XQuery

Web Data Management and Distribution

Serge Abiteboul   Ioana Manolescu   Philippe Rigaux
Marie-Christine Rousset   Pierre Senellart

Web Data Management and Distribution
http://webdam.inria.fr/textbook

September 23, 2011

Why XQuery?

XQuery, the XML query language promoted by the W3C. See:

http://www.w3.org/XML/Query

Check your queries online (syntactic analysis):

http://www.w3.org/2005/qt-applets/xqueryApplet.html

Sample queries:

http://www.w3.org/TR/xquery-use-cases/

XQuery vs XSLT

- XSLT is a procedural language, good at transforming XML documents
- XQuery is a declarative language, good at efficiently retrieving some content from large (collections of) documents

Remark

In many cases, XSLT and XQuery can be used interchangeably. The choice is a matter of context and/or taste.

Main principles

The design of XQuery satisfies the following rules:

Closed-form evaluation. XQuery relies on a data model, and each query maps an instance of the model to another instance of the model.

Composition. XQuery relies on expressions which can be composed to form arbitrarily rich queries.

Type awareness. XQuery may associate an XSD schema to query interpretation. But XQuery also operates on schema-free documents.

XPath compatibiliy. XQuery is an extension of XPath 2.0 (thus, any XPath expression is also an XQuery expression).

Static analysis. Type inference, rewriting, optimisation: the goal is to exploit the declarative nature of XQuery for clever evaluation.

At a syntactic level, XQuery aims at remaining both concise and simple.

A simple model for document collections

A value is a sequence of 0 to *n* items.

An item is either a node or an atomic value.

There exist 7 kinds of nodes:

- **Document**, the document root;
- **Element**, named, mark the structure of the document;
- **Attributes**, named and valued, associated to an **Element**;
- **Text**, unnamed and valued;
- **Comment**;
- **ProcessingInstruction**;
- **Namespace**.

The model is quite general: everything is a sequence of items. This covers anything from a single integer value to wide collections of larges XML documents.

Examples of values

The following are example of values

- `47` : a sequence with a single item (atomic value);
- `<a/>` : a sequence with a single item (**Element** node);
- `(1, 2, 3)` : a sequence with 3 atomic values.
- `(47, <a/>, "Hello")` : a sequence with 3 items, each of different kinds.
- `()` the empty sequence;
- an XML document;
- several XML documents (a collection).

Sequences: details

There is no distinction between an item and a sequence of length 1 $\Rightarrow$ everything is a sequence.

Sequence cannot be nested (a sequence never contains another sequence)

The notion of “null value” does not exist in the XQuery model: a value is there, or not.

A sequence may be empty

A sequence may contain heterogeneous items (see previous examples).

Sequences are ordered: two sequences with the same set of items, but ordered differently, are different.

Items: details

Nodes have an identity; values do not.

Element and **Attribute** have type annotations, which may be inferred from the XSD schema (or unknown if the schema is not provided).

Nodes appear in a given order in their document. Attribute order is undefined.

Syntactic aspects of XQuery

XQuery is a case-sensitive language (keywords must be written in lowercase).

XQuery builds queries as composition of expressions.

An expression produces a value, and is side-effect free (no modification of the context, in particular variable values).

XQuery comments can be put anywhere. Syntax:

```
(:This is a comment :)
```

Evaluation context

An expression is always evaluated with respect to a context. It is a slight generalization of XPath and XSLT contexts, and includes:

- Bindings of namespace prefixes with namespaces URIs
- Bindings for variables
- In-scope functions
- A set of available collections and a default collection
- Date and time
- Context (current) node
- Position of the context node in the context sequence
- Size of the sequence

XQuery expressions

An expression takes a value (a sequence of items) and returns a value.

Expressions may take several forms

- path expressions;
- constructors;
- FLWOR expressions;
- list expressions;
- conditions;
- quantified expressions;
- data types expressions;
- functions.

Simple expressions

Values *are* expressions:

Literals: ’Hello’, 47, 4.7, 4.7E+2

Built values: `date(‘2008-03-15’)`, `true()`, `false()`

Variables: `$x`

Built sequences: `(1, (2, 3), (), (4, 5))`, equiv. to `(1, 2, 3, 4, 5)`, equiv. to `1 to 5`.

