An Introduction to XML and Web Technologies

Querying XML Documents with XQuery

Anders Møller & Michael I. Schwartzbach

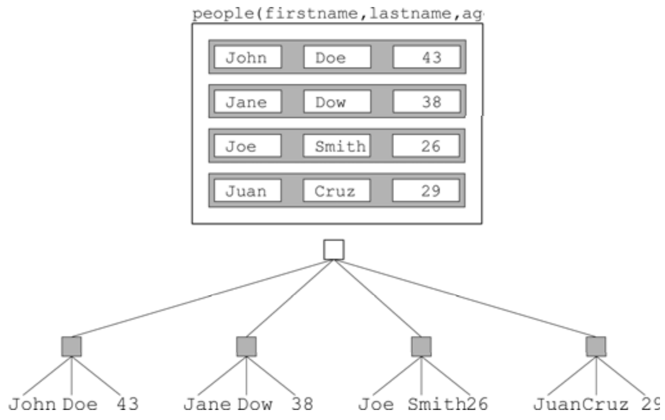
Objectives

- How XML generalizes relational databases
- The XQuery language
- How XML may be supported in databases

XQuery 1.0

- XML documents naturally generalize database relations
- XQuery is the corresponding generalization of SQL

From Relations to Trees



Only Some Trees are Relations

- They have *height two*
- The root has an *unbounded* number of children
- All nodes in the second layer (records) have a *fixed* number of child nodes (fields)

Trees Are Not Relations

- Not all trees satisfy the previous characterization
- Trees are *ordered*, while both rows and columns of tables may be permuted without changing the meaning of the data

A Student Database

Students(id,name,age)

100026	Joe Average	21
100078	Jack Doe	18

Majors(id,major)

100026	Biology
100078	Physics
100078	XML Science

Grades(id,course,grade)

100026	Math 101	C-
100026	Biology 101	C+
100026	Statistics 101	D
100078	Math 101	A+
100078	XML 101	A-
100078	Physics 101	B+
100078	XML 102	A

A More Natural Model (1/2)

```
<students>
   <student id="100026">
      <name>Joe Average</name>
      <age>21</age>
      <major>Biology</major>
      <results>
        <result course="Math 101" grade="C-"/>
        <result course="Biology 101" grade="C+"/>
        <result course="Statistics 101" grade="D"/>
      </results>
   </student>
```

A More Natural Model (2/2)

```
  <student id="100078">
    <name>Jack Doe</name>
    <age>18</age>
    <major>Physics</major>
    <major>XML Science</major>
    <results>
      <result course="Math 101" grade="A"/>
      <result course="XML 101" grade="A-"/>
      <result course="Physics 101" grade="B+"/>
      <result course="XML 102" grade="A"/>
    </results>
  </student>
</students>
```


Usage Scenario: Data-Oriented

- We want to carry over the kinds of queries that we performed in the original relational model

Usage Scenario: Document-Oriented

- Queries could be used
  - to retrieve parts of documents
  - to provide dynamic indexes
  - to perform context-sensitive searching
  - to generate new documents as combinations of existing documents

Usage Scenario: Programming

- Queries could be used to automatically generate documentation

Usage Scenario: Hybrid

- Queries could be used to data mine hybrid data, such as patient records

XQuery Design Requirements

- Must have at least one *XML syntax* and at least one *human-readable* syntax
- Must be *declarative*
- Must be *namespace aware*
- Must *coordinate* with XML Schema
- Must support *simple* and *complex* datatypes
- Must *combine* information from multiple documents
- Must be able to *transform* and *create* XML trees

Relationship to XPath

- XQuery 1.0 is a *strict superset* of XPath 2.0
- Every XPath 2.0 expression is directly an XQuery 1.0 expression (a query)
- The extra expressive power is the ability to
  - *join* information from different sources and
  - *generate* new XML fragments

Relationship to XSLT

- XQuery and XSLT are both *domain-specific languages* for combining and transforming XML data from multiple sources
- They are vastly *different in design*, partly for historical reasons
- XQuery is designed from scratch, XSLT is an intellectual descendant of CSS
- Technically, they may *emulate* each other

XQuery Prolog

- Like XPath expressions, XQuery expressions are evaluated relatively to a *context*
- This is explicitly provided by a *prolog*
- Settings define various parameters for the XQuery processor language, such as:

```
xquery version "1.0";
declare xmlspace preserve;
declare xmlspace strip;
```


