Improve Efficiency of Mapping Data between XML and RDF with XSPARQL

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

[1] Digital Enterprise Research Institute, NUI Galway* `firstname.lastname@deri.org`
[2] Siemens AG Österreich, Siemensstrasse 90, 1210 Vienna, Austria

Abstract XSPARQL is a language to transform data between the tree-based XML format and the graph-based RDF format. XML is a widely adopted data exchange format which brings its own query language XQuery along. RDF is the standard data format of the Semantic Web with SPARQL being the corresponding query language. XSPARQL combines XQuery and SPARQL to a unified query language which provides a more intuitive and maintainable way to translate data between the two data formats. A naive implementation of XSPARQL can be inefficient when evaluating nested queries. However, such queries occur often in practice when dealing with XML data. We present and compare several approaches to optimise nested queries. By implementing these optimisations we improve efficiency up to two orders of magnitude in a practical evaluation.

Keywords: RDF, XML, SPARQL, XQuery, XSPARQL

1   Introduction

The Extensible Markup Language (XML) [2] is a widely adopted data format for exchanging data over the World Wide Web. To query XML data, the W3C recommends using XQuery [3]–a functional and strongly typed query language. XQuery features FLWOR expressions which consist of a list of `ForClause`s, comparable to `for` loops of imperative languages, and `LetClause`s, to assign values to variables. The `WhereClause` can be used for filtering items and the `OrderByClause` for ordering. The final `ReturnClause` contains the "body" of the loop and determines the format of the return values of the resulting sequence.

The Resource Description Framework (RDF) [4] is the data model used for Semantic Web data. The query language for RDF is SPARQL [5]–also a W3C Recommendation, with a syntax similar to SQL. The main part of a SPARQL query is the graph pattern which specifies the desired part of an RDF graph.

XSPARQL [1] is an integrated language to transform data between XML and RDF formats providing a more intuitive and maintainable solution than an ad-hoc setup using multiple scripts and queries in several query languages. XSPARQL is agnostic of concrete data serialisation syntaxes and processes data on the data
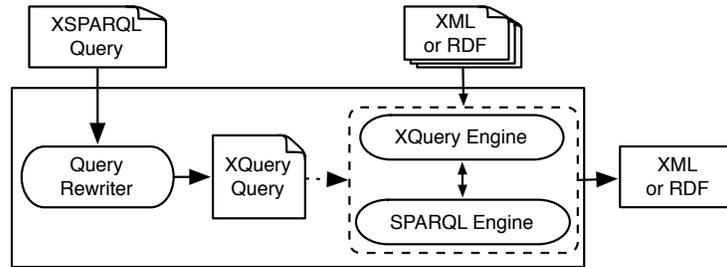
Figure 1. XSPARQL implementation architecture [1]

model level, which is tree-based for XML and graph-based for RDF. XSPARQL is built by unifying XQuery and SPARQL. Syntactically, and semantically, XSPARQL is built on top of XQuery by introducing a new kind of `ForClause` called `SparqlForClause`, which is syntactically similar to the SPARQL `SELECT` query. By this extension XSPARQL allows one to select data from RDF graphs using the convenient graph pattern syntax.

2 Implementation

Figure 1 shows the general architecture of our implementation. Queries are evaluated in the two steps Rewriting and Evaluation: First the query is rewritten to an XQuery query containing parts of SPARQL queries. In the second step the rewritten query is evaluated by an XQuery engine calling a SPARQL engine for the embedded graph patterns. These two engines process both XML and RDF data and eventually produce either XML or RDF.

The two engines used are probably highly optimised. Thus one source of inefficiency is the interface between the two engines. Stressing this interface, i.e., evaluating a high number of SPARQL graph patterns, would therefore lead to inefficient query evaluation times.

Claim. XSPARQL queries yielding a high number of SPARQL graph pattern evaluations are a source of inefficiency in a naive implementation.

3 The Problem: Evaluating Nested Graph Patterns

As stated in the last section, query evaluation can be very slow in some cases for a given naive implementation. A query containing a `SparqlForClause`, also called *inner loop*, nested in another `ForClause`, called *outer loop*, is said to perform a *join* if outer and inner loops share any variables. Especially evaluation of join queries is ineffective if the implementation takes no additional measures of optimising query evaluation.

Queries consisting of such a nested structure are common for non-trivial transformations of RDF to XML. This follows from the structure of the target

Listing 1. Query 9: For each person list the number of items bought in Europe [1]

```
1  prefix : <http :// xsparql . deri . org / data / >
2  prefix foaf: <http :// xmlns . com / foaf /0.1/ >
3  for $id $name from < data . rdf >
4  where { [] foaf : name $name ; :id $id . }
5  return
6    < person name = "{$name}" > {
7      for * from < data . rdf >
8      where {
9         $ca :buyer [:id $id] .
10        optional { $ca :itemRef $itemRef .
11          $itemRef :locatedIn [ :name "europe" ] .
12          $itemRef :name $itemname } .  }
13      return <item >{ $itemname } </ item >
14   } </ person >
```

XML format where nesting and grouping of objects (elements) are natural building blocks for which XSPARQL must cater.

