TriQuery: Modifying XQuery for RDF and Relational Data

David Bednářek, Jiří Dokulil
Charles University in Prague, Czech Republic
{bednarek, dokulil}@ksi.mff.cuni.cz

Abstract

The ability to convert between different data formats is important in large and heterogeneous information systems. Although XML was established as an universal standard for data exchange, XML-related languages like XQuery lack the ability to access data in other formats; in particular, relational data and RDF. In this paper, we describe TriQuery – an extension of the XQuery language which adds records (tuplex) and RDF-specific operators. Using the statically optimizable record types, relational data as well as the results from RDF sub-queries can be integrated more efficiently than with their traditional encoding using XML elements and attributes.

1 Introduction

Large information systems, service-oriented architecture, and similar environments often integrate components based on different concepts of data representation [10] – in this paper, we will address the relational model, the XML, and the RDF. The choice of model affects not only the physical representation but also the style how the physical reality is modeled. Thus, the output of a generic format converter is often unnatural with respect to the style of the target model [1]. In other words, the conversion of format shall be coupled with the corresponding change of style; therefore, man-made converters are required.

Implementing such a converter requires a programming language capable of reading data in one model and writing data in another one. In addition, the language shall be strong enough to allow the change of style (for instance, flattening XML trees to relational tables). On the other hand, the language shall allow global optimization of the data access at least on the input side. Ideally, the converter shall behave as a view which can be integrated to a query at its output side [3].

Surprisingly, there is no widely accepted language covering all the above requirements [7]. Although general procedural languages offer input/output libraries for all data models, there is no global data-access optimization above the level of individual embedded query statements; moreover, the presentation of a procedural program as a view is purely utopian. On the other hand, query languages like SQL and XQuery offer optimization and view integration. However, they significantly favor their particular data models. Other languages that might theoretically fulfill all the requirements, like Datalog or functional languages, are too exotic for industrial application. Consequently, we have to extend an existing and sufficiently widespread language – given the requirements for computing strength and some degree of global optimization, there only few candidates like XDuce [6], XQuery [2] or XSLT.

We decided to choose XQuery for three reasons. First, it is more widespread than XDuce and better readable than XSLT. Second, the type system used by RDF is most similar to the one used by XQuery. Last but not least, we are already have experience with XQuery evaluation. Extending XQuery to handle RDF has already been tried [9] – this approach uses user-defined functions, but the queries are poorly readable.

In this paper, we describe TriQuery – an extension of the XQuery language which adds records (tuplex) and a pattern matching operator. Although tuples could be easily modeled using XML attributes in XQuery, such approach lacks static type-checking which is considered a must in the relational model. Moreover, management and navigation of dynamically created XML data is a difficult task, imposing unnecessary burden on the evaluation engine when only relational tuples are required. For these reasons, we decided to add the record type and the corresponding operators, employing their advantage of being weaker than trees and statically type-checked.

With records, XQuery can be easily extended to handle the three most prominent data models used today – relational data, the XML, and the RDF. Besides records, we only need a pattern matching operation (to support graph matching in the RDF) and several functions (similar to the well known fn:doc used for accessing XML documents).

The following section describes the way in which the records were added to XQuery, the Section 3 describes the
RDF support that TriQuery provides, while the Section 4
does the same for relational data. The Section 5 deals with
implementation issues. The last section concludes the pa-
per.

2 Records in XQuery

First, let us stress out some of the properties of XQuery,
that are essential for understanding of the TriQuery defini-
tion. The basic building block of XQuery is an expression,
which may be constructed from keywords, symbols, and
operands. XQuery allows expressions to be nested with full
generality. An expression evaluates to an ordered sequence
of items. An item is either an atomic value (string literal,
numeric literal, etc.) or a node reference – a reference to a
node in an XML tree.

We extend the XQuery language to handle records – a
structured piece of data in the form “identifier - value”.
Each record has a set of identifiers (field names) and a value
corresponds to each of these identifiers (field value). The
set of identifiers either consists of qualified names to form
a named record or a sequence of numbers 1..n to form an
anonymous record. The field values are sequences of items.
The sequences are allowed to easily accommodate multiple
valued attributes that may be present in RDF data.

Sequences of records can be used in TriQuery as values
of expressions. This of course requires us to extend most of
the XQuery constructs to handle such sequences. These def-
initions are presented in the following section, along with
eamples of their use.

The following text references several concepts defined
by the XQuery or related standards:

(basic) item – an atomic value or a node reference,
Val – the set of all items and sequences of such items [4],
QName – the set of all qualified names.

