassila" AND object="Cre-
ator"],
    $email := $triples[subject=$creator/predicate AND object="Email"]
RETURN $email/predicate
```

Querying on the basis of triples works well for most simple examples, but many RDF queries involve multiple assertions about the same subject, resulting in many joins when this information is queried. We would like to illustrate this with some examples taken from an RDF description of museums, artists, and artifacts taken from a paper published by the authors of RQL [KCPA01]. We derived the following statements from the original RDF:

```
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>c:mime-type</predicate>
    <object>image/jpeg</object>
  </statement>
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>c:file_size</predicate>
    <object>20</object>
  </statement>
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>c:title</predicate>
    <object>Portrait of the artist at its easel</object>
  </statement>
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>rdf:type</predicate>
    <object>c:ExtResource</object>
  </statement>
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>c:exhibited</predicate>
    <object>http://www.louvre.fr/</object>
  </statement>
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>c:technique</predicate>
    <object>oil on canvas</object>
  </statement>
  <statement>
    <subject>http://www.artchive.com/rembrandt/artist_at_his_easel.jpg</subject>
    <predicate>rdf:type</predicate>
    <object>c:Painting</object>
  </statement>
```

We found that many queries want to use several assertions about the same resource, and we also found that it is relatively difficult for a human to read all the assertions pertaining to a given resource in order to verify query results. Therefore, we found it more convenient to merge all assertions about a given resource into one description. Here is the description that contains all statements given above:

```
    <description rdf:about="http://www.artchive.com/rem-
brandt/artist_at_his_easel.jpg">
 <rdf:type>c:Painting</rdf:type>
 <c:technique>oil on canvas</c:technique>
 <c:exhibited>http://www.louvre.fr/</c:exhibited>
 <rdf:type>c:ExtResource</rdf:type>
 <c:title>Portrait of the artist at its easel</c:title>
 <c:file_size>20</c:file_size>
 <c:mime-type>image/jpeg</c:mime-type>
    </description>
```

We will call a description of the above form a "merged description" in this paper. We create a merged description from a set of triples using the following XQuery. In this query, the return clause uses a computed tag name, creating an element whose name is the text of the predicate, and whose content is the text of the object:

```
    <rdf>
     {
     for $t in distinct(document("sirpac-culture-triples.rdf")//statement/subject)
      return
  <description about={$t/text() }>
   {
     for $s in document("sirpac-culture-triples.rdf")//statement
     where $s/subject=$t
     return
        <{$s/predicate/text()}>
   {
     $s/object/text()
   }
        </>
   }
  </description>
     }
    </rdf>
```

In a merged description, the subject is represented by the value of the rdf:about attribute, the predicate is represented by the element name of a child element, and the object is represented by the content of a child element. Note that the merged description contains both the properties of the instance (c:technique, c:title, c:file_size, and c:mime-type) and the RDF type information (rdf:type), so queries that combine type information with instance information are possible.

To illustrate queries on this representation, we will use queries proposed by the authors of RQL for the sample data we used above. The reason we choose these queries is that solutions are already available in two RDF-specific query languages, RQL and Squish, which makes comparisons of the languages much easier. We will compare the XQuery solutions to those of RQL and Squish. We asked the authors to provide a short introduction of each language. More complete information on these languages can be found in our bibliography.

RQL is a typed language following a functional approach (a la OQL) and supports generalized path expressions featuring variables on both labels for nodes (i.e., classes) and edges (i.e., properties). RQL relies on a formal graph model (opposed to triple-based approaches) that captures the RDF modeling primitives and permits the interpretation of superimposed resource descriptions by means of one or more schemas. The novelty of RQL lies in its ability to smoothly switch between schema and data querying while exploiting - in a transparent way - the taxonomies of labels and multiple classification of resources. The functionality and formal interpretation of RQL is given for several classes of useful queries required by Semantic Web Applications.

Squish is an query language for RDF, represented in an SQL-like syntax with SELECT, FROM and WHERE clauses. Squish queries are performed over the RDF model, not any particular RDF syntax, so the basic unit of query is a triple: predicate, subject, object, with none or more of these three elements represented by a variable. The general approach (although not the syntax) is inspired by the paper 'Enabling Inferencing' (http://www.w3.org/TandS/QL/QL98/pp/enabling.html) a position paper for the W3C Query Languages meeting in Boston, December 3-4th 1998. The syntax is close to that used by Guha's RDFDB query language (http://web1.guha.com/rdfdb/query.html). The general principle is to specify an RDF graph with parts missing; the Java Squish implementation (Inkling) returns the bound results as a table within the JDBC Java database interface or as an RDF Graph object.

The first set of queries are taken from the RQL demo found at the following URL: http://139.91.183.30:9090/RDF/RQL/rqldemo.html.

