Querying Embedded RDF Data with XML Technology: A Feasibility Study

Norman May, Heiner Stuckenschmidt
norman.may@sap.com, heiner@informatik.uni-mannheim.de

Abstract: XML has become the de facto standard for representing and accessing data on the Web. At the same time RDF is becoming more and more popular for representing metadata. While RDF also has an XML-based syntax, storage and query technologies for the two formats are not compatible due to differences in the data model. This is a potential problem when trying to query data that combine XML data with RDF-based metadata annotations. In this paper, we investigate the feasibility of querying such embedded RDF models with XML technologies. We motivate the problem using the vision of intelligent content objects, describe an approach for querying RDF data with XQuery and identify problems and opportunities based on experiments with real world data.

1 Motivation

With the increasing importance of the Web, annotation languages for semi-structured information have rapidly gained importance. The extensible markup language (XML) is definitely the most influential development in this direction, and a lot of attention has been paid on the development of technologies for storing and querying large amounts of XML data. Developments in this direction are mostly driven by the database community that has proposed XPath and XQuery as standard languages for interacting with XML data. A number of systems have been developed that support these languages (e.g. Tamino, MonetDB, Berkeley DB, or Natix) and some fundamental research has been done on query optimization, query execution, and XML storage for efficient retrieval of XML-based information. In parallel, the Resource Description Framework (RDF) has been proposed as suitable data model for representing and reasoning about metadata on the Web. While RDF comes with an XML-based syntax, it turns out that the underlying data models of XML and RDF are different and not really compatible with each other. In fact, the RDF community has spend significant effort to argue that RDF has fundamental differences compared to XML [DvHF+00]. Special technologies for storing and retrieving RDF data have been developed completely independent from XML technologies. Meanwhile the W3C standardizes an RDF query language, SPARQL, and a number of RDF systems exist that implement this standard or variations of it (e.g. JENA, Joseki, RDFStore and Sesame). As a result, there currently is a strict separation of XML and RDF data when it comes to storing and querying data sources in the two representations.

As we will argue in this paper, there are numerous important use cases that would benefit from a unified view on both worlds. Our results also show that an elegant and reasonable efficient solution can be crafted to satisfy the requirements of these use cases.

Intelligent Content Objects The clear separation of XML and RDF data that is dictated by the limitations of existing technology on both sides has some serious limitations with respect to many application scenarios. Recently the notion of intelligent content objects has gained a lot of attention, mostly in the connection with emerging multimedia standards such as MPEG-7. The idea of intelligent contents is that of self-contained information that does not only consist of the raw data, but also contains metadata describing the data. In the case of video information, for example an MPEG 7 file will not only contain the video data, but also a summary of the contents, information about the producer and descriptions of scenes and characteristic features in the video that support automatic content analysis. While most of an MPEG 7 file will be in XML format, it makes sense to represent content-related metadata using RDF descriptions [WK03]. In other multimedia standards, in particular SMIL 3.0, RDF is already used inside an XML file to represent metadata [BD06]. The question we address in this paper is how to store and query descriptions like the one above that consist of an XML document with embedded metadata in RDF format.

Querying Embedded RDF There are different options for dealing with the kinds of documents shown above. In particular, these options are:

- We could use a specialized system that is tuned to a particular standard such as MPEG-7 or SMIL. This is what we currently see in the multimedia area.
- We could regard the RDF part as a special data type and treat it as a black box that can be accessed using special functions. This approach is supported by some commercial database vendors to combine RDF with the relational model.
- We could extract the RDF-based metadata from the XML file and store and query it in a dedicated RDF system separately from the XML part of the document.

All of the these options are unsatisfactory for many practical situations. Specialized systems have a rather narrow application area and are often inflexible with respect to major changes in the standard. Treating the RDF part as a black-box has the major disadvantage that links between different embedded models possibly in different files cannot be treated adequately. This is a major drawback as the most important feature of RDF is the ability to specify links between entities and models. Currently the most viable option seems to be to extract the metadata parts, process it separately from the XML document and integrate the results a posteriori.