2.1 Definitions and examples

Our first extension of XQuery is the addition of records
to create an extended item. The extended items are used
instead of the basic items in all situations, except for the
situations where we explicitly specify otherwise. This is a
significant modification of the language and its semantics,
but it greatly simplifies the inclusion of RDF querying into
XQuery and it also creates new possibilities even when RDF
is not used.

(anonymous/named) record, signature A record is a
partial function from QName ∪ N to Val. Note that
this does not allow records as field values (nested records).
There are three limitations on the domain D of the function:
it must be finite, it must contain either numbers or qualified
names (but not both) and if it contains numbers, it must be
a sequence starting with number one (D = \{1..n\}). If the
domain consists of numbers, we call the record anonymous,
if it consists of qualified names, it is named.

An extended item is a basic item or a record. If any se-
quence contains a record, we require the sequence to con-
tain nothing but records and also that all of the records have
the same domain D. This domain is called the signature
of the sequence. TriQuery enforces the rule, that during
statical analysis of the query, it has to be possible to de-
termine whether any expression may return a sequence of
records and if it does, identify the signature of the sequence.
All definitions are created in a way that allows such statical
analysis.

record constructors A record can be created by a record
constructor. It is a language construct similar to the XML
constructors already present in XQuery. There are two ver-
sions – for anonymous and named records.

\{8, "Hello, world!", $x\} creates an anonymous
record where 1 is mapped to the integer 8, 2 to the string
"Hello, world!" and 3 to the value of the variable $x. The in-
dividual expressions may be exactly the same as in the case
of function call, except that they may not contain records –
this is checked statically.

\{eight := 8, hello := "Hello, world!",
var := $x\} creates a record where the qualified name
“eight” is mapped to the integer 8, etc. One field name can
be used at most once. The constraint on the expression is
the same as with anonymous records.

In anonymous record constructor, an item may be re-
placed by $x.* which is a short for $x.1, ..., $x.n,
where n is the number of fields in $x. The named record
constructor allows the form *:= $x.* which is expanded to
$x.f_1, ..., \$x.f_n$ where f_i are the fields of the record
in $x$. The requirement, that one field name cannot be used
more than once, must still be maintained, i.e. the expanded
form must fulfill the requirement.

In both cases, the $x$ is used only to shorten the defini-
tion; it may be any expression that evaluates to a record.
This is only a “syntactic sugar”, but it can significantly
shorten the textual representation of some queries.

concatenation of sequences The sequence concatenation
– the comma operator – is extended naturally to handle
records. However, if any of the sequences contains records,
then both sequences must have the same signature. Accord-
ing to this definition, the following query is illegal:

\[ \{ a:=1, b:=2 \}, \{ a:=1, c:=3 \} \]

The two concatenated records do not have the same sig-
nature: \{a, b\} \neq \{a, c\}. On the other hand, the following
query is valid since \{a, b\} = \{b, a\}:

\[ \{ a:=1, b:=2 \}, \{ b:=1, a:=3 \} \]
Similar rules are also enforced for other operators like
if ... then ... else ....

**dot – field access operator** Fields of a record can be accessed using the dot operator in a way that is similar to that of object oriented programming languages.

If \( r \) is a record and \( x \) a member of QName\( \cup \)N such that \( x \in \text{dom}(r) \), then \( r.x \) evaluates to \( r(x) \). For example, \$a.2 \) evaluates to \( r(2) \) if the value of \$a \) is the anonymous record \( r \). The situation is analogous for named records – \$a.hello \) evaluates to \( f(\text{hello}) \).

If \( A \) is a sequence \( (a_1, ..., a_n) \) such that \( \forall i \leq n : \text{dom}(a_i) = D \) (i.e., \( D \) is the signature of the sequence) and \( x \) identifies a field (i.e., \( x \in D \)), then \( A.x \) evaluates to a sequence \( (a_1(x), ..., a_n(x)) \). Note that \( a_i(x) \) may be a sequence in which case it is concatenated with other values to form one “flat” sequence.

The dot operator can be used to correct the example that demonstrated incompatibility of records and atomic values. The following query is valid:

```xml
let $x:=[ 1 ], $y:='2
return ($x.1,$y)
```

**record equality** All operators that test for equality require the operands to be compatible, i.e., if any of them contains records, they must have the same signature, otherwise the query is invalid. The operator \( = \) tests all individual fields like this: \( r = s \iff \forall f \in \text{dom}(r) : r(f) = s(f) \). Note that since nested records are not allowed, \( r(f) = s(f) \) never test records, thus the original XQuery operator is used.

To be consistent with the way XQuery handles sequence equality, we have to use an existential quantifier in the definition – two sequences are equal, if the first one contains an item equal to an item contained in the other sequence.

We do not allow the operator \( \neq \) to be used on records, since the extension of the original operator from XQuery would be very unnatural and counterintuitive.

