Translating SPARQL and SQL to XQuery

Peter M. Fischer
ETH Zurich
<peter.fischer@inf.ethz.ch>

Dana Florescu
Oracle Corporation
<dana.florescu@oracle.com>

Martin Kaufmann
ETH Zurich
<martin.kaufmann@inf.ethz.ch>

Donald Kossmann
ETH Zurich
<donald.kossmann@inf.ethz.ch>

Abstract

In our community there are three main models for representing and processing data: Relations, XML and RDF. Each of these models has its ”sweet spot” for applications and its own query language; very few implementations cater for more than one of these. We describe a uniform platform which provides interfaces for different query languages to retrieve and modify the same information or combine it with other data sources. This paper presents methods for completely and correctly translating SQL and SPARQL into XQuery since XQuery provides the most expressive foundation. Early results with our current prototype show that the translation from SPARQL to XQuery already achieves very competitive performance, whereas there is still a significant performance gap compared to SQL.

Keywords: SQL, SPARQL, XQuery, Common Runtime

1. Introduction

1.1. Background

Today, three common approaches of representing data in structured and formal form are being used: Relational (tables), XML (trees) and RDF (graphs). While relational and XML data have already been applied widely for a long period, RDF is
now gaining popularity, among others in the contexts of semantic web or social networks. These three approaches not only differ in terms of their data models but also at the level of data representation, use cases, query languages. As a consequence, implementations rarely cover more than one model [13]. Yet, there is a need to overcome this separation and to integrate data and operations. One possible solution would be a common runtime for all of these formats where each language can be exploited where it is suited best.

1.2. Problem Statement

In order to overcome the differences between the models, we investigate if and how one language can be translated into another. In this paper, we focus on a translation from both SPARQL and SQL to XQuery which has seen little attention so far. XQuery is an interesting target since it is the most expressive language and its implementations are now reaching maturity. The translation is required to express the semantics correctly, to cover all expressions and to create code that can be executed efficiently.

1.3. Contributions

In this paper, we present the following results:

• a complete and correct translation of SPARQL to XQuery which does not require any assumptions on the schema of the data or the particular workload
• a sketch of a translation of SQL92 to XQuery, again, with no assumption on schema or workload
• a working cross compiler which takes any SPARQL or SQL92 query and turns it into an XQuery expression
• initial performance results which show that, even with limited optimizations, XQuery is typically as fast as native SPARQL and often faster. In contrast, it still trails SQL implementations due to the simpler and more mature relational storage.

1.4. Outline

This paper is organized as follows: Section 2 gives a short introduction to SPARQL, XQuery and SQL, establishes a running example and outlines the challenges of the translation. A detailed description of the translation from SPARQL to XQuery is shown in Section 3, a summary of the translation of SQL to XQuery in Section 4. Section 5 describes the implementation of the translator and presents some initial correctness and performance results. Section 6 presents related work. The paper is concluded in Section 7 by a summary and directions for future work.
2. Fundamentals

2.1. RDF

The Resource Description Framework (RDF) is W3C Recommendation [12] for the specification of metadata models for the Semantic Web. An RDF document expresses information about a resource on the web by a list of subject-predicate-object triples which correspond to a directed graph. There are different formats for the serialization of RDF data. The most common format is RDF/XML [3] which stores RDF data with an XML syntax. Other formats like Notation 3, N-Triples and Turtle are more suitable for human readability. All formats are equivalent semantically and can be converted into one another easily. Example 1 shows a simple RDF document:

```
Example 1. RDF/XML Example: Periodic table and composite elements
<rdf:RDF>
  <Element rdf:ID="H"><name>hydrogen</name><number>1</number>
  </Element> ...
  <Gas rdf:ID="H2"><element rdf:resource="#H"/><weight>2</weight>
  </Gas> ...
</rdf:RDF>
```

2.2. The SPARQL Query Language

SPARQL [16] is often referred to as the query language for RDF. The basic operation is graph pattern matching, in particular triple patterns in which subject, predicate and/or object may be variables. These patterns can be combined using the operators AND, UNION, OPT and FILTER yielding "solution sequences" (actually unordered bags) which then can be changed by solution modifiers such as DISTINCT, ORDER BY, REDUCED, LIMIT, and OFFSET. SPARQL defines four query forms: SELECT, ASK, CONSTRUCT and DESCRIBE. Example 2 shows a SELECT query which retrieves the color of all elements ending in "ium" and returns the 4th to 14th color after ordering.

```
Example 2. SPARQL example query
PREFIX chemistry: <http://www.xql2xquery.org/chemistry#>
SELECT ?col
WHERE
{ ?element chemistry:name ?name.
}
FILTER (REGEX(?name, "ium")
ORDER BY ?col
```

Translating SPARQL and SQL to XQuery
2.3. XQuery

XQuery is a declarative and Turing complete programming language which was originally designed to extract information and perform transformations on XML data. It uses XDM as its data model which expresses sequences of atomic values or XML nodes. Support for graph structures is limited as there is no standard way to define links across the tree hierarchy and no expressive operations on these links exist.