From a conceptual point of view, separating the evaluation of the XML and RDF data is not a good idea because we lose the ability to evaluate or optimize queries across the different parts of the document. Further, the extraction and upload of RDF data into an external RDF storage as well as the re-integration of the results into a coherent answer set comes with additional overhead. Ideally, we want to use the same language with a single execution plan on the whole document. An obvious idea is to use the fact that RDF is represented in an XML syntax and use XML technologies to query this part of the information as well. This idea has been investigated in some early work on RDF but was dropped eventually because the work claimed that querying RDF data with languages such as XQuery is infeasible due to the differences in the underlying data model. To the best of our knowledge, however, corresponding experiments that clearly support this claim where

never published. Our aim is to fill this gap by experimentally investigating the feasibility of querying RDF data with XQuery as a basis for accessing intelligent content objects that contain embedded RDF metadata. The contribution of this paper is the following:

- We present a general approach for translating RDF queries into XQuery that covers a wide range of RDF query languages.
- We present performance experiments on real data sets.
- We discuss requirements and success factors of the approach.

In the following, we first present the general approach for translating RDF Queries into XQuery based on an abstract algebra. We then present a use case and experiments for querying real world data with XQuery. We conclude with a discussion of the feasibility of the approach.

2 Translating RDF Queries to XQuery

2.1 The RDF Query Model

RDF querying can be seen as a special form of graph matching, where the graphs to be matched have special properties. We use this view to get a language-independent representation of an RDF query. In particular, we use a slightly simplified version of the formal model of RDF queries proposed by Gutierrez et al. [GHM04]. In the following, we give some basic definitions connected to the abstract RDF query model:

Definition 1 (RDF Model) *Let U be a set of URI references, B is a set of blank nodes and L is a set of literals. An* RDF model *is a set of triples* $(s, p, o) \in (U \cup B) \times U \times (U \cup B \cup L)$.

Definition 2 (Instance of an RDF Model) *A* mapping *is a function* $\mu : (U \cup B \cup L) \rightarrow (U \cup B \cup L)$ *such that* $\mu(u) = u, \mu(l) = l$ *for all* $u \in U, l \in L$. *The mapping of an RDF model G is defined as*

$$\mu(G) = \{(\mu(s), \mu(p), \mu(o)) | (s, p, o) \in G\}$$

$\mu(G)$ *is called an* instance of G *if* $\mu(G)$ *is an RDF model in the sense of Definition 1.*

We can define RDF queries in a similar way. For this purpose, we extend Definition 1 for the case where some elements in a triple are variables rather than URIs, blank nodes, or literals.

Definition 3 (RDF Query) *Let V be a set of variables different from elements in* $(U \cup B \cup L)$. *An* RDF pattern *is a set of triple patterns* $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. *The set of all variables occurring in a pattern P is denoted as* $var(P)$, *the set of all blank nodes as* $blank(P)$. *A pattern* P *is* ground *if* $blank(P) = \emptyset$. *An* RDF query *is a pair of RDF patterns* $(H \leftarrow B)$ *where* $var(H) \subseteq var(B)$ *and B is ground. In this query,* H *is the head, and* B *is the body.*

The set of answers[1] for an RDF query can now be defined using the notion of an instance given in Definition 2.

Definition 4 (Query Answer) *Let $q = (H \leftarrow B)$ be an RDF query, D an RDF Model and $\nu : V \rightarrow (U \cup B \cup L)$ a valuation function that assigns an element (a URI, a literal, or a blank node) to each variable, then the* set of answers *for query q over RDF model D is defined as*

$$answer(q, D) = \{\nu(H) | \nu(B) \textit{is an instance of } \boldsymbol{D}\}$$

For technical reasons, we define that $\nu(x) = x$ for $x \in (U \cup B \cup L)$

This definition of a set of answers to a query captures the basic functionality of existing RDF query languages. In particular, the notion of an RDF pattern is the direct counterpart of path expressions that occur in various forms in different languages. Features that go beyond this model are either language specific (distinction between explicit and implicit statements) or represent higher level functionality that is based on this model (union, intersection, etc.). It is straightforward to add these higher level constructs into our model.

