XQuery Language

**Introduction to databases**

**CSCC43 Spring 2012**

Ryan Johnson

Thanks to Manos Papagelis, John Mylopoulos, Arnold Rosenbloom and Renee Miller for material in these slides

---

**Quick review of XPath**

**Strengths**
- Compact syntax
- Efficient XML tree traversal
- Predicates filter out nodes we don’t want

**Weaknesses**
- Declarative (no control flow)
- Most joins impossible (self-joins possible but ugly)
- Little/no ability to manipulate XML
- Can’t format results
- No way to specify input!

---

**Why might we manipulate XML?**

- Consider the XPath query
  - `<book-list>[book/author/last-name = 'Asimov']` => returns a list of complete book elements

- **How to turn that into the following?**

  ```xml
  <book-list>
    <book>
      <title>I, Robot</title>
      <publisher>Gnome Press</publisher>
    </book>
    ...
  </book-list>
  ```

  **XPath union operator isn’t enough**
  - Flat list of title and publisher elements
  => What if `<!ELEMENT book (title*, publisher*, ...)>`?

---

**How XQuery can be used?**

- Extracting information from a database for use in a web service or an application integration
- Generating reports on data stored in an XML database
- Transforming XML data to XHTML to be published on the Web
- Searching textual documents on the Web
- Splitting up a large XML document to multiple XML documents

---

**XPath can only return full (not “sparse”) subtrees**
**XQuery**

- Address most weaknesses with XPath
  - without adding too much complexity
- Primary features
  - access methods (read XML from file, etc.)
  - control flow: if/then/else, iteration
  - variables
  - functions (both user-defined and library flavors)
  - XML transformation: make data presentable
  - sorting, more powerful predicates, set operations...

*Expressiveness: XPath << SQL << XQuery*

**Key concepts of XQuery**

- Template of sorts: mixed output and logic
  - Statically create the overall structure
  - Embed logic to inject input data in right places
- All expressions return XML
  - Like in RA, outputs of one operation can be input to another
  - Returned value may be text, element, or node set
- “FLWOR” expressions
  - Allow iteration over node sets and other sequences
- Functions
  - Allow logic encapsulation, recursion

*NOTE: XQuery syntax bleeds back into XPath*

**Basic XQuery code format**

```xml
<title>Useful information about Tor:</title>
<books-by-tor> static template
{ //book[publisher='Tor']/title
  } interpreted code
</books-by-tor>
<authors-with-tor>
{ //book[publisher='Tor']/author/last-name}
</authors-with-tor>
```

*Oops! Author list has duplicates...*

**FLWOR (“flower”) expressions**

- XPath:
  ```xml
  //book[publisher='Tor'
  and author/last-name='Asimov'
  ]/*[self::author | self::title]
  ```
- FLWOR:
  ```xml
  for $b in //book
  let $a := $b/author
  where $b/publisher = 'Tor'
  and $a/last-name='Asimov'
  order by $b/title
  return <book>{$b/title,$a}</book>
  ```

*In what ways is the FLWOR superior to the XPath?*
Characteristics of a FLWOR

- **F(or)**
  - Iterate over each item in a sequence
  - Multiple sequences separated by commas

- **L(et)**
  - Declares a variable and assigns it a value
  - Multiple declarations separated by commas
  - Assigned at start of each iteration of the `for` above it

- **W(here), O(rder by)**
  - Stolen shamelessly from SQL...

- **R(eturn)**
  - The value produced by the current iteration
  - FLWOR is an expression, NOT a function call!
  
  => Result is a sequence of "returned" values

(for | let)+ where? order-by? return

Output behaviour of FLWOR

- In XPath, every node output at most once
  - Predicates just "mark" nodes which "pass"
  - All marked nodes output at end
  - Cartesian product impossible => most joins impossible

- In FLWOR, node output with every `return`
  - Every node in a node set bound to the loop variable
  - Emit any that make it past the `where` clause

- Distinction matters for nested loops!
  - Cartesian product: `for $x in //book, $y in //book...`

Sequences in XQuery

- **Most expressions return sequences of nodes**
  - LET $b = /bib/book ==> $b is a sequence
  - $b/@isbn ==> a sequence
  - $b/price ==> sequence of n prices
  - $b/price * 0.7 ==> sequence of n numbers

- **Sequence literals also allowed**
  - e.g. (1,2,3), also shortcut for ranges: (1 to 10)
  - empty sequence: ()

- **Sequences combine easily, flatten automatically**
  - (1, 2, (3, 4, 5), (), 6, 7) ==> (1, 2, 3, 4, 5, 6, 7)

If-then-else expressions

- **Syntax is very C-like:**
  - if ($expr) then $expr else $expr

- **BUT, like FLWOR, it is an expression!**
  - for $b in //book
    return
      if ($b/publisher = 'Tor')
        then <book>{$b/(title|author)}</book>
        else ()