More From the Prolog

```
declare default element namespace URI;
declare default function namespace URI;
import schema at URI;
declare namespace NCName = URI;
```


Implicit Declarations

```
declare namespace xml =
  "http://www.w3.org/XML/1998/namespace";
declare namespace xs =
  "http://www.w3.org/2001/XMLSchema";
declare namespace xsi =
  "http://www.w3.org/2001/XMLSchema-instance";
declare namespace fn =
  "http://www.w3.org/2005/11/xpath-functions";
declare namespace xdt =
  "http://www.w3.org/2005/11/xpath-datatypes";
declare namespace local =
    "http://www.w3.org/2005/11/xquery-local-functions";
```


XPath Expressions

- XPath expressions are also XQuery expressions
- The XQuery prolog gives the required static context
- The initial context node, position, and size are undefined

Datatype Expressions

- Same atomic values as XPath 2.0
- Also lots of primitive simple values:

```
xs:string("XML is fun")
xs:boolean("true")
xs:decimal("3.1415")
xs:float("6.02214199E23")
xs:dateTime("1999-05-31T13:20:00-05:00")
xs:time("13:20:00-05:00")
xs:date("1999-05-31")
xs:gYearMonth("1999-05")
xs:gYear("1999")
xs:hexBinary("48656c6c6f0a")
xs:base64Binary("SGVsbG8K")
xs:anyURI("http://www.brics.dk/ixwt/")
xs:QName("rcp:recipe")
```


XML Expressions

- XQuery expressions may compute *new XML nodes*
- Expressions may denote element, character data, comment, and processing instruction nodes
- Each node is created with a unique *node identity*
- Constructors may be either *direct* or *computed*

Direct Constructors

- Uses the standard XML syntax
- The expression

```
<foo><bar/>baz</foo>
```

- evaluates to the given XML fragment
- Note that

```
<foo/> is <foo/>
```

- evaluates to false

Namespaces in Constructors (1/3)

```
declare default element namespace "http://businesscard.org";
<card>
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

Namespaces in Constructors (2/3)

```
declare namespace b = "http://businesscard.org";
<b:card>
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.com</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo uri="widget.gif"/>
</b:card>
```


Namespaces in Constructors (3/3)

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```


Enclosed Expressions

```
<foo>1 2 3 4 5</foo>
<foo>{1, 2, 3, 4, 5}</foo>
<foo>{1, "2", 3, 4, 5}</foo>
<foo>{1 to 5}</foo>
<foo>1 {1+1} {" "} {"3"} {" "} {4 to 5}</foo>
```

```
<foo bar="1 2 3 4 5"/>
<foo bar="{1, 2, 3, 4, 5}"/>
<foo bar="1 {2 to 4} 5"/>
```


Explicit Constructors

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

```
element card {
  namespace { "http://businesscard.org" },
  element name { text { "John Doe" } },
  element title { text { "CEO, Widget Inc." } } ,
  element email { text { "john.doe@widget.com" } },
  element phone { text { "(202) 555-1414" } },
  element logo {
    attribute uri { "widget.gif" }
  }
}
```

Computed QNames

```
element { "card" } {
  namespace { "http://businesscard.org" },
  element { "name" } { text { "John Doe" } },
  element { "title" } { text { "CEO, Widget Inc." } },
  element { "email" } { text { "john.doe@widget.com" } },
  element { "phone" } { text { "(202) 555-1414" } },
  element { "logo" } {
    attribute { "uri" } { "widget.gif" }
  }
}
```


Biliingual Business Cards

```
element { if ($lang="Danish") then "kort" else "card" } {
  namespace { "http://businesscard.org" },
  element { if ($lang="Danish") then "navn" else "name" }
    { text { "John Doe" } },
  element { if ($lang="Danish") then "titel" else "title" }
    { text { "CEO, Widget Inc." } },
  element { "email" }
    { text { "john.doe@widget.inc" } },
  element { if ($lang="Danish") then "telefon" else "phone"}
    { text { "(202) 456-1414" } },
  element { "logo" } {
    attribute { "uri" } { "widget.gif" }
  }
}
```


FLWOR Expressions

- Used for general queries:

```
<doubles>
  { for $s in fn:doc("students.xml")//student
    let $m := $s/major
    where fn:count($m) ge 2
    order by $s/@id
    return <double>
             { $s/name/text() }
           </double>
  }
</doubles>
```


The Difference Between For and Let (1/4)