2.2 Abstract Evaluation Plan

The general structure of an RDF query in terms of a pair of RDF patterns that might share variables allows us to consider the translation problem on a rather abstract level. In particular, we can create an abstract evaluation plan based on the structure of the RDF query that uses abstract operations that can be specified independent of a target language. In the following, we discuss such abstract operators as well as a generic translation from an RDF query given in terms of two patterns.

As we have seen in the previous section, the notion of an answer to an RDF query is based on three models and two mappings between these models. The mapping function links the body pattern with the RDF model by comparing URIs and literals. The valuation function links the body and the head of the query by assigning values to variables. The following diagrams illustrates the general situation.

$$D \overset{\mu,\nu}{\leftrightarrows} B \overset{\nu}{\rightarrow} H$$

If we take an operational view on this situation, we can identify a number of abstract operations that have to be carried out in order to compute answers in this scheme. We chose to define these operators to be as closely as possible linked to the RDF data model in order to support a straightforward translation.

Triple Selection The first kind of operator is very similar to the traditional selection operator from the relational algebra. It establishes the mapping μ between a single triple pattern in the query body and the RDF model. A triple selection operation is specified in terms of a triple pattern $(s, p, o) \in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup L \cup V)$. The input

[1] note that our notion of answer is equivalent to the notion of a pre-answer in [GHM04]

to a selection operation is a set of valuations. The result is the subset of the valuations for which the triple pattern is an instance of the RDF model the selection applies to.

$$\text{select}[s,p,o,D](\{\nu_1,\cdots,\nu_n\}) \quad = \quad \{\nu_i|\mu((\nu_i(s),\nu_i(p),\nu_i(o))) \in D\} \tag{1}$$

Note that in this case, D can be any RDF Model. This allows us to model the case where we have more than one source file. The values s, p, and o can be either URIs literals, blank nodes, or variables. This leaves us with eight different kinds of selection operations ranging from the operation that selects the entire model (all three elements of the triple are variables) to an operation that returns a single statement or the empty set if the specified triple is not in the model (none of the three elements of the triple is a variable).

Triple Join The selection operator only works on the level of individual triples. In order to match a complete pattern against the model, we have to be able to join triple sets that have been selected. Due to the fixed arity of relational statements in RDF, the concept of a join is quite different from the one in the relational model. Rather than defining the join in terms of the triples involved, we define the join over two variables V_1 and V_2 as a selection over valuation functions that satisfy the requirement that the same value has to be assigned to the same value to the two variables.

$$\text{join}[V_1,V_2](\{\nu_1,\cdots,\nu_m\}) \quad = \quad \{\nu_i|\nu_i(V_1) = \nu_i(V_2)\} \tag{2}$$

Notice that selections and joins are fully composable. The definition of a join operation in terms of a selection of valuations allows us to apply the join operation to the results of several selection operations. These selections return sets of possible valuations that are just restricted with respect to the variables that occur in the triple pattern that has been used in the selection.

Triple Construction The advantage of using valuations as the basis for defining the abstract operations is that there is a straightforward way of defining operators that construct the result set. As we have seen above, the structure of the result set is defined in terms of a set of triple patterns that have to be instantiated using a set of valuation functions. As the other operators are used to determine the right set of valuation functions, the construction of the results with respect to a single triple pattern can be done by just applying all valuation functions to the pattern resulting in a set of triples that are part of the result.

$$\text{construct}[s,p,o](\{\nu_1,\cdots,\nu_k\}) \quad = \quad \{(\nu_i(s),\nu_i(p),\nu_i(o))\} \tag{3}$$

Using the triple construction operator, the complete result set can be created by applying construction operators for each triple pattern in the query head to the same set of valuation functions. If this set of valuation functions has been created by a suitable combination of selection and join operations, they already implement the requirements specified in the query body leading to a correct result set.