Advanced predicates on node sets

• So far, two ways of predicing on node sets
  – Test for empty/non-empty
  – Iterate over their members and apply a predicate

• Two other techniques exist also
  – Explicit quantification
  – Single-object matching

Quantification

• XPath implicitly uses existential quantification
  – //genre[../author/last-name='Asimov']/@name
  => Names every genre containing at least one book by Asimov
  => Tests whether //author/last-name[. = 'Asimov'] is empty

• XQuery adds explicit control
  – Existential (exists): some $x$ in $y$ satisfies $expr$
  – Universal (forall): every $x$ in $y$ satisfies $expr$

• Examples
  – //genre[some $n$ in ../author/last-name satisfies $n$='Asimov']/@name
  – /transcript/semester[every $m$ in mark satisfies $m$ > 3.5]
  – /transcript/semester[some $m$ in mark satisfies $m$ < 2.0]

Comparisons

• General comparisons
  – Apply to sequences of objects (same as XPath)
  – Operators: = !< > <= >=
  – A op B is true <=⇒ $\exists x \in A$ and $\exists y \in B$ s.t. $x$ op $y$ is true

• Value comparison operators
  – Compare the values of two objects
  – Operators: eq, ne, lt, gt, le, ge
  – Error if both sides cannot be “atomized” to primitive types

• Node comparisons
  – Compare identity of two nodes based on position in document
  – Operators: is, <= (for preceding), => (for following)
  – “is” only true if both sides are actually the same node

“=” vs. “eq”

for $b1$ in //book, $b2$ in //book
where $b1/* = b2/* return <hit>{$b1,$b2}</hit>
=> Return every pair of books with any of: same author, same
genre, same publisher, same price...
=> Using “eq” gives error if count($b/*) != 1

//book[price gt 10]
=> Type error: price (untyped) != 10 (integer)
=> Use //book[xs:integer(price) gt 10]
=> “=” converts simple types automatically
Set operations

• XPath defines only union (“|”)
• XQuery adds union, intersect, except operators
  – stolen shamelessly from SQL
• Also, duplicate elimination: distinct-values()
• All based on node comparisons, not values
  – Attributes and children (recursively) must also match
  => “match” usually means “same node in the source doc”

Duplicate elimination

• XQuery often produces duplicates

//book[publisher='Tor']/author/last-name

• Solution: distinct-values()?
  – “Atomizes” inputs (elements become strings)
  => Good for ints, strings; useless for elements
  => distinct-values((1, 2, 3, 1, 2)) = (1, 2, 3)
• Solution: XPath’s index-of() function
  – index-of((x1,x2, ...), $y) returns position() of $x where $x=$y
  => index-of((a,b,c,b,a), a) = (1,5)
  => index-of ((15, 40, 25, 40, 10), 18) = ()
  => index-of ("a", "dog", "and", "a", "duck"), "a") = (1, 4)

User-defined functions

• Example
declare function count-nodes($e as element()) as integer
  { return type
    1 + sum( for $c in $e/*
      return count-nodes($c)
    )
  };
• Arguments can be typed
  – Parenthesis after type names
  – Cardinality controlled in the usual way: + ? *
  – Default type: item()*
• Function body is an expression to evaluate
  – Ironically, functions don’t use the return keyword!
  – Recursion allowed

Deduplication using index-of()

• Consider index-of($x, $y)[1]
  – Return the first (in document order) $z in $x satisfying $y=$z
• Solution: $x[position()=index-of($x, .)[1]]
  – Return every $z in $x which is the first in its set of matches
  – Abbreviated syntax: $x[index-of($x, .)[1]]
• WARNING: only works at document root!
  – /a/b[index-of(/a/b,.)[1]] fails in strange ways
  => Assign non-root sequences to a variable first
Node deduplication examples

- Author list example
  `<authors>{//book[publisher='Tor']/author}</authors>`

- One-off deduplication
  `<authors>{let $a := //book[publisher='Tor']/author
    return $a[index-of($a, .)]}</authors>`

- Even better: encapsulate it in a function!
  `declare function distinct-nodes($a) { $a[index-of($a, .)[1]] };`

- Can also define “duplicate” more narrowly
  `let $b := /book-list/book[publisher='Tor']
    return $b[index-of($b/author/last-name, ./author/last-name)]`
  - Returns one book for each author last name

Example: XML Doc

```xml
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
</bib>
```