2.4. SQL

SQL is the most popular language as an interface to a relational database management system (DBMS) and has been extended to suit many additional purposes. It provides expressions for data definition (DDL), data manipulation (DML), access privileges (DCL) and transaction control (TCL). Given the complexity of the language, we only consider the SQL92 DML subset in this work.

2.5. Challenges and Opportunities

At the level of the data model, the differences between the relational, tree/sequence and graph models are already attenuated by the serializations, in particular the RDF/XML mapping. Similarly, type system differences are resolved by SQL/XML mapping which is described in [18] and the shared XML Schema/XPath 2.0 type system (SPARQL/XQuery). Both SQL and SPARQL use three-valued Boolean logic, explicitly addressing null values and errors, respectively. In contrast to this, XQuery uses two-valued Boolean logic, does not represent null values and will only provide error handling in the next version (3.0). The process of graph pattern matching in SPARQL is quite different from the path navigation-style interaction. For this reason, emulation is required which is less concise and possibly less efficient.

3. Mapping and Translating SPARQL to XQuery

In his section, we provide a description of the translation of SPARQL to XQuery. For space reasons, we only show the general idea and the most relevant parts of the translation. The full set of rules is available at [10]. We define a function sparl2xquery() which takes a SPARQL query as an argument and returns the corresponding XQuery representation as a result. The following translation tables show the SPARQL code INSPA in the left column and the corresponding XQuery
3.1. Basic Graph Pattern Matching

Matching triple patterns is the core operation of SPARQL out of which more complex graph combinations can be built. These patterns, in turn, can be filtered and modified. Triple patterns such as (element chemistry:name ?name) contain specifications for subject, predicate and object of an RDF triple. These specifications can be either constants (to match) or variables (which are bound). Our translation maps these variables to XQuery variables and generates a sequence of bindings for all variables. In the result, every element contains a single value for each of the variables without eliminating duplicates (see Example 3). While this is very similar to the "tuple" stream in an XQuery 3.0 FLWOR expression, we explicitly materialize these bindings which enables this intermediate result to be used as an argument to general functions.

Example 3. Variable Binding Sequence

```xml
<result>
  <var name="color">silvery</var>
  <var name="name">aluminium</var>
</result>
<result>
  <var name="color">metallic white</var>
  <var name="name">uranium</var>
</result>
```

A basic graph pattern contains a set of triple patterns which all need to hold and can be joined/correlated by using the same variable. In our translation, one triple can yield up to three nested for iterations since one loop is generated for each different variable. Given the subject-predicate-object nesting in RDF/XML, we start by retrieving subjects using a path expression and bind the variables specified on subjects. Nested into an iteration over all subjects, we retrieve the predicates below them, bind the variables and again iterate over these variables, if necessary. Objects are handled in a similar nested fashion. In general, constants, correlations and other direct filter conditions are expressed as part of the where clause since all possible combinations are generated by the loops. Whenever possible, we push these predicates "down" in order to minimize the size of the intermediate result. Wherever necessary, access to named graphs (as external resources) is mapped to doc() or collection() calls.
<table>
<thead>
<tr>
<th>pattern\textsubscript{SPA} :=</th>
<th>pattern\textsubscript{XQu} :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>triplePattern\textsubscript{SPA} ...</td>
<td>foreach subjectName (subjVars(pattern\textsubscript{SPA}))</td>
</tr>
<tr>
<td></td>
<td>for $subjectName in xqllib:getSubj()</td>
</tr>
<tr>
<td></td>
<td>foreach predicateName (predVars(pattern\textsubscript{SPA}))</td>
</tr>
<tr>
<td></td>
<td>for $predicateName in xqllib:getPred($subjectName)</td>
</tr>
<tr>
<td></td>
<td>foreach objectName (objVars(pattern\textsubscript{SPA}))</td>
</tr>
<tr>
<td></td>
<td>for $objectName in xqllib:getObj($predicateName)</td>
</tr>
<tr>
<td>(where</td>
<td>(where</td>
</tr>
<tr>
<td></td>
<td>foreach constant (constants(pattern\textsubscript{SPA},</td>
</tr>
<tr>
<td></td>
<td>subjectName, predicateName, objectName)))</td>
</tr>
<tr>
<td></td>
<td>$subjectName = constant</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>foreach filterCondition (filter\textsubscript{XQu})</td>
</tr>
<tr>
<td></td>
<td>(and)? filterCondition</td>
</tr>
<tr>
<td></td>
<td>)? return</td>
</tr>
<tr>
<td></td>
<td>&lt;result&gt;</td>
</tr>
<tr>
<td></td>
<td>foreach varName (vars(pattern\textsubscript{SPA}))</td>
</tr>
<tr>
<td></td>
<td>&lt;varName&gt;{data($varName)}&lt;/varName&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/result&gt;</td>
</tr>
</tbody>
</table>

