XML and Semantic Web Technologies

I. XML / 6. XML Query Language (XQuery)

Lars Schmidt-Thieme

Information Systems and Machine Learning Lab (ISMLL)
Institute of Economics and Information Systems
& Institute of Computer Science
University of Hildesheim
http://www.ismll.uni-hildesheim.de

I. XML / 6. XML Query Language (XQuery)

XQuery Specification

XQuery is specified in

1. XQuery 1.0: An XML Query Language (2nd ed., Rec 2010/12/14) and
2. XML Syntax for XQuery 1.0 (XQueryX; 2nd ed., Rec 2010/12/14)

as well as documents about requirements, use cases, serialization, and formal semantics.

XQuery **extends** XPath 2.0, i.e., (most) any XPath expressions **are** XQuery "queries"

XQuery does not have an XML Syntax
(like XPath, but contrary to XSLT).

XQuery Modules

The XQuery processing unit is the **module**:

$\langle \textit{Module} \rangle$:= (`xquery version` $\langle \textit{StringLiteral} \rangle$ `;`)?
($\langle \textit{ModuleDecl} \rangle$ `;`)?
$\langle \textit{Prolog} \rangle$
$\langle \textit{Expr} \rangle$

Usually one module is stored in one file.

Library modules have a module declaration, but no body expression;
main modules have a body expression, but no module declaration.

All XPath expressions are XQuery expressions $\langle \textit{Expr} \rangle$.

```
xquery version "1.0" ;
//title[contains(string(.),"XML")]/../author
```

Figure 1: Example XQuery consisting of an XPath expression.

Literal Result Nodes

Like in XSLT, nodes can be created in two ways:

1. literal result elements (called **direct constructors** in the XQuery spec), their attributes, namespaces and nested character data is copied literally to the result tree.

   XML comments and PIs can be created in the same way (as XQuery does not have an XML syntax).

```
xquery version "1.0" ;
<article author="John Doe" version="2004/06/07">
  <title>What <em>others</em> say</title>
  A <em>short</em><!-- 20 pages--> overview ...
</article>
```

Figure 2: Example XQuery consisting of literal result items only.

Literal Result Node Sequences

```
xquery version "1.0" ;
<!-- first ideas -->,
<?xml-stylesheet href='article.css' type='text/css'?>,
<article author="John Doe" version="2004/06/07">
  <title>What <em>others</em> say</title>
  A <em>short</em><!-- 20 pages--> overview ...
</article>
```

Figure 3: Example XQuery consisting of literal result items almost only.

Computed Result Nodes

2. computed result nodes (called **computed constructors** in the XQuery spec):

```
⟨computedConstructor⟩ := document { ⟨Expr⟩? }
    | element ⟨QName⟩ { ⟨Expr⟩? }
    | attribute ⟨QName⟩ { ⟨Expr⟩? }
    | text { ⟨Expr⟩? }
    | comment { ⟨Expr⟩? }
    | processing-instruction ⟨NCName⟩ { ⟨Expr⟩? }
    | namespace ⟨NCName⟩ { ⟨Expr⟩? }
```

⟨*QName*⟩ and ⟨*NCName*⟩ can also be replaced by `{` ⟨*Expr*⟩ `}`.

Computed Result Nodes

```
<?xml version="1.1"?>
<!-- first ideas -->
<?xml-stylesheet href='article.css' type='text/css'?>
<article author="John Doe" version="2004/06/07">
  <title>What <em>others</em> say</title>
  A <em>short</em><!-- 20 pages--> overview ...
</article>
```

Figure 4: Sample XML document.

```
xquery version "1.0" ;
for $a in //article return
  element article {
    for $att in $a/@* return
      element { name($att) } { string($att) },
    element title { string($a/title) }
  }
```

Figure 5: XQuery `element.xq` creating elements with fixed and computed names.

```
<?xml version="1.0" encoding="UTF-8"?>
<article>
   <author>John Doe</author>
   <version>2004/06/07</version>
   <title>What others say</title>
</article>
```

Figure 6: Result document of the query.