```
for $x in (1, 2, 3, 4)
let $y := ("a", "b", "c")
return ($x, $y)
```

```
1, a, b, c, 2, a, b, c, 3, a, b, c, 4, a, b, c
```

The Difference Between For and Let (2/4)

```
let $x in (1, 2, 3, 4)
for $y := ("a", "b", "c")
return ($x, $y)
```

⇩

```
1, 2, 3, 4, a, 1, 2, 3, 4, b, 1, 2, 3, 4, c
```


The Difference Between For and Let (3/4)

```
for $x in (1, 2, 3, 4)
for $y in ("a", "b", "c")
return ($x, $y)
```

⇩

```
1, a, 1, b, 1, c, 2, a, 2, b, 2, c,
3, a, 3, b, 3, c, 4, a, 4, b, 4, c
```


The Difference Between For and Let (4/4)

```
let $x := (1, 2, 3, 4)
let $y := ("a", "b", "c")
return ($x, $y)
```

⇩

```
1, 2, 3, 4, a, b, c
```


Computing Joins

- What recipes can we (sort of) make?

```
declare namespace rcp = "http://www.brics.dk/ixwt/recipes";
for $r in fn:doc("recipes.xml")//rcp:recipe
for $i in $r//rcp:ingredient/@name
for $s in fn:doc("fridge.xml")//stuff[text()=$i]
return $r/rcp:title/text()
```

```
<fridge>
  <stuff>eggs</stuff>
  <stuff>olive oil</stuff>
  <stuff>ketchup</stuff>
  <stuff>unrecognizable moldy thing</stuff>
</fridge>
```

Inverting a Relation

```
declare namespace rcp = "http://www.brics.dk/ixwt/recipes";
<ingredients>
  { for $i in distinct-values(
                  fn:doc("recipes.xml")//rcp:ingredient/@name
                )
    return <ingredient name="{$i}">
             { for $r in fn:doc("recipes.xml")//rcp:recipe
               where $r//rcp:ingredient[@name=$i]
               return <title>$r/rcp:title/text()</title>
             }
           </ingredient>
  }
</ingredients>
```


Sorting the Results

```
declare namespace rcp = "http://www.brics.dk/ixwt/recipes";
<ingredients>
  { for $i in distinct-values(
                  fn:doc("recipes.xml")//rcp:ingredient/@name
                )
    order by $i
    return <ingredient name="{$i}">
             { for $r in fn:doc("recipes.xml")//rcp:recipe
               where $r//rcp:ingredient[@name=$i]
               order by $r/rcp:title/text()
               return <title>$r/rcp:title/text()</title>
             }
           </ingredient>
  }
</ingredients>
```


A More Complicated Sorting

```
for $s in document("students.xml")//student
order by
  fn:count($s/results/result[fn:contains(@grade,"A")]) descending,
  fn:count($s/major) descending,
  xs:integer($s/age/text()) ascending
return $s/name/text()
```


Using Functions

```
declare function local:grade($g) {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B")  then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C")   then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D")  then 1.0
  else if ($g="D-") then 0.7 else 0
};

declare function local:gpa($s) {
  fn:avg(for $g in $s/results/result/@grade return local:grade($g))
};

<gpas>
  { for $s in fn:doc("students.xml")//student
    return <gpa id="{$s/@id}" gpa="{local:gpa($s)}"/> }
</gpas>
```

A Height Function

```
declare function local:height($x) {
  if (fn:empty($x/*)) then 1
  else fn:max(for $y in $x/* return local:height($y))+1
};
```


A Textual Outline

```
Cailles en Sarcophages
  pastry
    chilled unsalted butter
    flour
    salt
    ice water
  filling
    baked chicken
      marinated chicken
        small chickens, cut up
        Herbes de Provence
        dry white wine
        orange juice
        minced garlic
        truffle oil
...
```


Computing Textual Outlines

```
declare namespace rcp = "http://www.brics.dk/ixwt/recipes";
declare function local:ingredients($i,$p) {
  fn:string-join(
    for $j in $i/rcp:ingredient
    return fn:string-join(($p,$j/@name,"
",local:ingredients($j,fn:concat($p,"  "))),""),"")
};

declare function local:recipes($r) {
  fn:concat($r/rcp:title/text(),"
",local:ingredients($r,"  "))
};

fn:string-join(
  for $r in fn:doc("recipes.xml")//rcp:recipe[5]
  return local:recipes($r),""
)
```


Sequence Types