3.2. Graph Pattern Combination

3.2.1. Optional Patterns

The purpose of an optional pattern is to supplement the solution with additional information. If the pattern within an \texttt{OPTIONAL} clause matches, the variables defined by that pattern are bound to one or many solutions. If the pattern does not match, the solution remains unchanged. The optional pattern is implemented in XQuery by a binary function which implements a "left outer join" over the intermediate graph representations. Since the \texttt{OPTIONAL} keyword is left-associative, the rule can be applied repeatedly to handle multiple consecutive optional patterns.

3.2.2. Alternative Graph Pattern

In an alternative graph pattern two possible patterns are evaluated and the union of both is taken as a result. The alternative pattern can be expressed in XQuery by a sequence of the results of both patterns since \texttt{UNION} does not specify duplicate elimination.
3.2.3. Group Graph Pattern

All graphs associated in a group graph pattern must match. The pattern is implemented by an XQuery function that correlates the groups on shared variables using a join and the other function capturing equality in SPARQL.

The intermediate results generated by SPARQL patterns are combined by means of custom XQuery functions. The mapping is shown in the following table:

Table 2. Translation of graph pattern combinations from SPARQL to XQuery

<table>
<thead>
<tr>
<th>patternSPA :=</th>
<th>patternXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ patternLSPA } OPTIONAL { patternRSPA }</td>
<td>xqllib:optional(patternLXQu , patternRXQu)</td>
</tr>
<tr>
<td>{ patternLSPA } UNION { patternRSPA }</td>
<td>(patternLXQu , patternRXQu)</td>
</tr>
<tr>
<td>{ patternLSPA } { patternRSPA }</td>
<td>xqllib:and(patternLXQu , patternRXQu)</td>
</tr>
</tbody>
</table>

3.3. Filter

A SPARQL FILTER function can be added to graph patterns in order to restrict the result according to a Boolean condition. In the running example, elements whose name end in "ium" are filtered according to a regular expression. Since comparison operations, effective Boolean value and several other functions are actually defined by the related XPath 2.0 functions and operators, we can use XQuery value comparison (eq, neq, ...). Yet, we need to consider the differences in Boolean logic and error handling: SPARQL does not allow empty sequences as parameters and suppresses errors in certain logical operations (e.g. TRUE OR Error becomes TRUE). We use the empty sequence (e.g., generated by OPTIONAL expressions) as a placeholder for error and put additional checking code to capture wrong input values. For AND, OR, NOT and effective Boolean value we create helper functions that interpret (e.g.) correctly or catch the error on XQuery 3.0, respectively.

3.4. Modifiers

SPARQL solution modifiers either affect the (unordered) result sequence by imposing an order, projecting variables or limiting the number of results. In any case, there is a straightforward translation to XQuery. Given the intermediate variable binding sequence, projection is expressed in the final return clause by only showing referenced variables. The SPARQL ORDER BY maps directly to the order by in an XQuery
FLWOR expression, both working on sequences. Result size LIMIT and OFFSET are handled by placing positional predicates on the result sequence, e.g. [position() lt 11] for LIMIT 10. DISTINCT is pushed into the query plan affecting operators on patterns as well as the (custom) XQuery functions implementing SPARQL identity semantics. REDUCED is currently translated into a NO-OP since dropping some duplicates is only an optimization.