Nesting Computed Result Nodes in Literal Result Nodes

In

- values of attributes and
- the contents

of literal result elements, expressions of type

$\{\ \langle \mathit{Expr} \rangle\ \}$

can be used to insert computed items (for attribute values) or nodes (for element content).

```
xquery version "1.0" ;
for $a in //article return
 <article>
  { for $att in $a/@* return
      element { name($att) } { string($att) },
    <title>{ string($a/title) }</title>
  }
 </article>
```

Figure 7: XQuery creating elements with fixed and computed names.

If in these contexts "{" or "}" should appear literally, the symbols must be escaped as "{{" and "}}".

Performing XQuery Queries by Saxon

XQuery queries can be performed, e.g., by Saxon.

call (with saxon8.jar in classpath):

```
java net.sf.saxon.Query -s anarticle.xml element.xq
```

I. XML / 6. XML Query Language (XQuery)

Clauses

⟨*FLWORExpr*⟩ := (⟨*ForClause*⟩ | ⟨*LetClause*⟩)+
(`where` ⟨*ExprSingle*⟩)? ⟨*OrderByClause*⟩?
`return` ⟨*Expr*⟩

⟨*ForClause*⟩ := `for`
`$` ⟨*QName*⟩ (`as` ⟨*SequenceType*⟩)? (`at $` ⟨*QName*⟩)? `in` ⟨*Expr*⟩
(`, $` ⟨*QName*⟩ (`as` ⟨*SequenceType*⟩)? (`at $` ⟨*QName*⟩)? `in` ⟨*Expr*⟩)*

⟨*LetClause*⟩ := `let $` ⟨*QName*⟩ (`as` ⟨*SequenceType*⟩)? `:=` ⟨*Expr*⟩
(`, $` ⟨*QName*⟩ (`as` ⟨*SequenceType*⟩)? `:=` ⟨*Expr*⟩)*

XPath's for-expressions is a special case of an XQuery FLWOR expression.
New is:

- `let` binds additional variables,
- `where` filters tuples,
- `order by` orders tuples,
- `at $` ⟨*QName*⟩ binds an additional positional variable,
- `as` ⟨*SequenceType*⟩ types the for-/let-variable.

`for` and `let` Clauses

```
xquery version "1.0" ;
for $s in (<one/>, <two/>, <three/>)
  return <out>{$s}</out>
```

Figure 8: FLWOR-expression using `for`.

```
xquery version "1.0" ;
let $s := (<one/>, <two/>, <three/>)
  return <out>{$s}</out>
```

Figure 9: FLWOR-expression using `let`.

```
<?xml version="1.0"?>
<out>
   <one/>
</out>
<out>
   <two/>
</out>
<out>
   <three/>
</out>
```

Figure 10: Result of `for`-expression.

```
<?xml version="1.0"?>
<out>
   <one/>
   <two/>
   <three/>
</out>
```

Figure 11: Result of `let`-expression.

`where` Clause

```
xquery version "1.0" ;
let $inputvalues := 1 to 1000 return
  avg(for $x at $i in $inputvalues
      where $i mod 100 = 0
      return $x)
```

Figure 12: FLWOR-expression using `where`.

```
xquery version "1.0" ;
let $inputvalues := 1 to 1000 return
  avg($inputvalues[position() mod 100 = 0]
```

Figure 13: Same query using a predicate.

```
550
```

Figure 14: Result of the queries.