**node and value comparators** The operators \( << \) and \( >> \) are not allowed for records, the operator \( = \) is extended for record sequences \( A \) and \( B \) with the following conditions:

- length of \( A \) and \( B \) is at most one
- \( A \) and \( B \) have the same signature
- all fields of all records in \( A \) and \( B \) contain sequences with length at most one

If either \( A \) or \( B \) is empty sequence or any field of any record in \( A \) or \( B \) is an empty sequence, then the result is an empty sequence. Otherwise, \( A \) contains one record \( a \) and \( B \) contains one record \( b \). Then the result is true if \( \forall f \in \text{dom}(a) : a(f) \) \( \neq b(f) \). Otherwise, the result is false.

The definitions of \( \text{eq} \) and \( \text{ne} \) are the same as the definition of \( = \), except that the condition for \( \text{eq} \) is \( \forall f \in \text{dom}(a) : a(f) \) \( \text{eq} b(f) \). For \( \text{ne} \), the condition is \( \exists f \in \text{dom}(a) : a(f) \) \( \neq b(f) \).

**FLWOR** The FLWOR operators are naturally extended to handle sequences of records. Let us give a few examples where FLOWR is used to process records. The signature \{ a, b \} of the sequence bound to \$x \) can be modified to \{ \( x, y \) \} like this:

```xml
for $r in $x return [ x:=$r.a, y:=$r.b ]
```

Even an anonymous record may be created:

```xml
for $r in $x return [ $r.a, $r.b ]
```

The following example demonstrates how a sequence of records can be created from XML data:

```xml
let $x={
    for $n in fn:doc("ex.xml")//item
        return [nr:=$n/number, name:=$n/name]
}
```

The reverse would be:

```xml
for $r in $x return
    <item><number>{$r.nr}</number>
    <name>{$r.name}</name></item>
```

### 2.2 Pattern matching operation

An important operation on records is a selection of a specific pattern from a large collection of records. The user specifies a set of records where some values are replaced by variables. The system then tries to find all possible substitutions of the variables that transform the pattern to a subset of the input data. In other words, it tries to find all instances of such pattern in the input data. The result of the operation are the applicable variable bindings.

Although this pattern matching could be handled by the FLWOR operator, we decided to include a new operation that simplifies the usage of such selection and also adds new functionality. The basic form of the operator is either \$x \) match \{ \text{pattern} \} or \$x \) with \text{uri} \) match \{ \text{pattern} \}. The \text{pattern} \) is a sequence of tuples that may contain variables. Note that the variables may be written as either \$var \) or \?var \) for better “visual compatibility” with other languages.

Let \$x \) be a variable that contains a record sequence with signature \{1, 2, 3\}. Then an example of pattern matching operation may look like this:

```xml
$x \) match \{ "John" (1+2) \?a \ . \?a \?b \?c \}
```

This expression searches the sequence of records (the value of \$x \) for records that look like \{ "John", 3, a \} and for each such record tries to find a record \{ a, b, c \} \) where \( a \) is the same value on both cases, while \( b \) and \( c \) may be any value. Then, it returns a sequence of records \{ a:= a, b:= b, c:= c \}. A more complete example could look like this:
for $r$ in $x$ match
{ "John" {1+2} ?a . ?a ?b ?c }
return <r><a>{$r.a}</a><b>{$r.b}</b></r>

There are some limitations on the pattern matching operation. Let us assume it is in the form of $D$ match \{ $P$ \} or $D$ with $T$ match \{ $P$ \}. Then the following must conditions must be met:

- The expression $D$ must evaluate to a sequence of anonymous records.
- All patterns in $P$ must have the number of elements equal to the cardinality of the signature of $D$.
- Each XQuery expression used in any pattern must return a sequence of basic items with at most one item.
- Acceptable values of $T$ are implementation-defined.

The operation then returns a sequence of named records, whose signature is equal to the set of all variable names used in $P$. In general, if the $T$ argument is present, the system may perform any operation on the input data and pattern and produce any result, as long as it produces a sequence of records with the right signature. However, in situations where $T$ is missing or if the default behavior is sufficient, the “basic” definition is used.

In the “basic” mode, the pattern matching operation tries to find all variable mappings that transform the pattern to a subset of the input data and returns the mappings in the form of a sequence of (named) records.

### 3 RDF support

With records, inclusion of RDF querying is much easier. The whole RDF dataset can be represented as a sequence of anonymous records with three fields. The pattern matching operation can be used in a way that is similar to a basic graph pattern in SPARQL. The optional $with$ modifier can be used to specify entailment regimes, for instance RDFS entailment. Since we have a constructor for new anonymous records, we can also create new RDF datasets.