3.5. Query Forms

SPARQL supports four query forms, SELECT, ASK, CONSTRUCT and DESCRIBE. We show the translation of SELECT here since it is the most common form. The result of the SELECT form is a list of value combinations which are bound in varListSPA.

**Table 3. Translation of SELECT query from SPARQL to XQuery**

<table>
<thead>
<tr>
<th>resultSPA :=</th>
<th>patternXQu :=</th>
</tr>
</thead>
<tbody>
<tr>
<td>nsListSPA</td>
<td>nsListXQu</td>
</tr>
<tr>
<td>SELECT varListSPA WHERE { patternSPA }</td>
<td>let $result := patternXQu</td>
</tr>
<tr>
<td>( ORDER BY orderListSPA )?</td>
<td>( order by orderListXQu )?</td>
</tr>
<tr>
<td>( limitOffsetSPA ) ?</td>
<td>return $result([positionXQu])?</td>
</tr>
</tbody>
</table>

3.6. Translation of the Running Example

In Example 5 we show the result of the automatic translation of the SPARQL example to XQuery as introduced in Section 2.2.

First, the namespaces required for the query are declared and the data of the involved collections is assigned to variables which are named according to $GRAPH_x with $x \in \mathbb{N}_0$. These placeholders represent the different intermediate results.

The first variable $GRAPH_0$ contains the result of a basic graph pattern as described in Section 3.1. A for loop is generated for each variable because the SPARQL semantics adds one result for each possible binding of values to variables. We start with the bindings for the subject parts (?element). In a next step, we look at the predicate and object steps for each subject. This can be done efficiently because RDF/XML nests the predicates and objects into the subjects. Since there are two variables for the objects (?name and ?col), we create two additional loops. For each possible binding the value of each variable must be non-empty. Therefore, we add an explicit check `fn:exists()` to a where clause in XQuery.

The intermediate result represented by $GRAPH_0$ is limited by a FILTER expression as shown in Section 3.3 and the reduced set of possible bindings is assigned to $GRAPH_1$. In the next step, the result is sorted by the attribute "col". Finally, the
output is generated by the function `formatSparqlXML()` which renders the result according to the SPARQL Query Results XML Format which is defined in [19].

To make the code generated by the cross-compiler more concise, a number of custom XQuery functions are used. As an example, the source code of the function `getSubj()` from the xqllib package is shown in Example 4. This function returns the identifier of a given node which can be obtained by reading the `rdf:ID` attribute. The source code of the remaining functions can be found in [10].

**Example 4. Source Code of the Custom Function getSubj()**

```xquery
declare function xqllib:getSubj ($subj as node())
    as xs:string
{
    return $subj/@rdf:ID
}
```

**Example 5. Translation Result of the SPARQL (Code 2) Example to XQuery**

```xquery
declare namespace chemistry =
    "http://www.xql2xquery.org/chemistry";
let $doc_chemistry := fn:collection("chemistry")
let $result :=
    let $GRAPH_0 :=
        for $element in $doc_chemistry[@rdf:ID]
            let $value_element :=
                xqllib:getSubj($element)
            for $value_col in
                xqllib:getData("chemistry",$element/chemistry:color)
            for $value_name in
                xqllib:getData("chemistry",$element/chemistry:name)
            where fn:exists($value_col) and fn:exists($element)
            and fn:exists($value_name)
        return
            <result>
                <var name="col">{$value_col}</var>
                <var name="element">{$value_element}</var>
                <var name="name">{$value_name}</var>
            </result>
    let $GRAPH_1 :=
        $GRAPH_0[fn:matches(var[@name="name"], "ium")]
    for $node in $GRAPH_1
        order by $node/var[@name="col"]
    return $node
return
xqllib:formatSparqlXml(
```
4. Mapping and Translating SQL to XQuery

Given that XQuery was designed to also handle relational data (see Use Case R described in [20]) and that the expressions can be fully nested, the translation of selection, projection, (inner) joins, ordering, sub queries as well as updates is straightforward. The translation only requires an adaptation of predicates and path expressions to the concrete serialization of relational data into XML. For group by and outer join we consider both explicit nested for loops and the specialized constructs for XQuery 3.0. Null values are mapped to an empty sequence in evaluation and empty elements in the results since the Boolean evaluation rules are a close match. Yet, many differences in the data model and semantics make a fully correct translation rather complex. An example for a translation rule is shown in Table 4 by means of an SQL SELECT statement.