`order by` Clause

⟨*OrderByClause*⟩ := (`order by` | `stable order by`) ⟨*OrderSpecList*⟩

⟨*OrderSpecList*⟩ := ⟨*OrderSpec*⟩ (`,` ⟨*OrderSpec*⟩)*

⟨*OrderSpec*⟩ := ⟨*Expr*⟩ ⟨*OrderModifier*⟩

⟨*OrderModifier*⟩ := (`ascending` | `descending`)?
(`empty greatest` | `empty least`)?

```
xquery version "1.0" ;
<html><body>
 Authors: <ol>
 { for $a in distinct-values(//author)
  let $sn := substring-after($a, ' '),
      $fn := substring-before($a, ' ')
  order by $sn, $fn
  return <li>{ concat($sn, ", ", $fn) }</li>
 }
 </ol></body></html>
```

Figure 15: FLWOR-expression with `order by` clause.

```
<html>
  <body>
    Authors:
    <ol>
      <li>Eckstein, Rainer</li>
      <li>Eckstein, Silke</li>
      <li>Muellner, Leonard</li>
      <li>T. Ray, Erik</li>
      <li>Walsh, Norman</li>
    </ol>
  </body>
</html>
```

Figure 16: Result of the query on the `books.xml` document.

Positional Variable with `at`

`at` allows to bind the position of the actual item to a variable.

This is the only possibility to access the position
(as the XPath function `position()` retrieves the position of the context item only).

```
xquery version "1.0" ;
<books>{
 for $b at $no in //book return
  <book no="{$no}">{ string($b/title) }</book>
}</books>
```

Figure 17: FLWOR-expression with positional variable.

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book no="1">XML und Datenmodellierung</book>
  <book no="2">Learning XML</book>
  <book no="3">DocBook: The Definitive Guide</book>
</books>
```

Figure 18: Result of the query on the `books.xml` document.

XPath's `for`-expressions and XQuery's FLWOR-expressions

clause	equivalent in XPath
if	if
– at	position(), if used in where-clause; none, else
– as	cast as, for atomic types; none, else
let	variable can be replaced by its value
where	predicates in if-expressions
order by	none

I. XML / 6. XML Query Language (XQuery)

Namespaces

$\langle$*Prolog*$\rangle$:= (($\langle$*Setter*$\rangle$ | $\langle$*ImportDecl*$\rangle$
| $\langle$*ImportSchemaDecl*$\rangle$ | $\langle$*NamespaceDecl*$\rangle$ `;`)*
(($\langle$*VarDecl*$\rangle$ | $\langle$*FunctionDecl*$\rangle$ | $\langle$*OptionDecl*$\rangle$) `;`)*

$\langle$*NamespaceDecl*$\rangle$:= `declare namespace` $\langle$*NCName*$\rangle$ `=` $\langle$*StringLiteral*$\rangle$
| `declare default element namespace` $\langle$*StringLiteral*$\rangle$
| `declare default function namespace` $\langle$*StringLiteral*$\rangle$

Variables

$\langle$*VarDecl*$\rangle$:= `declare variable $` $\langle$*QName*$\rangle$ (`as` $\langle$*SequenceType*$\rangle$)? ((`:=` $\langle$*Expr*$\rangle$) | `external`)

XQuery variables are read-only.

Alternatively, variables can be bound with a let-clause of a FLWOR-expression.

```
xquery version "1.0" ;
declare variable $no := 0 ;
<books>{
 for $b in //book
 let $no := $no + 1 return
  <book no="{$no}">{ string($b/title) }</book>
}</books>
```

Figure 19: What might happen here?

Variables

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
  <book no="1">XML und Datenmodellierung</book>
  <book no="1">Learning XML</book>
  <book no="1">DocBook: The Definitive Guide</book>
</books>
```

Figure 20: Result of broken query on last slide.

Defining Functions

⟨*FunctionDecl*⟩ := `declare function` ⟨*QName*⟩ `(` ⟨*ParamList*⟩? `)`
(`as` ⟨*SequenceType*⟩)? (`{` ⟨*Expr*⟩ `}` | `external`)

⟨*ParamList*⟩ := ⟨*Param*⟩ (`,` ⟨*Param*⟩)*

⟨*Param*⟩ := `$` ⟨*QName*⟩ (`as` ⟨*SequenceType*⟩)?

```
xquery version "1.0" ;
declare namespace f = "http://www.cgnm.de/xml/xquery/functions";
declare function f:shortauthor($author as node()*) {
 let $numAuthors := count($author) return
   concat($author[1], if ($numAuthors=2) then concat(' and ', $author[2])
              else if ($numAuthors>=3) then ' et al.' else '')
};
<html><body><ol>
 { for $b in //book return
     <li>{ concat(f:shortauthor($b/author), ": ", string($b/title), ", ", $b/year) }</li>
 }</ol></body></html>
```

Figure 21: Function declaration.