An XML document is *also* an expression.

```
<employee empid="12345">
<name>John Doe</name>
<job>XML specialist</job>
<deptno>187</deptno>
<salary>125000</salary>
</employee>
```

The result of these expressions is the expression itself!

Retrieving documents and collections

A query takes in general as input one or several sequences of XML documents, called *collections*.

XQuery identifies its input(s) with the following functions:

- *doc*() takes the URI of an XML document and returns a singleton document tree;
- *collection*() takes a URI and returns a sequence.

The result of the *doc*() function is the *root node* of the document tree, and its type is **Document**.

XPath and beyond

Any XPath expression is a query. The following retrieves all the movies titles in the `movies` collection (for movies published in 2005).

```
collection('movies')/movie[year=2005]/title
```

The result is a sequence of `title` nodes:

```
<title>A History of Violence</title>
<title>Match Point</title>
```

Remark

The XPath expression is evaluated for each item (document) in the sequence delivered by `collection('movies')`.

Constructors

XQuery allows the construction of new elements, whose content may freely mix literal tags, literal values, and results of XQuery expressions.

```
<titles>
  {collection('movies')//title}
</titles>
```

Expressions can be used at any level of a query, and a constructor may include many expressions.

Remark

An expression *e* must be surrounded by curly braces {} in order to be recognized and processed.

Constructors

Other element constructors

```
<chapter ref="[{1 to 5, 7, 9}]">
```

same as:

```
<chapter ref="[1 2 3 4 5 7 9]">
```

```
<chapter ref="[1 to 5, 7, 9]">
```

same as

```
 <chapter ref="[1 to 5, 7, 9]">
```

The constructor:

```
<paper>{$myPaper/@id}</paper>
```

will create an element of the form:

```
<paper id="271"></paper>
```

Variables

A variable is a name that refers to a value. It can be used in any expression (including identity) in its scope.

```
<employee empid="{$id}">
  <name>{$name}</name>
{$job}
  <deptno>{$deptno}</deptno>
  <salary>{$SGMLspecialist+100000}</salary>
</employee>
```

Variables `$id`, `$name`, `$job`, `$deptno` and `$SGMLspecialist` must be bound to values.

FLWOR expressions

The most powerful expressions in XQuery. A FLWOR (“flower”) exp.:

- iterates over sequences (**f**or);
- defines and binds variables (**l**et);
- apply predicates (**w**here);
- sort the result (**o**rder);
- construct a result (**r**eturn).

An example (without `let`):

```
for $m in collection(’movies’)/movie
where $m/year >= 2005
return
<film>{$m/title/text()},
     (director: {$m/director/last_name/text()})
</film>
```

FLWOR expressions and XPath

In its simplest form, a FLWR expression provides just an alternative to XPath expressions. For instance:

```
let $year:=1960
for $a in doc('SpiderMan.xml')//actor
where $a/birth_date >= $year
return $a/last_name
```

is equivalent to the XPath expression

```
//actor[birth_date>=1960]/last_name
```

Not all FLWR expressions can be rewritten with XPath.

A complex FLWOR example

"Find the description and average price of each red part that has at least 10 orders" (assume collections *parts.xml* and *orders.xml*):

```
for $p in doc("parts.xml")//part[color = "Red"]
let $o := doc("orders.xml")//order[partno = $p/partno]
where count($o) >= 10
order by count($o) descending
return
<important_red_part>
{ $p/description }
<avg_price> {avg($o/price)} </avg_price>
</important_red_part>
```

for and let

Both clauses bind variables. However:

for successively binds each item from the input sequence.
`for $x in /company/employee` binds each employee to $x, for each item in the *company* sequence.

let binds the whole input sequence.
`let $x := /company/employee` binds $x to all the employees in *company*.

Note the `for` may range over an heterogeneous sequence:

```
for $a in doc("Spider-Man.xml")//*
where $a/birth_date >= 1960
return $a/last_name
```

Here, $a is bound in turn to all the elements of the document! (Does it work? Yes!)

`for` + `return` = an expression!

The combination `for` and `return` defines an expression: `for` defines the input sequence, `return` the output sequence.