Table 4. Translation of SELECT query from SQL to XQuery

<table>
<thead>
<tr>
<th>SQL</th>
<th>XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT exprList_SQL FROM source1_SQL, ..., sourceN_SQL ( WHERE boolExpr_SQL )? ( GROUP BY groupBy_SQL )? ( ORDER BY orderList_SQL )?</td>
<td>for $source1 := source1_XQu ... for $sourceN := sourceN_XQu ( group by groupBy_XQu )? ( where boolExpr_XQu )? ( order by orderList_XQu )? return nodes_XQu</td>
</tr>
</tbody>
</table>

5. Implementation and Evaluation

5.1. Implementation

We have implemented our formal translation framework as a cross-compiler from SPARQL and SQL to XQuery which is available as a command-line tool and as a web service [11]. The code is written in C++ and follows the classical compiler architecture: Lexer (Flex), Parser (Bison), Semantic Checks and Code Generation towards textual XQuery which can be consumed by any XQuery engine.

The current focus is on complete and correct translation. Thus, only a limited amount of optimizations are present – mostly pushdown of constants and filter predicates within basic pattern matches. As the results show, these optimizations are already useful, but for more complex queries more effort to minimize intermediate results is necessary.
5.2. Evaluation

We evaluated our translation with regard to correctness, completeness and performance. Although at this stage we have no formal verification of our translation, tests on a range of sample queries covering all language features turned out to be correct. The translated XQuery tests were run on the XQBench Benchmarking Service [7], putting native SPARQL execution on ARQ/Jena 2.8.7 [2] and SQL on MySQL 5.1.41 against a number of open-source XQuery processors and databases, capturing execution results and timings: Saxon HE 9.3.0-2, Qizx Open 4.1, XQilla 2.2.4, Zorba 1.5 pre-release (processors), BaseX 6.5, MonetDB/Pathfinder October 2010, Sedna 3.4, BerkeleyDB XML 2.5.16 and eXist 1.4.0 (databases). To the best of our knowledge, this is a fairly complete collection of freely available XQuery implementations. For an initial performance study, we chose the Berlin SPARQL Benchmark [5] which is one of the few existing benchmarks suites for SPARQL. It provides queries and a data generator for both SPARQL and SQL on the same conceptual schema. The tests have been executed for Berlin scaling factors from 10 (~ 5K triples) to 5000 (~ 1.8M triples) on an Intel Xeon X3360 quad-core 2.8 ghz with 8 GiB RAM and 2x 750 GB S-ATA disks. In our measurements we excluded start-up/warm-up time by deducing the time of a dummy query and repeating the measurements until they became stable. Furthermore, we kept the setup of all engines to the out-of-the-box settings, and did not any additional optimizations like custom indexes. The translation from both SPARQL and SQL to XQuery took around 10-25 msec, while the execution times varied quite significantly for different queries and scale factors as shown in Figures 1-8.

We are omitting a number of results in graphs, which are available on the XQBench Benchmarking Service [7]:

- We were not able to gather any result from MonetDB. After rewriting the queries to conform to mandatory static typing, we ran into non-implemented functions or accessors. While a full rewrite for this functionality might have been possible, the effort would have been significant.
- XQilla performed very similar to Zorba, sometimes slightly worse, we therefore omitted the results to make the graphs more readable.
- BerkeleyDB and eXist performed similar or slower than the other XML databases. We again omitted the graphs to improve readability.

For a simple triple match query (Figure 1), all XQuery engines outperform ARQ significantly and scale better indicating that the translation of triple and basic graph pattern matching is quite efficient. For the XQuery processors, the total cost is completely dominated by XML parsing, thus making Zorba slower than Saxon. The XQuery databases do not need to pay the price for parsing, and only need to deal with a bigger data size. As a result, they maintain a lead of 3 orders of magnitude over ARQ even without the support of manually added indexes. When comparing
the semantically same query translated from SQL and comparing the results against MySQL, the quality of the XQuery execution does not change much. MySQL’s query processing and storage engines are well tuned for such simple retrieval queries, but sees a strong competition from the XML databases, with Sedna catching up at scale factor 5000.