```
Q1: Find the resources having a title

RQL:
    select X,Y from {X}title{Y}

Squish:

    SELECT ?resource
    FROM
    http://139.91.183.30:9090/RDF/RQL/demo/culture.rdf
    WHERE
    (c::title ?resource ?title)
    USING
    c FOR http://www.oclc.org/schema.rdf#

XQuery - using triples:

    LET $t := document("sirpac-culture-triples.rdf")//statement
    RETURN $t[predicate="c:title"]

XQuery - using merged descriptions:

    NAMESPACE c = "www.example.com/culture"
    LET $d := document("sirpac-culture-merged.rdf")//description
    RETURN $d[c:title]
```

To enable fair comparison, a few differences in the information content, form, and results of the above queries should be mentioned. In the XQuery solutions, we use a form with multiple clauses that explicitly references the source of the RDF assertions. The above query is easy to express in a single clause, and it is not necessary to state the data source if it is known to the query environment. The query based on triples could also have been expressed more compactly as follows:

```
    //statement[predicate="c:title"]
```

The query based on merged descriptions could have been expressed as follows:

```
    NAMESPACE c = "www.example.com/culture"
    //description[c:title]
```

We will generally use queries based on FLWR expressions in XQuery in this paper.

Note that the XQuery solution based on merged descriptions must declare the namespace for title. The solution based on triples is simply matching the text "c:title", but this is due to a limitation in current state of the implementation. If the XQuery implementation provided full support for XML Schema, and the datatype of object were QName, then prefix expansion should be supported for both representations.

The results of the above queries are not precisely equivalent. The XQuery example based on triples returns results as a series of statements - here is an excerpt of the results:

```
  <statement>
    <subject>http://www.museum.es</subject>
    <predicate>c:title</predicate>
    <object>Reina Sofia Museum</object>
  </statement>
  <statement>
    <subject>http://www.rodin.fr</subject>
    <predicate>c:title</predicate>
    <object>Rodin Museum</object>
  </statement>
```

The XQuery solution based on merged descriptions returns the entire merged descriptions, which include the titles. For instance, here is one result in this format:

```
  <description about="http://www.museum.es">
    <rdf:type>c:Museum</rdf:type>
    <rdf:type>c:ExtResource</rdf:type>
    <c:title>Reina Sofia Museum</c:title>
    <c:last_modified>2000/06/09</c:last_modified>
  </description>
```

It is possible, in XQuery, to return results that are not well-formed. For instance, to return only the subject and the title, in a query based on triples, we could use the following query:

```
    LET $t := document("sirpac-culture-triples.rdf")//statement,
        $r := t[object="c:title"]
    RETURN ($r/subject, $r/object)
```

One of the advantages of XQuery, in fact, is that it can return results in any desireable XML format. However, in most of our examples, we have found that returning the entire triple or merged description is convenient, and we will generally do so in the rest of the paper. The differences we have described for solutions to the first query generalize to solutions for further queries. We now return to exploring these queries, starting with a query requiring joins across resources.

The following query returns resources which are museums, according to their rdf:type, together with their titles. Note that this query uses a condition that combines RDF metadata with RDF property data.

```
Q2: Find the Museum resources and their titles

RQL:
     select X,Y from Museum{X}.title{Y}

Squish:
    SELECT ?resource, ?title
    FROM
    http://139.91.183.30:9090/RDF/RQL/demo/culture.rdf
    WHERE
    (rdf::type ?resource c:Museum)
    (c::title ?resource ?title)
    USING
    c FOR http://www.oclc.org/schema.rdf#
    rdf FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#


XQuery - using triples:

    LET $t := document("sirpac-culture-triples.rdf")//statement

    FOR $museum in $t[predicate="rdf:type" and object="c:Museum"],
        $title in $t[predicate="c:title"]
    WHERE $museum/subject=$title/subject
    RETURN $museum

XQuery - using merged descriptions:

    LET $t := document("sirpac-culture-merged.rdf")//description
    RETURN $t[rdf:type="c:Museum"]
```

Note that the XQuery using merged descriptions is much simpler than the one based on triples, because it does not need to do a join to find assertions based on the same subject. Again, it returns all the assertions about the subjects:

```
  <description about="http://www.museum.es">
    <rdf:type>c:Museum</rdf:type>
    <rdf:type>c:ExtResource</rdf:type>
    <c:title>Reina Sofia Museum</c:title>
    <c:last_modified>2000/06/09</c:last_modified>
  </description>
  <description about="http://www.louvre.fr/">
    <rdf:type>c:Museum</rdf:type>
    <rdf:type>c:ExtResource</rdf:type>
    <c:title>Louvre Museum</c:title>
  </description>
  <description about="http://www.rodin.fr">
    <rdf:type>c:Museum</rdf:type>
    <rdf:type>c:ExtResource</rdf:type>
    <c:title>Rodin Museum</c:title>
    <c:last_modified>2000/02/01</c:last_modified>
  </description>
```

If the assertions should be limited to the title, it can be done by using the RETURN clause to construct an appropriate result:

```
    LET $t := document("sirpac-culture-merged.rdf")//description
    FOR $d IN  $t[rdf:type="c:Museum"]
    RETURN
     <description rdf:about={ @d/rdf:about } >
      {
        $d/c:title
      }
     </description>
```

Here is the output of the above query.

```
  <description about="http://www.museum.es">
    <c:title>Reina Sofia Museum</c:title>
  </description>
  <description about="http://www.louvre.fr/">
    <c:title>Louvre Museum</c:title>
  </description>
  <description about="http://www.rodin.fr">
    <c:title>Rodin Museum</c:title>
  </description>
```

So far, our queries have not required RDF schema information. The following query requires the ability to exploit RDF type hierarchies, and to perform joins against the range and domain of an RDF property.

```
Q3: Find the names of those who have created artifacts which are exhibited in
Museums, along with the Museum titles

RQL:

    select Y,Z, V, R
    from {X}creates.exhibited{Y}.title{Z}, {W}first_name{V}, {Q}last_name{R}
    where X=W and X=Q

Squish:

    SELECT ?y, ?z, ?v, ?r
    FROM
    http://139.91.183.30:9090/RDF/RQL/demo/culture.rdf
    WHERE
     (rdf::type ?y ns::Museum)
     (c::title ?y ?z)
     (c::exhibited ?a ?y)
     (?prop ?b ?a)
     (c::first_name ?b ?v)
     (c::last_name ?b ?r)
    c FOR http://www.oclc.org/schema.rdf#
    rdf FOR http://www.w3.org/1999/02/22-rdf-syntax-ns#
```

The XQuery solution will be developed during the course of this section. Following the RQL solution, the XQuery solution willuse the range and domain of the "creates" property, as well as the properties derived from it. In the merged descriptions, this property and the properties derived from it are represnted as follows:

```
 <description rdf:about="c:creates">
  <rdf:type>rdf:Property</rdf:type>
  <rdfs:domain>c:Artist</rdfs:domain>
  <rdfs:range>c:Artifact</rdfs:range>
 </description>
 <description rdf:about="c:paints">
  <rdf:type>rdf:Property</rdf:type>
  <rdfs:domain>c:Painter</rdfs:domain>
  <rdfs:range>c:Painting</rdfs:range>
  <rdfs:subPropertyOf>c:creates</rdfs:subPropertyOf>
 </description>
 <description rdf:about="c:sculpts">
```

```
  <rdf:type>rdf:Property</rdf:type>
  <rdfs:domain>c:Sculptor</rdfs:domain>
  <rdfs:range>c:Sculpture</rdfs:range>
  <rdfs:subPropertyOf>c:creates</rdfs:subPropertyOf>
 </description>
```

Since the "creates" property connects artists to the artifacts they created, and the "exhibited" property connects artifacts to the museums where they are displayed, linking these two relationships gives us much of the functionality we need to perform the query. As an XML-based language, XQuery has no built-in knowledge of RDF properties, but since we represent the RDF schema information in our normalized function, we can write a function library that does. First, we need functions that return the range and domain of a predicate:

```
    define function rdf:predicate-range(ListOfDescription $d, charstring $predicate-
name)
     RETURNS rdfs:range
    {
     $d[@rdf:about=$predicate-name and rdf:type="rdf:Property"]/rdfs:range
    }

    define function rdf:predicate-domain(ListOfDescription $d, charstring $predicate-
name)
     RETURNS rdfs:domain
    {
     $d[@rdf:about=$predicate-name and rdf:type="rdf:Property"]/rdfs:domain
    }
```

The above functions can tell us that the domain of "creates" is "c:Artist" and the range is "c:Artifact", but we need to be able to find specific artists like Rodin or Picasso to find the works they created. In other words, we need a function that returns the instances of a given class. In our merged representation, instances contain a sub-element named "rdf:type", and this sub-element contains the name of the class to which the instance belongs:

```
 <description rdf:about="c:rodin424">
  <rdf:type>c:Sculptor</rdf:type>
  <c:creates>http://www.artchive.com/crucifixion.jpg</c:creates>
  <c:last_name>Rodin</c:last_name>
  <c:first_name>August</c:first_name>
 </description>
 <description rdf:about="c:picasso132">
  <rdf:type>c:Painter</rdf:type>
  <c:paints>http://www.museum.es/guernica.jpg</c:paints>
  <c:paints>http://www.museum.es/woman.qti</c:paints>
  <c:first_name>Pablo</c:first_name>
  <c:last_name>Picasso</c:last_name>
 </description>
```

In this data, however, the "rdf:type" element for these two artists does not contain "c:Artist"; instead, it contains the types "c:Sculptor" and "c:Painter", which are derived from "c:Artist". Fortunately, our merged description contains further RDF Schema information that allows us to find the entire class hierarchy for "c:Artist". Here is an excerpt of this hierarchy:

```
 <description rdf:about="c:Artist">
  <rdf:type>rdfs:Class</rdf:type>
```

```
</description>
<description rdf:about="c:Sculptor">
 <rdf:type>rdfs:Class</rdf:type>
 <rdfs:subClassOf>c:Artist</rdfs:subClassOf>
</description>
<description rdf:about="c:Painter">
 <rdf:type>rdfs:Class</rdf:type>
 <rdfs:subClassOf>c:Artist</rdfs:subClassOf>
</description>
```

Now it is clear that the function which finds all instances of a class must search for instances of every class derived from the initial class as well. Here is a function that does this by calling itself recursively for every subclass:

```
    define function rdf:instance-of-class(ListOfDescription $t, charstring $base-
name)
     RETURNS ListOfDescription
    {
     $t[rdf:type = $base-name]
     ,
     for $i in $t[rdfs:subClassOf = $base-name]
     return rdf:instance-of-class($t, string($i/@rdf:about))
    }
```

The above function returns all descriptions which have the rdf:type indicated by the $base-name parameter, then calls itself recursively for each sub-class of that type. We have already seen that the "c:creates" property is also part of a class hierarchy, and although our two artists do not contain the "c:creates" property, they do contain a "c:sculpts" or "c:paints" property. When we retrieve an artist, we need a function that will return the works created by that author, regardless of the medium. In other words, we need a function that, given a description, returns all properties derived from a given property. This function is quite similar to the last - it first returns any properties that have the original property name, then it calls itself recursively for each property derived from it:

```
    define function rdf:property(ListOfDescription $t, description $d, charstring
 $property-name)
     returns element
    {
     $d/*[name(.) = $property-name]
     ,
     for $i in $t[rdfs:subPropertyOf = $property-name]
     return rdf:property($t, $d, string($i/@rdf:about))
    }
```

This function is similar to the previous function. First, it returns all child elements of the parameter $d whose element names are the same as $property-name, then it calls itself recursively for each sub-property. Finally, we need a function that allows us to join a description in the domain of a property to the corresponding descriptions in the range of the property:

```
    define function rdf:join-on-property(ListOfDescription $t, description $d,
charstring $predicate-name)
     returns ListOfDescription
    {
     FOR $i IN rdf:instance-of-class($t, rdf:predicate-range($t, $predicate-name))
     WHERE $i/@rdf:about=rdf:property($t, $d, $predicate-name)
     RETURN $i
    }
```

Given the above functions, we can find the artists and the museums they exhibit in, returning the same four pieces of information returned by the corresponding RQL query:

```
    LET $t := document("sirpac-culture-merged.rdf")//description

   FOR $artist IN rdf:instance-of-class($t, rdf:predicate-domain($t, "c:creates"))
    LET $artifact := rdf:join-on-property($t, $artist, "c:creates"),
 $museum := rdf:join-on-property($t, $artifact, "c:exhibited")
    RETURN
 <result>
        {
     $artist/c:last_name,
     $artist/c:first_name,
     $museum/@rdf:about,
     $museum/c:title
        }
 </result>
```

The above query is precisely equivalent to the corresponding RQL query, but the results are somewhat odd, since they lose the original grouping and return an attribute of museum as an attribute of the result:

```
 <result rdf:about="http://www.rodin.fr">
    <c:last_name>Rodin</c:last_name>
    <c:first_name>August</c:first_name>
    <c:title>Rodin Museum</c:title>
  </result>
  <result rdf:about="http://www.louvre.fr/">
    <c:last_name>Buonarroti</c:last_name>
    <c:first_name>Michelangelo</c:first_name>
    <c:title>Louvre Museum</c:title>
  </result>
  <result rdf:about="http://www.museum.es">
    <c:last_name>Picasso</c:last_name>
    <c:first_name>Pablo</c:first_name>
    <c:title>Reina Sofia Museum</c:title>
  </result>
```

An easy way to retain grouping is to use the filter() function, which is part of the native library of XQuery:

```
    LET $t := document("sirpac-culture-merged.rdf")//description

   FOR $artist IN rdf:instance-of-class($t, rdf:predicate-domain($t, "c:creates"))
    LET $artifact := rdf:join-on-property($t, $artist, "c:creates"),
 $museum := rdf:join-on-property($t, $artifact, "c:exhibited")
    RETURN
 <result>
   {
     filter($artist | $artist/c:last_name | $artist/c:first_name),
     filter($museum | $museum/c:title)
   }
 </result>
```

The output of the above query is as follows:

```
<result>
 <description rdf:about="c:rodin424">
  <c:first_name>August</c:first_name>
  <c:last_name>Rodin</c:last_name>
 </description>
 <description rdf:about="http://www.rodin.fr">
  <c:title>Rodin Museum</c:title>
 </description>
</result>
<result>
 <description rdf:about="c:michelangelo">
  <c:first_name>Michelangelo</c:first_name>
  <c:last_name>Buonarroti</c:last_name>
 </description>
 <description rdf:about="http://www.louvre.fr/">
  <c:title>Louvre Museum</c:title>
 </description>
</result>
<result>
 <description rdf:about="c:picasso132">
  <c:first_name>Pablo</c:first_name>
  <c:last_name>Picasso</c:last_name>
 </description>
 <description rdf:about="http://www.museum.es">
  <c:title>Reina Sofia Museum</c:title>
 </description>
</result>
```

3.1. Summary: Querying RDF with XQuery

In the above queries, we have shown that an XML query language can query RDF much more conveniently if represented using merged descriptions, which pre-compute joins that would otherwise need to be performed at run-time. We have also shown that an RDF function library can exploit the RDF schema if the assertions of the schema are also present in the merged descriptions. In this paper, we have written only a few functions, and a more complete function library would be extremely helpful. However, we believe that we have demonstrated representative and practical queries, showing that querying RDF with an XQuery library is not only possible, but reasonably straightforward. We need greater coverage before our results should be considered conclusive.

Although we are able to perform sophisticated queries, our RDF library is not type-safe with respect to RDF. For instance, it is helpful to compare the rdf:predicate-range() function to the following RQL query:

```
  bag(range(Artist)) Union subclassof(Artifact)
```

RQL returns a type error for this query, since the range function is defined on properties and not classes. Our rdf:predicate-range() function, if called using Artist as predicate-name, returns an empty result without signalling an error. The final query result is the evaluation of the second subquery (assuming that an appropriate XQuery function is defined). Type safety may be an important reason for native RDF query languages. Some of the loss of type information in our current implementation is gratuitous, and could be corrected in a future design. For instance, our merged representation could be changed to include more type information, perhaps using an XML Schema, and we should consider how much type information can be preserved by the operations in our type library. However, there are limits to the amount of type safety we can provide. For those who are interested in data integration, this disadvantage is offset by the clear advantage of being able to integrate information from RDF with topic maps and XML, and is an inherent tradeoff when working across models.

4. Normalized Topic Maps

In this section, we will proceed largely as we did for RDF, using a normalized representation of topic maps, performing queries that might typically be executed against a topic map, and developing a library as we go. In future work, we expect to compare multiple normalized forms of topic maps, but at the time of writing all queries are performed against a representation based on a topic map data model created by Lars Marius Garshol [http://www.ontopia.net/topicmaps/materials/proc-model.html]. The sample data we will use is a topic map of operas developed originally by Steve Pepper.

```
 <topic id="id506">
  <baseName>
   <baseNameString>composer</baseNameString>
  </baseName>
 </topic>
```

It is quite easy to query the topic map to find a topic by name in XQuery:

```
        document("opera.dxtm")//topic[baseName/baseNameString = "composer"]
```

However, in our library we would like our function to be a little more general. First, we will write it to allow a list of names to be specified. Second, we allow topic map scopes to be specified for the name - we will explain this in the first query that uses a scope. The complete function looks like this:

```
define function tm:get-topic-with-name(ListOfTopic $topics, ListOfName $names,
ListOfTopic $nameScope)
      returns ListOfTopic
{
 for $t in $topics
 where some $n in $names satisfies $n = $t/baseName/baseNameString
   and if (empty($nameScope))
        then true()
        else some $s in $nameScope/@id satisfies $s = $t/base-
Name/scope/topicRef/@href
 return $t
}
```

In the above function, if the $nameScope parameter is an empty list, then the second condition in the where clause evaluates to true(), and any topic whose baseNameString matches one of the given names is returned. If it is not empty, a name will only match if its scope matches one of the topics in the $nameScope.

Now we can use this function to find the topic representing the composer class. We do not wish to specify a scope, so we will pass the empty list, represented in XQuery as (), for the nameScope parameter:

```
    let $topics := document("opera.dxtm")//topic,
  $Composer := tm:get-topic-with-name($topics, "composer", () )
    return $Composer
```

This query simply returns the topic for the composer class, which we have already seen. Next we want to find all the composers; that is, all instances of a given class. In our representation, an instance always indicates the class to which it corresponds using an instanceOf element:

```
  <topic id="id222">
   <instanceOf href="id503"/>
   <instanceOf href="id623"/>
   <baseName>
    <baseNameString>Giuseppe Hagenbach</baseNameString>
   </baseName>
  </topic>
```

A topic is an instance of a class if any instanceOf element has an href that matches the id attribute of the class. Here is a function that finds all instances of a given class. Since topic maps also allow an instance to derive itself from another instance, this function calls itself recursively:

```
define function tm:get-instances(ListOfTopic $topics, topic $base)
  returns ListOfTopic
{
 for $t in $topics[instanceOf/@href=$base/@id]
 return (
   $t,
   tm:get-instances($topics, $t)
 )
}
```

Now we can use the two functions we have defined to find all instances of the class "composer":

```
    let $topics := document("opera.dxtm")//topic,
  $composerClass :=  tm:get-topic-with-name($topics, "composer", () ),
  $composers := tm:get-instances($topics, $composerClass)
    return $composers
```

If we are interested in composers, we may well be interested in finding the works that they composed. The relationship between a composer and the composer's works is represented with an association:

```
   <association>
     <instanceOf href="id505" />
     <member>
       <instanceOf href="id67" />
       <topicRef href="id1" />
     </member>
     <member>
       <instanceOf href="id506" />
       <topicRef href="id40" />
```

```
    </member>
  </association>
```

Associations have types in topic maps. This association says it is an instance of the topic whose id attribute is "id505", which looks like this:

```
    <topic id="id505">
     <baseName>
      <baseNameString>composed by</baseNameString>
     </baseName>
     <baseName>
      <scope>
       <topicRef href="id506"/>
      </scope>
      <baseNameString>composed</baseNameString>
     </baseName>
    </topic>
```

Retrieving associations based on the name of their class is much like retrieving topics based on the name of their class. Our function will need access to a list of topics and a list of associations to select from:

```
define function tm:get-association-with-type-name(ListOfTopic $topics, ListOfAsso-
ciation $associations, ListOfName $names, ListOfTopic $nameScope)
      returns ListOfTopic
{
    let $associationType := tm:get-topic-with-name($topics, $names, $nameScope)
 return $associations[instanceOf/@href=$associationType/@id]
}
```

We can call the above function to retrieve all instances of the "composed" association:

```
   tm:get-association-with-type-name($topics, $associations, "composed", ())
```

Given a set of associations, we may want to list the composers or their works. As we have seen above, an association contains a set of members. Each of these members represents one reference. It contains an instanceOf element to express the role that the reference plays, and a topicRef element that points to the topic:

```
    <member>
      <instanceOf href="id67" />
      <topicRef href="id1" />
    </member>
```

In our topic map, the topic with id "id67" is the class of operas:

```
 <topic id="id67">
  <instanceOf href="id494"/>
  <baseName>
   <baseNameString>opera</baseNameString>
  </baseName>
  . . .
 </topic>
```

This tells us that the above member points to an opera. The topicRef tells us which opera:

```
<topic id="id1">
 <instanceOf href="id67"/>
 <baseName>
  <baseNameString>Manon Lescaut</baseNameString>
 </baseName>
 . . .
</topic>
```

We will frequently want to find the topic that plays a particular role in an association, or the set of topics playing a role in a set of associations; for instance, given the "composed" association, we may well want to find the composer or the composition. The following function finds a list of topics playing a given role in a list of associations:

```
define function tm:get-topic-playing-role(ListOfTopic $topics, ListOfAssociation
$a, charstring $roleName, ListOfTopic $nameScope)
 returns ListOfTopic
{
 let $role := tm:get-topic-with-name($topics, $roleName, $nameScope)
 return distinct($topics[@id=$a/member[instanceOf/@href=$role/@id]/topicRef/@href])
}
```

For instance, we can use this to find the topics playing the "composer" role in a set of "composed" associations:

```
    let    $composed := tm:get-association-with-type-name($topics, $associations,
 "composed", ())
    return tm:get-topic-playing-role($topics, $associations, "composer", ())
```

The result of the above query is the set of composers who are associated with their compositions in this topic map.

Another common operation on topic maps is to ask for topics that reference a given resource. For instance, suppose Lyle Neff, who has written a number of web pages related to opera, decides he would like to know if his libretto for Macbeth is referenced by any topic in our topic map. To find out, we need a function that returns the topics that reference a given URI in their occurrences. Here is an example of such a topic:

```
<topic xmlns:ino="http://namespaces.softwareag.com/tamino/response2"
xmlns:xql="http://metalab.unc.edu/xql/" id="id20">
  <instanceOf href="id67" />
  <baseName>
    <baseNameString>Macbeth</baseNameString>
  </baseName>
  <occurrence>
    <instanceOf href="id604" />
    <resourceData>14 Mar 1847</resourceData>
  </occurrence>
  <occurrence>
    <instanceOf href="id583" />
    <resourceRef href="http://php.indiana.edu/~lneff/libretti/macbeth.html" />
  </occurrence>
```

```
    . . .
</topic>
```

Now we will write a function that searches for topics with a particular URI in their occurrences:

```
define function tm:get-topic-with-occurrence-uri(ListOfTopic $topics, ListOfUri
$resourceList, ListOfTopic $occurrenceScope)
  returns ListOfTopic
{
 for $t in $topics
 where some $r in $resourceList satisfies $r = $t/occurrence/resourceRef/@href
   and if (empty($occurrenceScope))
        then true()
        else some $s in $occurrenceScope/@id satisfies $s = $t/occur-
rence/scope/topicRef/@href
  return $t
}
```

The URI for the Macbeth libretto is "http://php.indiana.edu/~lneff/libretti/macbeth.html", and we can call the function as follows:

```
    let $topics := document("opera.dxtm")//topic,
 $associations := document("opera.dxtm")//association,
    return tm:get-topic-with-occurrence-uri($topics, "http://php.indi-
ana.edu/~lneff/libretti/macbeth.html", ())
```

At this point, we have shown several representative operations on topic maps, and have exploited information about class hierarchies, and have retrieved instances of a class. We would like to present a query that further exploits the schema level information in a topic map to print out the entire class hierarchy together with all instances of each class. We will begin with the root level classes; that is, with classes which are not subclasses of any other class. In a topic map, relationships between superclasses and subclasses are represented using associations. We can use tm:get-association-with-type-name() to return all associations representing derivations:

```
   tm:get-association-with-type-name($topics, $associations, "subclass of", ())
```

Given a set of derivations (or any set of associations that contains derivations), we can use the following function to return the root classes for a topic map:

```
define function tm:get-root-classes(ListOfTopic $topics, ListOfAssociation $deriv-
ations)
  returns ListOfTopic
{
 let $superclassRefs := tm:get-topic-playing-role($topics, $derivations, "super-
class", ()),
     $subclassRefs:= tm:get-topic-playing-role($topics, $derivations, "subclass",
 ())
 for $su in $superclassRefs
 where not(some $s in $subclassRefs satisfies $s = $su)
 return $topics[@id=$su/@id]
}
```

The following query calls the above function:

```
    let $topics := document("opera.dxtm")//topic,
 $associations := document("opera.dxtm")//association,
 $derivations := tm:get-association-with-type-name($topics, $associations, "subclass
 of", ())
    return tm:get-root-classes($topics, $derivations)
```

For the opera topic map, it returns the following topics:

```
  <topic id="id598">
    <baseName>
      <baseNameString>person</baseNameString>
    </baseName>
  </topic>
  <topic id="id599">
    <baseName>
      <baseNameString>place</baseNameString>
    </baseName>
  </topic>
  <topic id="id629">
    <baseName>
      <baseNameString>work</baseNameString>
    </baseName>
  </topic>
```

Only two instances in this topic map are directly derived from one of the above three classes. To find most of the instances, we need a function that returns the classes derived from each root class, so that we can ask for instances of each of these classes. This function calls itself recursively to generate the entire set of classes in the hierarchy:

```
define function tm:get-derived-classes(ListOfTopic $topics, ListOfAssociation
$derivations, topic $base)
  returns ListOfTopic
{
 let $a := tm:get-association-by-topic-role($topics, $derivations, $base, "super-
class")
 for $subclass in tm:get-topic-playing-role($topics, $a, "subclass", ())
 return (
  $subclass,
  tm:get-derived-classes($topics, $derivations, $subclass)
 )
}
```

We now have all the functions we need to print out the entire class hierarchy, together with the instances of each class:

```
    let $topics := document("opera.dxtm")//topic,
 $associations := document("opera.dxtm")//association,
 $derivations := tm:get-association-with-type-name($topics, $associations, "subclass
 of", ())
    for $root in tm:get-root-classes($topics, $derivations)
    let $derived := tm:get-derived-classes($topics, $derivations, $root)
    return
      <root>
       {
   $root,
   <instances>{ tm:get-instances($topics, $root) } </instances>,
   for $d in $derived
   return
      <derived>
       {
    for $d in $derived
    return ($d,
     <instances>
     {
        tm:get-instances($topics, $d)
     }
     </instances>
    )
       }
      </derived>
       }
      </root>
```

4.1. Summary: Querying Topic Maps with XQuery

In this section, we have performed a number of representative queries on topic maps, including queries based on schema-level information. We find that topic maps can also be queried in a straightforward manner with XQuery, and the functions we have written are quite similar to those we used to query RDF. We wonder to what extent a common set of function names and signatures could be used to query both sources. Differences in the underlying model may make this difficult - for instance, RDF has unidirectional binary associations while topic maps have bidirectional N-ary associations, and the nature of scope in topic maps does not have an equivalent in RDF. Nevertheless, it may be possible to design libraries that have many functions in common, and to design the parts that must differ such that the two libraries are easy to use together.

At this point, we have only been able to create a data set for one normalized representation of topic maps. We believe it would be useful to compare several representations based on ease of use for queries and readability. In the functions we have written, many joins are performed, and this affects the speed of our library. In the future, work comparing the performance characteristics of various physical and logical representations would be very helpful.

4.2. Conclusion

We believe that we have shown the feasability of using XQuery libraries to query both RDF and topic maps, making it possible to correlate information from both these sources with XML. Now it is time for members of all three communities to work together to design these libraries and to create normalized syntactic representations that are reasonably convenient to process using tools like XQuery, XSLT, and the DOM. There should be just one web, and we should carefully consider how syntax and the various kinds of semantics can be coordinated.

Acknowledgements

The following people have contributed significantly to this paper, which I could not have written without their help: Lars Marius Garshol, Steve Newcomb, Michel Biezinski, Matthew Fuchs, Libby Miller, Dan Brickley, Vassilis Christophides, and Gregorius Karvounarakis.

They do not all agree with the conclusions I have drawn.

Bibliography

[Inferencing] R.V. Guha, Ora Lassila, Eric Miller, Dan Brickley. *Enabling Inferencing* http://www.w3.org/TandS/QL/QL98/pp/enabling.html

[XQuery] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, J me Sim , Mugur Stefanescu, editors, *XQuery, An XML Query Language* http://www.w3.org/TR/xquery

[SAF] Lee Buck, Jonathan Robie, Scott Vorthmann. *Schema Adjunct Framework* ### This will be published as a W3C Note very soon, but the URL is not out yet ###

[KCPA01] G. Karvounarakis, V. Christophides, D. Plexousakis, S. Alexaki *Querying RDF Descriptions for Community Web Portals* The French National Database Conference, BDA'2001 29 octobre - 2 novembre 2001 Agadir, Maroc

[ACKP01] S. Alexaki, V. Christophides, G. Karvounarakis, D. Plexousakis. *On Storing Voluminous RDF Descriptions: The case of Web Portal Catalogs* 4th International Workshop on the the Web and Databases (WebDB2001 - in conjunction with ACM SIGMOD/PODS).

[TMDM] Lars Marius Garshol *A Topic Map Data Model, An infoset-based proposal* http://www.y12.doe.gov/sgml/sc34/document/0229.htm

[Squish] Libby Miller *RDF Squish query language and Java implementation* http://ilrt.org/discovery/2001/02/squish/

[TMQL-req] Hans Holger Rath, Lars Marius Garshol, editors. *TMQL requirements (1.0.0)* http://groups.yahoo.com/group/tmql-wg/files/official-docs/tmqlreqs.html

Biography

Jonathan **Robie**
Software AG
U.S.A.

Jonathan Robie is an XML Research Specialist at Software AG. He is an editor of XQuery, the W3C XML Query Language, and also an editor of the W3C XML Query Requirements, Use Cases, and Data Model. He was also a co-author of two earlier XML query languages: Quilt, a precursor of XQuery, and XQL, a precursor of XPath. In addition to his work on

XML query languages, he is a member of the W3C XML Schema Working Group, where he is an editor of the Schema Formalization document, and was an editor of the W3C Document Object Model for both Level 1 and Level 2.

Mr. Robie has been on the architectural team for XML databases or repositories at three different companies - Software AG, Texcel Research, and POET Software. He has been a regular speaker at SGML and XML conferences since 1996, and a regular speaker at object oriented developer's conferences from 1991 to 1995.

Prior to his work with XML, Mr. Robie was an object database specialist at POET Software, where he implemented transactions in the kernel, designed and presented workshops on object database programming, and provided key-customer support. Before that, he worked as a free-lance consultant for five years. Mr. Robie has a M.S. in Computer Science from Michigan State University.