- A simple loop:

```
for $i in (1 to 10) return $i
```

- Nested loops:

```
for $i in (1 to 10) return
   for $j in (1 to 2) return $i * $j
```

- Syntactic variant:

```
for $i in (1 to 10),
   $j in (1 to 2) return $i * $j
```

- Combination of loops:

```
for $i in (for $j in (1 to 10) return $j * 2)
  return $i * 3
```

Defining variables with let

let binds a name to a value, i.e., a sequence obtained by any convenient mean, ranging from literals to complex queries:

```
let $m := doc("movies/Spider-Man.xml")/movie
return $m/director/last_name
```

A variable is just a synonym for its value:

```
let $m := doc("movies/Spider-Man.xml")/movie
for $a in $m/actor
return $a/last_name
```

The scope of a variable is that of the FLWR expression where it is defined. Variables cannot be redefined or updated within their scope.

The where clause

where is quite similar to its SQL synonym. The difference lies in the much more flexible structure of XML documents.

“Find the movies directed by M. Allen”

```
for $m in collection("movies")/movie
where $m/director/last_name="Allen"
return $m/title
```

Looks like a SQL query? Yes but predicates are interpreted according to the XPath rules:

1. if a path does not exists, the result is false, no typing error!
2. if a path expression returns several nodes: the result is true if there is at least one match.

“Find movies with Kirsten Dunst” (note: many actors in a movie!)

```
for $m in collection("movies")/movie
where $m/actor/last_name="Dunst"
return $m/title
```

The return clause

return is a mandatory part of a FLWR expression. It is instantiated once for each binding of the variable in the for clause.

```
for $m in collection("movies")/movie
let $d := $m/director
where $m/actor/last_name="Dunst"
return
  <div>
   {$m/title/text(), "directed by",
        $d/first_name/text(),  $d/last_name/text()},
     with
     <ol>
      {for $a in $m/actor
        return <li>{$a/first_name, $a/last_name,
                         " as ",  $a/role}</li>
      }
     </ol>
  </div>
```

Joins

Nested FLWOR expressions makes it easy to express joins on document, à la SQL:

```
for $p in doc("taxpayers.xml")//person
  for $n in doc("neighbors.xml")//neighbor
  where $n/ssn = $p/ssn
  return
<person>
  <ssn> { $p/ssn } </ssn>
            { $n/name }
   <income> { $p/income } </income>
 </person>
```

Remark

The join condition can be expressed either as an XPath predicate in the second **for**, or as a **where** clause.

Join and grouping

“Get the list of departments with more than 10 employees, sorted on the average salary”

```
for $d in doc("depts.xml")//deptno
 let $e := doc("emps.xml")//employee[deptno=$d]
  where count($e) >= 10
  order by avg($e/salary) descending
  return <big-dept>
      { $d,
        <headcount>{count($e)}</headcount>,
         <avgsal>{avg($e/salary)}</avgsal>
       }
    </big-dept>
```

Operations on lists

XQuery proposes operators to manipulate lists:

1. concatenation
2. set operations: (union, intersection, difference)
3. Functions (*remove*(), *index-of*(), *count*(), *avg*(), *min*(), *max*(), etc.)

The distinct *values* from a list can be gathered in another list. (This loses identity and order.)

“Give each publisher with their average book price”

```
for $p in
  distinct-values(doc("bib.xml")//publisher)
  let $a :=
   avg(doc("bib.xml")//book[publisher=$p]/price)
  return
    <publisher>
      <name>{ $p/text() }</name>
      <avgprice>{ $a }</avgprice>
     </publisher>
```

if-then-else expressions

“Give the holding of published documents”

```
for $h in doc("library.xml")//holding
return
 <holding>
    { $h/title,
         if ($h/@type = "Journal")
         then $h/editor
         else $h/author }
</holding>
```

some expressions

some expresses the existential quantifier:

“Get the document that mention sailing and windsurfing activities”

```
for $b in doc("bib.xml")//book
where some $p in $b//paragraph
   satisfies (contains($p,"sailing")
               and contains($p,"windsurfing"))
return $b/title
```