![Figure 1. BERLIN SPARQL Query 1](image1)

![Figure 2. BERLIN SQL Query 1](image2)

For a query with an `OPTIONAL` clause (Figure 3) the XQuery translation from SPARQL requires a join and duplicate elimination which is currently implemented using nested loop. We measured a significant difference in the quality of the XQuery op-
timizers: Saxon and BaseX seem to exploit more optimization possibilities than the other XQuery implementation and scale in the same way as ARQ, but maintain a lead of more than one order of magnitude. The other XQuery systems scale somewhat worse, but still maintain a measureable lead at scale 5000. For the translation from SQL (Figure 4), the simpler query structure gives all optimizers the chance to detect the relevant information, leading to almost the same performance as for the simple retrieval query before.

Figure 3. BERLIN SPARQL Query 3

Figure 4. BERLIN SQL Query 3
For a query that uses a simple filter (Figure 6), the results for both the SPARQL and the SQL translation closely resemble the result for the triple pattern specification in Q1: All optimizers detect enough relevant information to let the XQuery implementations scale better than ARQ, while parsing becomes the dominant cost. MySQL leads against the XML databases, but the gap is closing quickly.

For a query with a join and multiple `OPTIONAL` statements (Figure 7) our approach is beginning to show its limitations for the SPARQL translation. Expressing each `OPTIONAL` expression as a nested loop with additional helper functions limits the
ability of the XQuery optimizers to detect relevant information, so that the XQuery implementations are now roughly as fast as ARQ. Again, Saxon and QizX scale best. On the SQL side, the query is expressed in somewhat simpler terms, so that `OPTIONAL` is just another selection. The results of the XQuery implementation are generally better, but also diverge more: Zorba and Sedna do not seem to detect any join optimizations and scale fairly bad. Saxon, QizX and BaseX seem to detect the join and scale well when the document size increases (with the former two mostly dominated by parsing). MySQL, however, scales even better, not being affected by the increasing workload. This can be attributed to the very low selectivity of the query and the presence of predicates which can use the implicitly created primary key index.

![Figure 7. BERLIN SPARQL Query 8](image-url)

**Figure 7. BERLIN SPARQL Query 8**
As a conclusion, the results indicate that our translation is working quite well for most cases, outperforming the SPARQL reference implementation for most of the queries and scaling similarly or better than the SQL results, albeit at a higher initial cost. Certain join queries still seem to be problems in both translations, necessitating further investigation and more translation hints.

6. Related Work

A significant number of translations from SPARQL to SQL have been performed which achieve both completeness and efficiency as summarized in [15]. Even though SPARQL and XQuery are significantly closer than SPARQL and SQL, we are only aware of two works that have tackled such a translation. In [4] the authors aim to query XML sources from SPARQL without having to use XQuery explicitly. In order to do so, they require the presence of XML schema which is mapped to an equivalent OWL schema for RDF. SPARQL queries written against this OWL schema are then translated into XQuery expressions on the corresponding XML schema. The translation is incomplete and, based on the publicly available information, it is not possible to verify its correctness. [8] embeds SPARQL into XQuery and provides a translation to XQuery without assuming schema knowledge. Again, this translation is incomplete and their evaluation shows significantly worse performance with ARQ clearly outperforming the XQuery engines. Although there are several approaches to translate SQL to XQuery, all of them suffer from quite severe restrictions. One approach and an overview of the competing methods are given in [14]. For the opposite direction, the Pathfinder Relational XQuery Processor [9] has achieved a very high degree of correctness and performance.
7. Conclusion and Future Work

In this work, we tackled the problem of aligning SPARQL, SQL and XQuery semantics and execution by providing a translation from SPARQL and SQL to XQuery. The translation is complete, correct and universally usable as it does not need any schema or specific workloads. An initial performance evaluation shows that in areas on which we already performed optimizations the XQuery translation beats the native SPARQL implementations whereas in other areas it still lags behind. When comparing to SQL, storage seems to be the relevant difference. For both translations, current XQuery optimizers seem to exploit many optimizations when queries are reasonably simple, but fail once a certain level of complexity is exceeded.

We see the following avenues for future work: Our translation should be further validated, both by formal verification and testing against the official DAWG SPARQL test suite. We aim to incorporate additional optimizations in order to reduce intermediate results for pattern combinations and duplicate elimination; among them are explicit and implicit join reordering, incorporating XQuery 3.0 features like native group by, outer joins and error handling as well as index usage. Furthermore, we plan to investigate the upcoming SPARQL 1.1 language which provides updates, grouping and aggregation. All of these features should be easy to express in XQuery.

Bibliography


[16] Eric Prud’hommeaux et al. SPARQL Query Language for RDF.