Plan Generation Given an RDF Query $q = (H \leftarrow B)$ over an RDF model D, we can now generate an abstract evaluation plan based on the operators defined above. The idea of the plan generation is the following:

1. We start with the set of all possible assignments of variables to URIs, literals and blank nodes in the given model (denoted as $\mathfrak{V}_D$)
2. For every triple pattern $(s, p, o) \in B$, we apply a corresponding selection function

$$\text{select}[s, p, o, D](\mathfrak{V}_D), (s, p, o) \in B$$

3. For all shared variables V_i in B, we apply a join operation on the union of the selection results:

$$\text{join}[V_i, V_i](\bigcup_{(s,p,o) \in B} \text{select}[s, p, o, D](\mathfrak{V}_D)), V_i \text{ shared in } B$$

4. For each triple pattern $(s', p', o') \in H$ we apply a construction operator on the intersection of the results of the join operations:

$$\text{construct}[s', p', o'](\bigcap_{V_i \text{ shared in } B} \text{join}[V_i, V_i](\bigcup_{(s,p,o) \in B} \text{select}[s, p, o, D](\mathfrak{V}_D))$$

5. The result set of the query is the union of the results of the construct operators. The corresponding abstract evaluation plan for an RDF query has the following form:

$$\bigcup_{(s',p',o') \in H} \text{construct}[s', p', o'](\bigcap_{V_i \text{ shared in } B} \text{join}[V_i, V_i](\bigcup_{(s,p,o) \in B} \text{select}[s, p, o, D](\mathfrak{V}_D)))$$

This abstract evaluation plan is the basis for generating a concrete evaluation plan in a specific target language. In the following section we show how the abstract plan can be implemented using XQuery expressions that directly work on a normalized XML representation of an RDF model.

3 XQuery-based Implementation of Operators

The ability to match trees or even graphs on XML documents and to construct arbitrarily structured XML qualifies XQuery as a natural target language for our translation. In this section, we develop the XQuery queries needed to answer general queries on RDF data stored in its XML syntax. We assume that the XML syntax of the RDF data is stored in a normalized form, i.e.

- all statements about a resource are contained in a single Description element.
- there is no nesting of Descriptions.
- blank Nodes are identified by node identifiers and specified using description elements.
- types of resources are described using type statements.
- all deducible information is explicitly contained in the descriptions, i.e. we work on the intensional database.

As an advantage of establishing the normal form, we need not handle the wealth of synonymous serializations of RDF in XML, see [CS04] for a discussion. The serialization, we have characterized above is used, e.g. by Jena, as RDF/XML serialization format. A similar approach was taken before us [Rob01, PSS03].

3.1 Implementing Triple Patterns

First, we show how to retrieve the complete (extensional) database from the document. Then, we extend the resulting query to support variable bindings in triple patterns. An immediate extension will be support for chains of triple patterns. Finally, we show how to handle complex head conditions using constructors in XQuery. Notice that we employ the `unordered` ordering mode to signal the XQuery processor document order is not relevant when we query RDF data. As a consequence, the XQuery processor can compute the XQuery expressions much faster.

Triple Selection First, we show how we implement triple selection as defined in Eqn. 1. We start with a basic query that retrieves the extensional database, i.e. the triple pattern in the body of the query only contains variables. The XQuery statement below computes all triples of subjects, predicates and objects, i.e. $\mathfrak{V}_D$ mentioned in Section 2.2. For each subject, all contained predicates are obtained. Our normalizes RDF representation assures that all information about a subject is contained in its descendants. Finally, the object is bound to variable obj. The **if** statement handles objects that either refer to a resource or are literals. We discuss function `local:printNormalized` at the end of this section. It constructs the result in normalized RDF/XML format.

```
unordered {
for $subj in $doc//*[@rdf:about or @rdf:ID],
    $pred in $subj/descendant::*
let $obj := if($pred/@rdf:resource | $pred/@rdf:ID)
            then ($pred/@rdf:resource | $pred/@rdf:ID)
            else $pred/text()
return
  local:printNormalized($subj, $pred, $obj)
}
```

When we are only interested in triples that match some condition, we add a filter to the **where** clause of the XQuery statement. In the predicates below, we use abstract conditions for the subject, predicate, or object, e.g. ”S” as condition for the subject. Notice that the conditional in the query fragment above already distinguishes between literals and references to resource. These conditions replace a variable in the triple pattern by a valid value as defined in Definition 1.

```
(: subject bound :)
some $s in $subj satisfies fn:contains(fn:string($s), "S")

(: predicate bound :)
some $p in $pred satisfies fn:contains(fn:name($p),"P")

(: object bound :)
some $o in $obj satisfies fn:contains(fn:string($o), "O")
```

With the conjunction of these conditions, we can construct all 8 possible instances to filter triple patterns.