A direct advantage of TriQuery is that it can perform transformations between XML and RDF – data from records (RDF triples) can be extracted using the dot operator and used in XML constructors. And, of course, the other direction can be done using record constructors.

The usual scenario is that the data is extracted from an RDF database, using the appropriate function (e.g., let $sc=triq:doc("data1")$). Then, pattern matching is performed as the very first step:

for $m$ in $sc$ with triq:rdfs match
{ $x$ ex:firstName $fn$ .
  $x$ ex:lastName $ln$ }

After that, the rest of the query is written in a manner similar to the original XQuery, except that records can still be used not only to represent RDF triples and results of pattern matching. A more complete example is a simple extraction of facts about people into XML:

```
let $d := triq:doc("people.rdf")
for $x$ in $d$ match ( $x$ ex:id $id$ .
  $x$ ex:firstName $fn$ . $x$ ex:email $m$ )
return <person id="{$x.id}">
  <name>{$x.name}</name>
  <email>{$x.mail}</email></person>
```

We may also create new RDF data sets, for instance by extracting information from an XML document. Consider the following example:

```
let $d := fn:doc("in.xml")
for $x$ in $d//item
return [ $x/id, ex:name, $x/name],
[ $x/id, ex:price, $x/price]
```

This creates a sequence of anonymous records with three fields – an RDF data set.

The $with$ modifier of the match operator can be used for advanced entailment modes, for instance in the form $d$ with triq:rdfs match (...).

### 4 Relational data support

An interesting consequence of the addition of records to XQuery is the ability to easily extend it to handle data stored in relational databases. All we need is a function, e.g. rel:table, that takes the name of the table and returns a sequence of named records that correspond to the rows of that table. For instance, consider tables `person(id,name)` and `contact(id,mail)`.

Then, we can use the following query:

```
for $p$ in rel:table("person")
for $m$ in rel:table("contact")
where $p.id=$m.id
return <contact name="{$p.name}">
  {$m.mail}</contact>
```

This extension is easy to implement, providing there is a suitable data-store for the relational data. The rest could be either directly reused or adopted from TriQuery. The whole system could then be used to transform data in any direction between relational databases, XML and RDF.

### 5 Implementation

Since XQuery is already Turing-complete, the TriQuery extensions certainly do not provide more power from the...
theoretical point of view [8]. Furthermore, the original intention is that the fields of the records are used to store atomic values, rather than node references. If this is the case, the records have a very straightforward mapping to XML fragments.

For example, a named record \([a:=1, b:=2]\) can be represented by a XML fragment \(<r><a>1</a><b>2</b></r>\). Then \(\$x/a\) would be used instead of \(\$x.a\). There are other differences, for instance the equivalence testing would have to be handled by a function, since the behavior of the equivalence operator is different. But overall the mapping is straightforward. The XML fragment representation even allows the user to mix atomic values and records with different signature within one sequence.

If node references are used as field values, the mapping cannot be done this way, since it is not possible to create references between XML tree fragments. There is a requirement that requires transforming the records into a sequence of XML nodes. However, the upcoming XQuery 1.1 will likely support node references in XML fragments [5], enabling us to use the easier (and more efficient) transformation in all cases.

A more preferred way of implementing TriQuery is to modify an existing XQuery engine to natively handle records. This may be a significant modification, however, the limitations placed of the usage of records greatly simplify it. Especially the requirement that each sequence that contains records contains only records and that all of them have the same layout – the requirement for the existence of a signature.

Since this has to be statically identifiable, the evaluation engine may use a less general but more efficient representation when handling records compared to a situation where the records are transformed into XML fragments. A sequence of node references (XML fragments) is often represented as an array of references to memory where the fragments are stored. A corresponding sequence of records may be stored as a table.

But, in many cases it may even be possible to determine whether or not a certain field of a record may contain more than one value (be a sequence). If no field may contain a sequence, the whole data stream is in fact a table, where columns correspond to fields and rows to records. Such analysis may be performed statically during the translation of the query.

6 Conclusions and future work

This paper proposes a modification of XQuery that extends it to provide a convenient way of handling not only XML files, but also relational databases and RDF collections. The data may be accessed and freely transformed among the three formats. The language may even be used to create views on the data. The limitations imposed on the records provide some degree of static checking of the query and makes creating an efficient evaluation engine significantly easier.

An obvious next step is to create a pilot implementation of an evaluation engine for TriQuery. At the moment, we have a working lexical and syntactical analyzer that shows that the extension of the XQuery grammar is correct and we are working on a referential implementation of the language.

Acknowledgments

This work was supported by the Grant Agency of the Czech Republic, grant number 201/09/0990 (XML Data Processing) and Grant Agency of the Charles University in Prague, grant number 127109 (RDF repository).

References