For Saxon, names of user-defined functions must have a namespace.

Defining Functions

```
<html>
  <body>
    <ol>
      <li>Rainer Eckstein and Silke Eckstein: XML und Datenmodellierung, 2004</li
      <li>Erik T. Ray: Learning XML, 2003</li>
      <li>Norman Walsh and Leonard Muellner: DocBook: The Definitive Guide, 19
    </ol>
  </body>
</html>
```

Figure 22: Result of query on last slide.

Library Modules

$\langle \textit{ModuleDecl} \rangle$:= `module namespace` $\langle \textit{NCName} \rangle$ `=` $\langle \textit{StringLiteral} \rangle$

$\langle \textit{ImportDecl} \rangle$:= `import module` (`namespace` $\langle \textit{NCName} \rangle$ `=`)? $\langle \textit{StringLiteral} \rangle$
(`at` $\langle \textit{StringLiteral} \rangle$)?

```
xquery version "1.0" ;
module namespace f = "http://www.cgnm.de/xml/xquery/functions";
declare function f:shortauthor($author as node()*) {
 let $numAuthors := count($author) return
  concat($author[1], if ($numAuthors=2) then concat(' and ', $author[2])
               else if ($numAuthors>=3) then ' et al.' else '')
};
```

Figure 23: Library module `biblib.xq`.

```
xquery version "1.0" ;
import module namespace f = "http://www.cgnm.de/xml/xquery/functions" at "biblib.
<html><body><ol>
 { for $b in //book return
    <li>{ concat(f:shortauthor($b/author), ": ", string($b/title), ", ", $b/year) }</li>
 }</ol></body></html>
```

Figure 24: Query importing a library module.

I. XML / 6. XML Query Language (XQuery)

Working with Types

To use types defined in a schema, e.g., as

- types of variables,
- as types of function parameters or return types,
- in cast expressions, etc.

the namespace of the schema defining the types has to be declared.

```
xquery version "1.0" ;
declare namespace f = "http://www.cgnm.de/xml/xquery/functions";
declare namespace xs="http://www.w3.org/2001/XMLSchema";

declare function f:shortauthor($author as node()*) as xs:string {
 let $numAuthors := count($author) return
  concat($author[1], if ($numAuthors=2) then concat(' and ', $author[2])
                else if ($numAuthors>=3) then ' et al.' else '')
};
<html><body><ol>
 { for $b in //book return
    <li>{ concat(f:shortauthor($b/author), ": ", string($b/title), ", ", $b/year) }</li>
 }</ol></body></html>
```

Figure 25: Working with XML Schema types.

Validating and Static Type Checking (1/5)

```
<?xml version="1.0"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.cgnm.de/xml/books.xsd"
           xmlns="http://www.cgnm.de/xml/books.xsd"
           elementFormDefault="qualified">
 <xs:element name="books">
  <xs:complexType>
   <xs:sequence maxOccurs="unbounded">
    <xs:element ref="book"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="book">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="author" minOccurs="1" maxOccurs="unbounded"
                type="xs:string"/>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="year" type="xs:string"/>
```

```
   </xs:sequence>
   <xs:attribute name="isbn" type="xs:string"/>
   <xs:attribute name="cites">
    <xs:simpleType><xs:list itemType="xs:string"/></xs:simpleType>
   </xs:attribute>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 26: XML Schema `books.xsd` of the source document.