Triple Join To implement the triple join as defined in Eqn. 2, we retrieve two triples from the RDF model and add join predicates to the query. Notice that we can join triples that originate from different XML documents. For example, we use the following query for the triples {`S P1 O1`} {`S P2 O2`}:

```
unordered {
for $subj1 in $doc//*[rdf:about],
    $pred1 in $subj1/descendant::*
let $obj1 := if($pred1/@rdf:resource)
             then $pred1/@rdf:resource
             else $pred1/text()
for $subj2 in $doc//*[rdf:about],
    $pred2 in $subj2/descendant::*
let $obj2 := if($pred2/@rdf:resource)
             then $pred2/@rdf:resource
             else $pred2/text()
where
    $subj1 = $subj2 and $obj1 != $obj2
return
   local:printNormalized($subj1,$pred1,$obj1)
}
```

Triple Construction Notice that triple construction, defined in Eqn 3, is the final operator of an evaluation plan. Thus, we assume that every result triple is computed before calling the result construction function. Every invocation of this function generates exactly one result triple. Thus, we can turn the constructed result into our normalized representation by grouping triples by their subject and adding all contained predicates and objects as children. The constructed result obeys to our RDF normal form. Consequently, query results may again serve as inputs to other queries.

```
declare function local:printNormalized($subj as node(),
                                       $pred as node(),
                                       $obj  as node()) as node() {
  element rdf:Description {
     attribute rdf:about {$subj},
     element { fn:name($pred/self::node()) } {
        $obj
     }
  }
};
```

The basic building blocks triple selection and triple join allow us to query the extentional database. It is not difficult to generalize the query pattern of the triple join to compute transitive closures. Therefore, we construct a recursive XQuery function that given all found triples and the triple detected in the last recursive call, detects new triples until a fix point is reached [BR86]. The ability to compute transitive closures allows us to infer information from given the extensional data.

4 Performance Evaluation

We now study the feasability of our approach. Therefore, we combine XML metadata stored both in RDF or in XML. We also compare the performance of querying the XML text files using XQuery with querying RDF data stored and preprocessed in an RDF reasoning system.

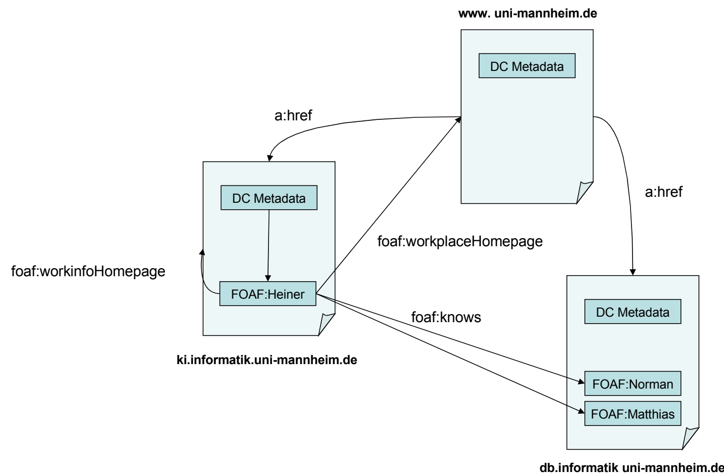

Figure 1: Embedded RDF Metadata

4.1 Analyzing Blogs

In our scenario, shown in Figure 1, we are interested in blogs on the Web, e.g. what are the hottest topics currently discussed, who contributes to the content of some blog, looking for keywords in the text of the blogs. The answers to these queries might be interesting to a company that wants to measure the success of its PR campaign.

Among the different formats for storing blogs, RSS is the most popular. RSS is an RDF vocabularly that is based on Dublin Core. Hence, we can load RSS feeds into an RDF store and query it using an RDF query language. At the same time, the syntax of RSS is XML, and it is natural to analyze it with XQuery. Typical blog entries contain information about the author, the title of the blog entry and some description. Blog entries may reference external content, e.g. via XLink or links in XHTML, but technically every RSS feed is self-contained.