When evaluating a join query, the SPARQL engine will be called N times, N being the number of iterations of the outer loop. If the outer loop is also a `SparqlForClause` the XQuery engine will call the SPARQL engine once more.

Example 1. In the query in List. 1 (query 9 from the XMark benchmark suite [6], adapted to XSPARQL) first the outer loop iterates over persons (starting on line 3) while the nested (inner) loop extracts all items bought in Europe by each person (lines 7–12). The outer and the inner loops are `SparqlForClause`s. For an example dataset containing 1000 persons, the XQuery engine would call the SPARQL engine 1001 times.

One might try to simplify the query by using one single `SparqlForClause` only. Although possible, one has to take care to not unintentionally change the semantics of the query. Especially ordering and grouping, which are solved elegantly in XQuery, would need special attention.

As outlined in Sect. 2 the interface between the XQuery and SPARQL engines is crucial when thinking about query evaluation performance. Evaluating queries containing nested `SparqlForClause`s yields a high number SPARQL engine calls.

Claim. Nested graph patterns, i.e., nested `SparqlForClause`s, yield the evaluation of a high number of SPARQL graph patterns therefore such queries are evaluated inefficiently in a naive implementation.

Additionally we assume that the evaluation of a single `SparqlForClause` results in a performance penalty by itself because the SPARQL engine must parse the query and generate a query plan every time when used. For nested `SparqlForClause`s the SPARQL engine has to parse, plan, optimise, and evaluate several queries which only differ in few variable values.

4 The Solution: Proposed Optimisation

We aim to improve performance by minimising the number of SPARQL calls to reduce the impact of repeated parsing, planning, optimising and evaluating of similar queries. This includes queries containing nested `SparqlForClause`s.

We differentiate between optimisations which perform the join via XQuery or via SPARQL during query evaluation.

4.1 XQuery Optimisations

The idea is to rewrite the inner loop to perform only one single SPARQL call instead of N SPARQL calls.

Nested Loop Join (NL). The nested loop join is achieved by issuing first an unconstrained SPARQL call and then iterating over the join candidate sequences in XQuery in a nested loop. We implemented this specific approach twice, once with joining in an XQuery `WhereClause` (NL-W) and one joining in an XPath expression in the XQuery `ForClause` (NL-X).

Sort-Merge Join (SM). The sort-merge join is implemented similarly. But instead of iterating over the join candidate sequences in a nested loop, the actual join is performed as a standard sort-merge join. The two join candidate sequences are first ordered and then joined by a tail-recursive merge function.

4.2 SPARQL Optimisations

The idea is to push the join to the SPARQL engine and thus reducing the number of SPARQL calls.

Merge Graph Patterns (SR). If both, the inner and the outer loop are featuring graph patterns, then both graph patterns can be merged into one and executed at once on the SPARQL engine.

Inject Named Graph (NG). If only the inner loop contains a graph pattern then the join candidate sequence can be encoded in RDF and inserted in a triple store. Next the whole join is executed at once on the SPARQL engine, similar to the Merge Graph Patterns optimisation.

5 Practical Evaluation and Results

For the practical evaluation we used the XQuery benchmark suite XMark [6]. We adapted queries and documents/datasets to support our use case of transforming data from RDF to XML. The naive rewriting and the optimised rewritings were tested using datasets with sizes ranging from 1 MB to 100 MB (timeout after ten hours). For the experiments we used Saxon 9.3 as XQuery engine and ARQ 2.8.7 as SPARQL engine.