```
<?xml version="1.1"?>
<books>
 <book isbn="3-89864-222-4" cites="0-596-00420-6 1-565-92580-7">
  <author>Rainer Eckstein</author><author>Silke Eckstein</author>
  <title>XML und Datenmodellierung</title><year>2004</year></book>
 <book isbn="0-596-00420-6">
  <author>Erik T. Ray</author><title>Learning XML</title><year>2003</year></bo
 <book isbn="1-565-92580-7">
  <author>Norman Walsh</author><author>Leonard Muellner</author>
  <title>DocBook: The Definitive Guide</title><year>1999</year></book>
</books>
```

Figure 27: Source document `books.xml`.

Validating and Static Type Checking (2/5)

```
<?xml version="1.0"?>
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.cgnm.de/xml/html-toy.xsd"
     xmlns="http://www.cgnm.de/xml/html-toy.xsd"
     elementFormDefault="qualified">

 <xs:element name="html">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="body">
     <xs:complexType mixed="true">
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
       <xs:element name="ol">
        <xs:complexType>
         <xs:sequence minOccurs="0" maxOccurs="unbounded">
          <xs:element name="li" type="xs:string"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
```

```
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 28: XML Schema of the target document.

Validating and Static Type Checking (3/5)

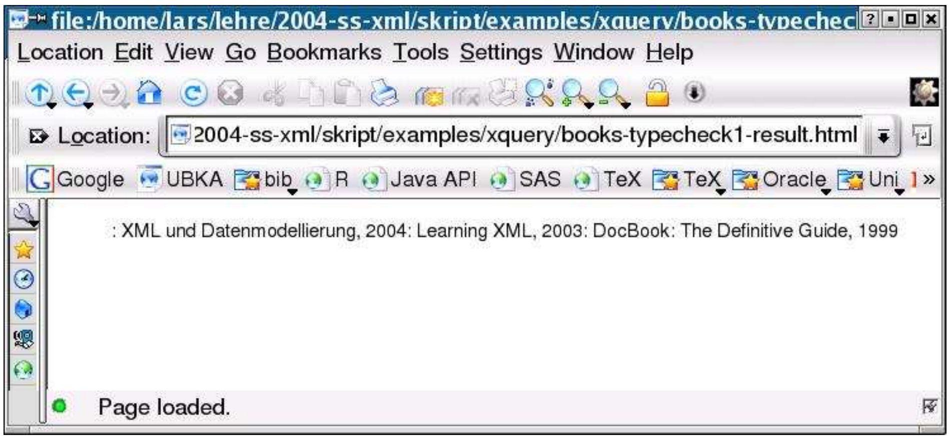
```
xquery version "1.0" ;
declare namespace f = "http://www.cgnm.de/xml/xquery/functions";
declare function f:shortauthor($author as node()*) {
 let $numAuthors := count($author) return
   concat($author[1], if ($numAuthors=2) then concat(' and ', $author[2])
                 else if ($numAuthors>=3) then ' et al.' else '')
};
<html><body><ol>
 { for $b in //book return
     <item>{ concat(f:shortauthor($b/auhtor), ": ", string($b/title), ", ", $b/year) }</iten
 }</ol></body></html>
```

Figure 29: Query containing errors w.r.t. schemata.

Validating and Static Type Checking (3.5/5)

```
<html>
  <body>
    <ol>
      <item>: XML und Datenmodellierung, 2004</item>
      <item>: Learning XML, 2003</item>
      <item>: DocBook: The Definitive Guide, 1999</item>
    </ol>
  </body>
</html>
```

Figure 30: Result of broken query on last slide.


file:/home/lars/lehre/2004-ss-xml/skript/examples/xquery/books-typechec
Location Edit View Go Bookmarks Tools Settings Window Help
Location: 2004-ss-xml/skript/examples/xquery/books-typecheck1-result.html
Google UBKA bib R Java API SAS TeX TeX Oracle Uni

: XML und Datenmodellierung, 2004: Learning XML, 2003: DocBook: The Definitive Guide, 1999

Page loaded.