However, references to external content are beyond the scope of an RDF reasoner. If we want to analyze social networks, we might be interested in the relationships of some author. Therefore, we might explore friend-of-a-friend (FOAF) relationships. This analysis may yield interesting background information, e.g. the academic track of some discussant might help to interpret his arguments.

Consider Figure 1, where a blog contains references to us via our FOAF identifier. You might be interested, who else we know. Given our flexible translation scheme, it is possible to combine the queries developed in Section 3 with application specific queries to analyze our FOAF data to find out.

4.2 Experimental Setup

We have downloaded eleven blogs randomly from the Web. The overall size of the experimental data is 800KB resulting in an RDF model of 6206 triples.

In our experiments, we use Galax 0.7.2 [Gal07] as the test plattform for XQuery. Galax loads all data into a main-memory representation and performs all query evaluation in main memory. Its implementation targets completeness of the XQuery standard rather than efficiency.

The other competitor is Jena 2.5.2 [Jen07]. Jena is a Java API for processing RDF including functions to construct, browse or even query RDF models. RDF models can either be accessed via a main-memory representation but they can also be stored in relational databases, e.g. MySQL, PostgreSQL, or Derby. Queries can be expressed in the RDF query language SPARQL. If a persistent data store is used, Jena can push predicates into the relational engine. However, in our experiments, we restrict ourselfs to the main-memory representation of the RDF model.

The experiments were carried out on a simple 2.4 GHz Pentium PC running SuSE Linux 10.2 with 1GB RAM. Jena was executed with Sun Java 2 version 1.5 with up to 512MB heap size.

4.3 Performance Results

We have executed example queries for every basic query pattern, identified in Section 2.2. Table 1 contains a summary of our results; below we discuss them. We report the average execution times of ten invocations of every query.

RDF Loading Time A first interesting question is the time to generate the main-memory representation used during query processing because both evaluators perform all evaluation in main memory. Galax parses every XML document accessed in a query before the query is evaluated. Thus, we can measure the parsing time when we measure the time to evaluate a query that evaluates to an empty result without traversing the document. Since the Jena API allows us to trigger this operation explicitly, we can isolate the loading time of Jena. Both systems also support persistent data store and, thus, the effort for parsing can be reduced.

Our experimental results show that Galax needs 3.6 seconds to load the XML data into main memory while Jena only needs 2.6 seconds. It is likely that Galax needs longer because the XQuery data model includes much more information than then the triple store of RDF, e.g. structural relationships between XML nodes.

Queries We now discuss the performance of processing queries that implement the basic patterns, we have identified.

Query Q1 selects a specific subject, in our case a certain blog entry. Thus, this query is a very selective one because only 101 out of 6205 triples exist for the desired subject. Galax performs a complete traversal over the whole document requiring 10.5 seconds to

Query	Description	result size	XQuery	Jena
Loading		6206	0.6s	2.6s
Q1	select a single subject	101	10.0s	3.1s
Q2	select all triples with a given predicate	2810	2.0s	3.8s
Q3	search for content in an object	24	2.1s	3.3s
Q4	join triples	244	443s	3.4s

Table 1: Performance Summary

evaluate this query. Jena is much faster and answers this query in 3.2 seconds.

Query Q2 finds all triples with predicate ”title”. The selectivity of this query is approx. $1/3$. Evidently, Galax is faster than Jena in returning rather large fragments of the RDF model .

Query Q3 performs a selection on objects. Particularly, it finds all triples that contain the string literal ”Web Service”. This string pattern match is more expensive to evaluate than the remaining selections. For our data it returns only 24 result triples. Again, Galax returns the query result faster than Jena.

Query Q4 computes the join over two triples. In a first experiment, we tried to return all pairs of triples that refer to the same subject but to distinct objects. However, Galax did not finish this query within 45 minutes, and Jena ran out of memory.