The number of iterations of the outer loop, i.e., saved SPARQL calls, is directly related with the dataset size. Thus we expect the evaluation runtimes for the optimised queries to increase slower with the dataset size.
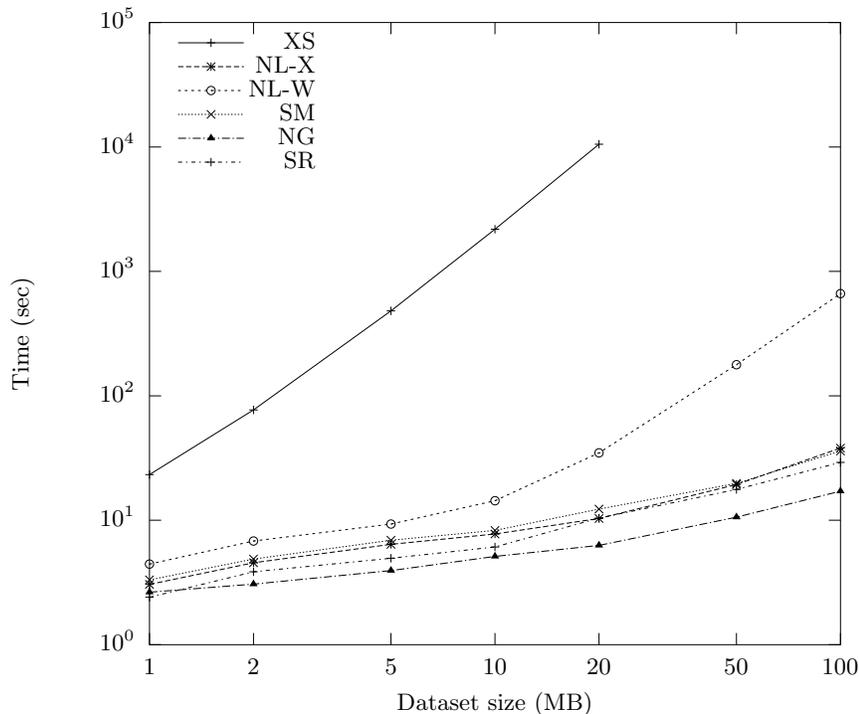
Figure 2. Evaluation times of query 9 [1]

We found query evaluation runtimes for 6 out of the 20 benchmark queries being very high. 5 out of those where queries containing nested `SparqlForClauses`. Our optimisation approaches were applicable for 3 out of these 5 queries (XMark queries 8, 9, and 10). In the following we describe and discuss the results for query 9 only. The results of queries 8 and 10 are comparable and support our conclusions.[3]

As an example of the results, Fig. 2 shows the query evaluation times of XMark query 9 (see List. 1) of the naive implementation together with the 5 different optimisation methods. The naive XSPARQL (XS) evaluation times increase polynomially with the dataset size. Although the NL implementations are similar, the XPath variant (NL-X) of the nested loop join was evaluated much faster than the variant joining in the `WhereClause` (NL-W). The differences between NL-X and the Sort-Merge join (SM) seem negligible; we assume that Saxon optimises nested loops similarly to our Sort-Merge join implementation. When applicable the SPARQL optimisations (NG and SR) show an even better

---

[3] For the concrete rewritings of the different optimisation approaches for the different queries and evaluation results refer to the recently published Technical Report [1].

performance than the XQuery optimisations. One possible explanation for the difference between NG and SR could be the time the SPARQL engine needs to inject the named graph into the RDF store.

All tested optimisations have a bigger performance gain the bigger the dataset is. Thus the performance gain is directly related to the number of SPARQL calls of the unoptimised query, i.e., the number of saved SPARQL calls.

6 Conclusions and Future Work

Query Evaluation Efficiency. Performance of XSPARQL is drastically reduced when evaluating queries containing nested `SparqlForClauses`.

Performance Improvement. Performance of XSPARQL queries containing such nested `SparqlForClauses` can be improved by different kinds of optimisations. This performance improvement increase with dataset size.

XSPARQL Usage. While the XSPARQL language can provide a more intuitive and maintainable solution to transforming data between RDF and XML, an XSPARQL engine can also provide a better performance for such tasks than ad-hoc setups.

In the future we will concentrate on finding new optimisation approaches in a more systematic way, by isolating fragments of the XSPARQL language which are easier to evaluate.

Furthermore we aim at broadening the scope of XSPARQL by allowing access to relational data and the increasingly popular JSON format. We also plan to provide measures to update data within XML or RDF databases.

References

1. Bischof, S., Krennwallner, T., Lopes, N., Polleres, A.: Mapping between RDF and XML with XSPARQL. Tech. rep., DERI Galway (April 2011), available at `http://www.deri.ie/fileadmin/documents/DERI-TR-2011-04-04.pdf`
2. Bray, T., Paoli, J., Sperberg-Mcqueen, C.M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation, W3C (Nov 2008), available at `http://www.w3.org/TR/2008/REC-xml-20081126/`
3. Chamberlin, D., Robie, J., Boag, S., Fernández, M.F., Siméon, J., Florescu, D.: XQuery 1.0: An XML Query Language (Second Edition). W3C Recommendation, W3C (Dec 2010), available at `http://www.w3.org/TR/2010/REC-xquery-20101214/`
4. Manola, F., Miller, E.: RDF Primer. W3C Recommendation, W3C (Feb 2004), available at `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`
5. Prud'hommeaux, E., (eds.), A.S.: SPARQL Query Language for RDF. W3C Recommendation, W3C (Jan 2008), available at `http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/`
6. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A Benchmark for XML Data Management. In: Proceedings of the 28th international conference on Very Large Data Bases. pp. 974–985 (2002)