Example: XML Doc (cont.)

```xml
<bib>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for DigitalTV</title>
    <editor><last>Gerbarg</last><first>Darcy</first><affiliation>CITT</affiliation></editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```
**FLWOR Expression Example**

**Query1:** Find all books titles published after 1995

FOR $x$ IN doc("bib.xml")/bib/book
WHERE $x$/year > 1995
RETURN $x$/title

**Result:**
- <title>TCP/IP Illustrated</title>
- <title>Advanced Programming the Unix environ...</title>
- <title>Data on the Web</title>
- <title>The Economics of Technology and ...</title>

**More complex FLWOR**

**Query2:** List all books of authors published by Morgan Kaufmann

FOR $a$ IN distinct-nodes(/)
    /bib/book[publisher="Morgan Kaufmann"]/author
RETURN <result>
    {$a}
    {FOR $t$ IN /bib/book[author=$a]/title
    RETURN $t}
</result>

**More complex FLWOR (cont.)**

**Query3:** Find books whose price is larger than average

LET $a$ = avg(/bib/book/price)
FOR $b$ in doc(/bib/book
WHERE $b$/price > $a
RETURN $b$

avg: aggregate function that returns the average

**GROUP BY and HAVING in XQuery**

**Query4:** Group by author name their first ten books, for authors having written more than ten books

FOR $a$ IN distinct-nodes(/book/author/lastname)
LET $books := /book[SOME $y$ IN author/lastname = $a$]
WHERE COUNT($books )>10
RETURN
    <result>
        {$a} { $books[1 to 10] }
    </result>
Joins in XQuery

**Query5**: For each book title list the prices offered by amazon and bn

```xml
<books-with-prices>
  {FOR $a IN doc("amazon.xml")/book, $b IN doc("bn.xml")/book
   WHERE $b/@isbn = $a/@isbn
   RETURN <book>
     { $a/title }
    <price-amazon>{ $a/price }</price-amazon>,
    <price-bn>{ $b/price }</price-bn>
  </book>
}
</books-with-prices>
```

Outer Joins in XQuery

**Query5**: For each book title list their prices offered by amazon and bn

```xml
<books-with-prices>
  {FOR $a IN doc("amazon.xml")/book
   RETURN <book>
     { $a/title }
    <price-amazon>{ $a/price }</price-amazon>,
    <price-bn>{ $b/price }</price-bn>
  }
}
</books-with-prices>
```

Full-outer Joins in XQuery

**Query5**: For each book title list their prices offered by amazon and bn

```xml
LET $allISBNs:= distinct-values (doc("amazon.xml")/book/@isbn union
doc("bn.xml")/book/@isbn)
RETURN <books-with-prices>
  {FOR $isbn IN $allISBNs
   RETURN <book>
     {FOR $a IN doc("amazon.xml")/book[@isbn=$isbn]
      RETURN <price-amazon>{ $a/price }</price-amazon>}
     {FOR $b IN doc("bn.xml")/book[@isbn=$isbn]
      RETURN <price-bn>{ $b/price }</price-bn>}
   </book>
  }
</books-with-prices>
```

If-Then-Else in XQuery

**Query5**: Make a list of holdings, ordered by title. For journals, include the editor, and for all other holdings, include the author

```xml
FOR $h IN //holding
RETURN <holding>{ $h/title}
  {IF ($h/@type = "Journal")
   THEN $h/editor
   ELSE $h/author}
</holding>
ORDERBY (title)
```
Existential Quantifiers in XQuery

**Query7**: Find titles of books in which both sailing and windsurfing are mentioned in the same paragraph

```xquery
FOR $b IN //book
WHERE SOME $p IN $b//para
    SATISFIES contains($p, "sailing")
    AND contains($p, "windsurfing")
RETURN $b/title
```

Universal Quantifiers in XQuery

**Query8**: Find titles of books in which sailing is mentioned in every paragraph

```xquery
FOR $b IN //book
WHERE EVERY $p IN $b//para
    SATISFIES contains($p, "sailing")
RETURN $b/title
```

Sorting in XQUERY

**Query9**: Find the publishers and the books they have published, order by publisher name and then by book price descending

```xquery
<publisher-list>
 FOR $p IN distinct-values(/publisher)
 RETURN <publisher>
  <name> $p/text() </name>,
  FOR $b IN //book[publisher = $p]
  RETURN <book>
    $b/title, $b/price
  </book> ORDERBY(price DESCENDING)
 </publisher>
ORDERBY(name)
</publisher-list>
```

User-specified Functions in XQuery

**Query10**: Find the maximum depth of the document named "partlist.xml"

```xquery
NAMESPACE xsd="http://www.w3.org/2001/XMLSchema
datatypes"
FUNCTION depth(ELEMENT $e) RETURNS xsd:integer {
  -- An empty element has depth 1
  -- Otherwise, add 1 to max depth of children
  IF empty($e/*) THEN 1
  ELSE max(depth($e/*)) + 1
}

depth(doc("partlist.xml"))
```