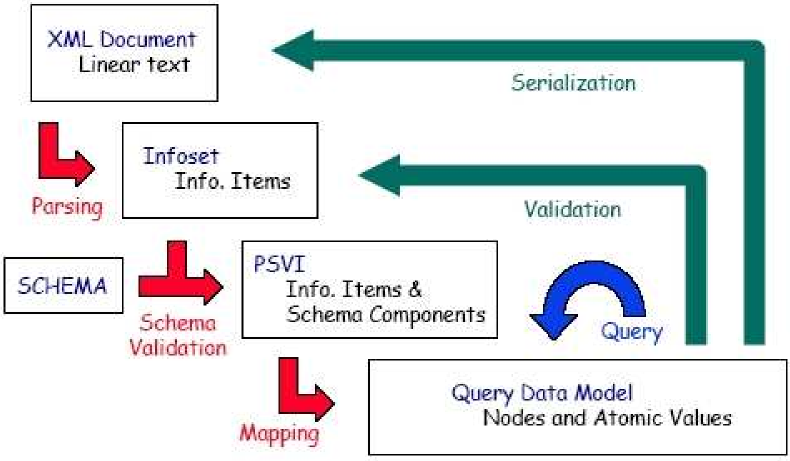
every expressions

every expresses the universal quantifier:

“Get the document where each paragraph talks about sailing“

```
for $b in doc("bib.xml")//book
where every $p in $b//paragraph
             satisfies contains($p,"sailing")
return $b/title
```

XQuery processing model



When XQuery doesn't behave as expected

1. The query does not parse (applet grammar check page) ⇒ reformulate it. You may start from the XQuery use cases.
2. The query parses, but does not work.
3. The query works, but the results are unexpected ⇒ figure out what the parser understood.

When XQuery doesn't behave as expected

Sometimes the query parses but will not work (the engine will refuse it).

The parser only checks that the production is well-formed. It does not check that the context provides sufficient information to run the query:

- the functions called in the query are defined
- the variables referred in the query are defined
- the numeric operations are legal etc.

This query parses but it does not work:

```
for $x in doc("bib.xml")//book
return <res1>{$x/title}</res1>,
       <res2>{$x/author}</res2>

org.exist.xquery.XPathException:
variable $x is not bound.
```

When XQuery doesn’t behave as expected

Sometimes the query parses but will not work (the engine will refuse it).

This query parses but it does not work:

```
for $x in doc("bib.xml")//book
return <res1>{$x/title}</res1>,
       <res2>{$x/author}</res2>

org.exist.xquery.XPathException:
variable $x is not bound.
```

The parser saw this as a sequence formed of:

- a `for`-`return` expression
- a path expression

You probably meant:

```
for $x in doc("bib.xml")//book
return (<res1>{$x/title}</res1>,
        <res2>{$x/author}</res2>)
```

When XQuery doesn’t behave as expected

The query gives unexpected results:

Query

```
//book[price<"39.95"]
```

Result

```
<book year="1999">
   <title>The Economics of Technology...</title>
   <editor>
      <last>Gerbarg</last>
      <first>Darcy</first>
         <affiliation>CITI</affiliation>
   </editor>
   <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
</book>
```

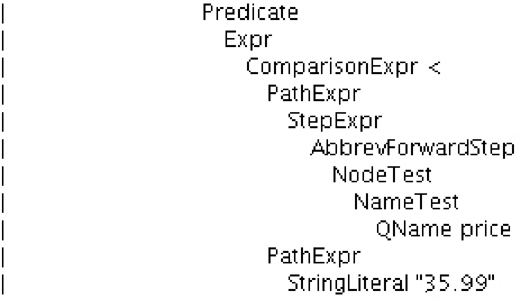
When XQuery doesn’t behave as expected

The query gives unexpected results:

Query

//book[price<"39.95"]

Parsing tree (partial):



```
|                    Predicate
|                      Expr
|                        ComparisonExpr <
|                          PathExpr
|                            StepExpr
|                              AbbrevForwardStep
|                                NodeTest
|                                  NameTest
|                                    QName price
|                          PathExpr
|                            StringLiteral "35.99"
```

The comparison is done in the string domain.