Thus, we restricted the query to titles (dc:title) that contain the text “Memory Leak”. Now, Jena clearly outperforms Galax. While the execution time of Jena only needs 3.4s, Galax needs more than 100 times longer.

5 Discussion

In this paper, we motivate the need for an integrated approach to process RDF data. Users want to combine metadata represented in RDF with other data stored in XML [Bat04, Bat06]. Thus, we propse an algorithm that translates RDF queries into XQuery. We present the relevant building blocks to implement selections on single triples, joins over serveral triples, and construction of new triples. Any query language that supports these three primitives can be the used to evaluate queries over RDF models. We argue for a mapping into XQuery because in this case querying can be performed directly on RDF serialized as XML. As a result, it now becomes possible to query accross diverse data sources that store RDF, or any other XML data. Our experiments show that the resulting queries can be evaluated with similar performance as RDF data. This is a very encouraging result because we have not considered query optimization or physical optimizations yet that may even speed up this integrated approach to query diverse XML data sources.

By mapping of RDF data model into the XQuery data model, we follow previous work that attempted to answer queries over RDF data with XQuery [Rob01, PSS03]. However, in their studies, they were not concerned with performance. Our contribution includes a first

performance assessment of the resulting queries. Our results indicate that we can expect similar query performance for typical RDF queries when we answer them in XQuery. Others tried to discover users intents by defining a canonical interpretation of XML data in terms of RDF [Mel99a, Mel99b, Haw01].

The XQuery patterns we propose work on an normalized XML representation. Evidently, our normal form is to some extent an arbitrary choice as there are many different ways to serialize RDF graphs as XML. We refer to [CS04] for a discussion on this topic. Thus, when RDF graphs are represented in a different format, our query patterns need to be adjusted. Nevertheless, our processing model still remains functional, and thus we assume that the various XML representations of RDF can be transformed into our normal form. Moreover, the fundamental performance characteristics we have identified in this paper will not change.

References

[Bat04] S. Battle. Round-tripping between XML and RDF. In *3rd Int. Semantic Web Conference*, 2004.

[Bat06] S. Battle. Gloze: XML to RDF and back again. In *1st JENA User Conference*, 2006.

[BD06] D. Bulterman and M. DeMeglio. The SMIL 3.0 Metainformation Module. W3C Recommendation, 2006.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. *SIGMOD Rec.*, 15(2):16–52, 1986.

[CS04] J. J. Carroll and P. Stickler. RDF Triples in XML. In *Extreme Markup Languages*, 2004.

[DvHF+00] S. Decker, F. van Harmelen, D. Fensel, M. Klein, J. Broekstra, M. Erdmann, and I. Horrocks. The Semantic Web: The Roles of XML and RDF. *IEEE Internet Computing*, 4(5):63–74, 2000.

[Gal07] Galax XQuery implementation. http://www.galaxquery.org/, 2007.

[GHM04] C. Gutierrez, C. Hurtado, and A. O. Mendelzon. Foundations of Semantic Web databases. In *ACM PODS*, pages 95–106, 2004.

[Haw01] S. Hawke. XML with Relational Semantics: Bridging the Gap to RDF and the Semantic Web. Technical report, W3C, 2001.

[Jen07] Jena: A Semantic Web Framework for Java. http://jena.sourceforge.net/, 2007.

[Mel99a] S. Melnik. Bridging the Gap between RDF and XML. Technical report, Stanford University, 1999. http://infolab.stanford.edu/ melnik/rdf/fusion.html.

[Mel99b] S. Melnik. Simplified Syntax for RDF. Technical report, Stanford University, 1999. http://www-db.stanford.edu/ melnik/rdf/syntax.html.

[PSS03] P. Patel-Schneider and J. Simeon. The Yin/Yang Web: A Unified Model for XML Syntax and RDF Semantics. *IEEE Trans. on Knowledge and Data Engineering*, 15(3), 2003.

[Rob01] J. Robie. The Syntactic Web: Syntax and Semantics on the Web. In *XML Conference*, 2001.

[WK03] U. Westermann and W. Klas. An analysis of XML database solutions for the management of MPEG-7 media descriptions. *ACM Comput. Surv.*, 35(4):331–373, 2003.