```
2 instance of xs:integer
2 instance of item()
2 instance of xs:integer?
() instance of empty()
() instance of xs:integer*
(1,2,3,4) instance of xs:integer*
(1,2,3,4) instance of xs:integer+
<foo/> instance of item()
<foo/> instance of node()
<foo/> instance of element()
<foo/> instance of element(foo)
<foo bar="baz"/> instance of element(foo)
<foo bar="baz"/>/@bar instance of attribute()
<foo bar="baz"/>/@bar instance of attribute(bar)
fn:doc("recipes.xml")//rcp:ingredient instance of element()+
fn:doc("recipes.xml")//rcp:ingredient
   instance of element(rcp:ingredient)+
```

An Untyped Function

```
declare function local:grade($g) {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B")  then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C")  then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D")  then 1.0
  else if ($g="D-") then 0.7 else 0
};
```


A Default Typed Function

```
declare function local:grade($g as item()*) as item()* {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B")  then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C")  then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D")  then 1.0
  else if ($g="D-") then 0.7 else 0
};
```


A Precisely Typed Function

```
declare function local:grade($g as xs:string) as xs:decimal {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B")  then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C")  then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D")  then 1.0
  else if ($g="D-") then 0.7 else 0
};
```


Another Typed Function

```
declare function local:grades($s as element(students))
        as attribute(grade)* {
  $s/student/results/result/@grade
};
```

Runtime Type Checks

- Type annotations are checked during runtime
- A *runtime type error* is provoked when
  - an actual argument value does not match the declared type
  - a function result value does not match the declared type
  - a valued assigned to a variable does not match the declared type

Built-In Functions Have Signatures

```
fn:contains($x as xs:string?, $y as xs:string?)
   as xs:boolean
```

```
op:union($x as node()*, $y as node()*) as node()*
```


XQueryX

```
for $t in fn:doc("recipes.xml")/rcp:collection/rcp:recipe/rcp:title
return $t
```

XML Databases

- How can XML and databases be merged?
- Several different approaches:
  - extract XML *views* of relations
  - use SQL to *generate* XML
  - *shred* XML into relational databases

The Student Database Again

Students(id,name,age)

100026	Joe Average	21
100078	Jack Doe	18

Majors(id,major)

100026	Biology
100078	Physics
100078	XML Science

Grades(id,course,grade)

100026	Math 101	C-
100026	Biology 101	C+
100026	Statistics 101	D
100078	Math 101	A+
100078	XML 101	A-
100078	Physics 101	B+
100078	XML 102	A

Automatic XML Views (1/2)

```
<Students>
  <record id="100026" name="Joe Average" age="21"/>
  <record id="100078" name="Jack Doe" age="18"/>
</Students>
```


Automatic XML Views (2/2)

```
<Students>
  <record>
    <id>100026</id>
    <name>Joe Average</name>
    <age>21</age>
  </record>
  <record>
    <id>100078</id>
    <name>Jack Doe</name>
    <age>18</age>
  </record>
</Students>
```


Programmable Views

```
xmlelement(name, "Students",
  select xmlelement(name,
                    "record",
                    xmlattributes(s.id, s.name, s.age))
  from Students
)
```

```
xmlelement(name, "Students",
  select xmlelement(name,
                    "record",
                    xmlforest(s.id, s.name, s.age))
  from Students
)
```

XML Shredding

- Each element type is represented by a relation
- Each element node is assigned a unique key in document order
- Each element node contains the key of its parent
- The possible attributes are represented as fields, where absent attributes have the `null` value
- Contents consisting of a single character data node is inlined as a field

From XQuery to SQL

- Any XML document can be faithfully represented
- This takes advantage of the existing database implementation
- Queries must now be phrased in ordinary SQL rather than XQuery
- But an automatic translation is possible

```
//rcp:ingredient[@name="butter"]/@amount
```

```
select ingredient.amount
from ingredient
where ingredient.name="butter"
```


Summary

- XML trees generalize relational tables
- XQuery similarly generalizes SQL
- XQuery and XSLT have roughly the same expressive power
- But they are suited for different application domains: *data-centric* vs. *document-centric*

Essential Online Resources

- http://www.w3.org/TR/xquery/
- http://www.galaxquery.org/
- http://www.w3.org/XML/Query/