Figure 31: Result viewed in HTML browser.

Kinds of Validation

Validity of specific input documents can be checked by the parser.

Validity of specific output documents can be checked by:

1. validating output as a post-process or by,
2. validating output while it is generated.

This is called **dynamic type checking** (depends on data, done at runtime) as opposed to **static type checking** (does not depend on data, done at query compile time).

Validate Expressions

$\langle$*ImportSchemaDecl*$\rangle$:= `import schema` $\langle$*SchemaPrefix*$\rangle$? $\langle$*StringLiteral*$\rangle$
(`at` $\langle$*StringLiteral*$\rangle$)?

$\langle$*SchemaPrefix*$\rangle$:= (`namespace` $\langle$*NCName*$\rangle$ `=`) | `default element namespace`

$\langle$*ValidateExpr*$\rangle$:= `validate` (`lax` | `strict`)? `{` $\langle$*Expr*$\rangle$ `}`

validate returns its argument node with type annotation
(and raises an error, if it is not valid w.r.t. its associated schema).

Validating and Static Type Checking (4/5)

```
xquery version "1.0" ;
declare namespace f = "http://www.cgnm.de/xml/xquery/functions";
declare function f:shortauthor($author as node()*) {
 let $numAuthors := count($author) return
   concat($author[1], if ($numAuthors=2) then concat(' and ', $author[2])
              else if ($numAuthors>=3) then ' et al.' else '')
};
import schema namespace bk="http://www.cgnm.de/xml/books.xsd" at "books.xsd";
import schema default element namespace "http://www.cgnm.de/xml/html-toy.xsd" a
validate strict {
<html><body><ol>
 { for $b in //bk:book return
    <item>{ concat(f:shortauthor($b/bk:auhtor), ": ", string($b/bk:title), ", ", $b/bk:yea
 }</ol></body></html>
}
```

Figure 32: Query containing errors w.r.t. schemata, with schema import.

Validating and Static Type Checking (5/5)

```
xquery version "1.0" ;
declare namespace f = "http://www.cgnm.de/xml/xquery/functions";
declare function f:shortauthor($author as node()*) {
 let $numAuthors := count($author) return
   concat($author[1], if ($numAuthors=2) then concat(' and ', $author[2])
              else if ($numAuthors>=3) then ' et al.' else '')
};
import schema namespace bk="http://www.cgnm.de/xml/books.xsd" at "books.xsd";
import schema default element namespace "http://www.cgnm.de/xml/html-toy.xsd" a
validate strict {
<html><body><ol>
 { for $b in //bk:book return
    <li>{ concat(f:shortauthor($b/bk:author), ": ", string($b/bk:title), ", ", $b/bk:year) }
 }</ol></body></html>
}
```

Figure 33: Query with errors fixed.

Comparison XSLT–XQuery

a. commonalities:

topic	details
0. basic expressions	both languages extend XPath.
1. variables	read-only
2. definition of functions	callable in XPath expressions
3. nested iterations / joins	FLWOR-expressions and `for-each`, `for-each-group`
4. creating nodes	both languages provide two mechanisms: • literal result nodes, • computed result nodes.
5. type strictness	both languages build on XML Schema and allow static type checking

Comparison XSLT–XQuery

b. differences:

topic	XSLT	XQuery
1. syntax	XML	non-XML
2. user-defined functions	can be used in expressions and in `value-of`	can be used everywhere
3. FLWOR expressions	can be replaced by combined statements (procedural)	declarative
4. template rules	`template` and `apply-templates`	–
5. type strictness	maybe not encouraged by templates	encouraged by FLWOR expressions

Summary

- XQuery is another **query / processing language** for XML (besides XSLT).
- XQuery **extends XPath** (and thus has a non-XML syntax; but there also is XQueryX).
- XQuery allows the **construction of result nodes** both, literally or computed.
- XQuery provides **FLOWR expressions** as powerful queries.
- XQuery **supports XML Schema** and esp. **static type checking**.