When XQuery doesn’t behave as expected

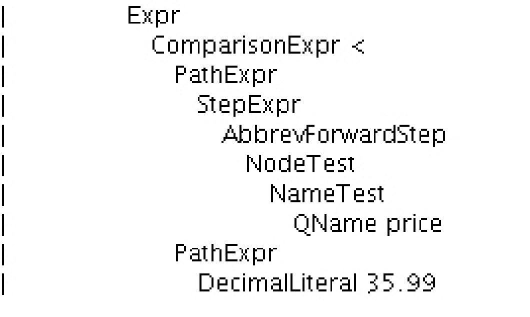
The query gives unexpected results:

This query has the desired meaning:

Query

//book[price<39.95]

Parsing tree (partial):

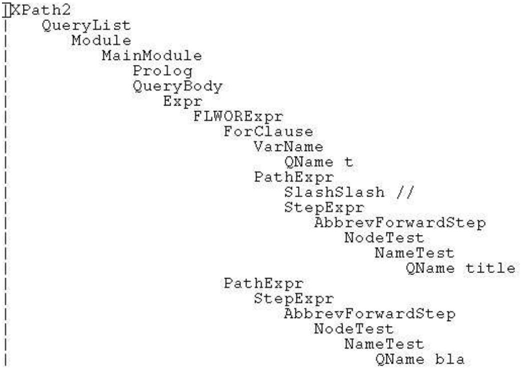


```
|          Expr
|            ComparisonExpr <
|              PathExpr
|                StepExpr
|                  AbbrevForwardStep
|                    NodeTest
|                      NameTest
|                        QName price
|              PathExpr
|                DecimalLiteral 35.99
```

This time, the comparison is done in the numeric domain.

When XQuery doesn't behave as expected

```
for $b in doc("bib.xml")//book return bla
```



```
 XPath2
|   QueryList
|      Module
|         MainModule
|            Prolog
|            QueryBody
|               Expr
|                  FLWORExpr
|                     ForClause
|                        VarName
|                           QName t
|                        PathExpr
|                           SlashSlash //
|                           StepExpr
|                              AbbrevForwardStep
|                                 NodeTest
|                                    NameTest
|                                       QName title
|                     PathExpr
|                        StepExpr
|                           AbbrevForwardStep
|                              NodeTest
|                                 NameTest
|                                    QName bla
```

When XQuery doesn't behave as expected

The query gives unexpected results

```
for $b in doc("bib.xml")//book return bla
```

The last part of the expression is a *path expression testing if the context node is named bla*.
If the context is empty, the query has an empty result.
Maybe you meant:

```
for $b in doc("bib.xml")//book return "bla"
```

More on comparisons

1. Two atomic values:
   - determine the types of both operands
   - cast then to a common type
   - compare the values according to the rules of that type
2. One atomic value and a node:
   - Cast the node to a string, then proceed as above.
3. Two lists (one list may be of length one):
   - Compare all list item pairs, return true if the predicate is satisfied at least for one item pair.

Casting is described in the XQuery Functions and Operators document.

Going in depth: W3C specifications

Web documents found under *http://www.w3.org*. *Not* articles! Typically very long *but* navigable. The Introduction clarifies the document role, then go directly to the interesting (sub)sections.

XML specification:

- XML and DTDs
- Namespaces in XML
- XML Schema

XQuery specification:

- XQuery 1.0 specification (syntax)
- XPath functions and operators (`op:equal`, `fn:text`, `fn:distinct-values`, `fn:document`, `op:gt`, ...)
- XQuery data model

XQuery implementations

Among those that are free and/or open-source:

Galax : complete, not very efficient

Saxon : in memory; by Michael Kay, XSL guru

MonetDB : based on in-memory column-oriented engine; among the fastest

eXist : very user-friendly interface

QizX : Xavier Franc. Nice but not great

BerkeleyDB XML : now belongs to Oracle

SQL/XML: bridging the two worlds

Recent SQL versions (2003) include:

- a native XML atomic type, which can be queried in XQuery style
- a set of XML publishing functions: extracting XML elements out of relational data by querying
- mapping rules: exporting relational tables in XML

Advantages:

- Unified manipulation of relational and XML data
- Efficient relational query engine well exploited
- Ease of transformation from one format to another

Disadvantage:

- Complexity

SQL/XML: bridging the two worlds

XML publishing functions:

```
select xmlelement(name Customer,
                  xmlattributes(c.city as city),
                  xmlforest(c.CustID,
                              c.Name as CustName))
from customer c
```

Mixed querying:

```
select customer, XMLExtract(order, ’/order/@date’)
from orders
where XMLExists(order,
                ’/order[//desc/text()="Shoes"]’)
      =1
```

The precise SQL/XML syntax sometimes depends on the vendor.