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!

We just spent the last couple of days becoming familiar with XPath, and its weaknesses... overall, it’s great for what it does, but very limited. It’s certainly far less capable than SQL, let alone a full programming language. The lack of loops, variables, and any form of aggregation is limiting, but one of the biggest problems is that the way xpath produces output.
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?
  
  ```
  <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*, ...)>>`?

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

The output of an xpath operation is a list of nodes, strings, and/or attributes (depending on the query), none of which are valid XML documents. As a simple example, suppose we wanted the title and publisher of all books by Isaac Asimov. We might issue something like the following (assuming xpath-2.0 syntax):

```
/book-list/book[author/last-name='Asimov']/(title|publisher)
```

It would return a list like the following:

```
```

In theory we could reconstruct the original title-publisher pairs, but it’s extra work and would fail if title or publisher were optional. The only alternative is to return entire book elements that meet the match condition, but that brings along unwanted extra information.

What we really want is a way to control the structure of output; given a set of matching elements we should be able to produce a new XML document that incorporates them. This is precisely what xquery was designed for.
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

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 define 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

XQuery is a form of static template language: code and output formatting can interleave freely, with code that returns snippets of templated XML embedded in a larger XML template. Unlike xpath, xquery always returns something that can become input for further processing, and user-defined functions (including recursion) allow flexible designs.
This is an example of a static template with some bits of live code embedded in strategic locations. Some header information is produced first, followed by a list of book titles; the xpath expression is not formatted further, so the returned nodes are simply inserted where the code that produced them used to be. A second section repeats the process, this time filling out an author list with author last names fetched by a second xpath expression.

A couple of things worth noting:

The author list probably has duplicates (authors who published multiple books), ideally we’d have a way to get rid of those (stay tuned!)

There is non-trivial redundancy here, left over from our xpath days: //book[publisher='Tor'] is used twice. It would be better to store the book list in a variable and process it twice, rather than running the xpath twice.
FLWOR ("flower") expressions

• XPath:
  //book[publisher='Tor'
       and author/last-name='Asimov'
   ]/*[self::author | self::title]

• FLWOR:
  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?_

The core of the xquery language is called a FLWOR expression (sometimes given as “FLOWR”). FLWOR brings five key features to the language:

Iteration over nodesets. Iterate over all elements in a variable or matched by an xpath expression

Variable declarations. Store intermediate results for later re-use, avoiding redundancy and improving performance

Selection. Allows more flexible types of predicates than those supported in xquery, particularly when comparing elements from nested loops

Ordering. Sorting capabilities like those we’ve come to expect from SQL

Output formatting. No need to return the raw output of an xpath; each item in an iteration can be formatted in near-arbitrary ways before returning it to its caller.

One really important thing about “return” in xquery: it is *not* like the return statement of a language like python or Java. In those languages, return ends the computation and returns a single value. In xquery, a return statement just adds a value to a list. That list (which may be empty) will be returned to the caller once the computation completes. The closest thing in python would be the yield statement, which returns a single value to the caller without ending the computation; Java has no such concept.
FLWOR expressions can be nested in two ways. The simple form of nesting lets you use F and L as many times as you want, in any order, but you only get to use W, O and R once.

The other way to nest FLWOR is to realize that they are expressions and can therefore be used anywhere an expression is allowed. In particular, you could have one FLWOR iterate over the result produced by another FLWOR. We’ll see examples of this a bit later.

**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...
- **Return**
  - The value produced by the current iteration
  - FLWOR is an expression, NOT a function call!
  => Result is a sequence of “returned” values

*Have as many F and L as you want, in any order*
We already talked a little bit about “return” in XQuery, and how it’s different than “return” in Python or Java. We should also point out that FLWOR returns things differently than XPath.

In XPath, every predicate can mark zero or more elements as valid, and the returned result is just the union of elements marked this way. There is no way to return an element twice, and there is no way to unmark an element after a predicate marks it as valid. There is no way to compute an intersection (nodes can’t be unmarked), and there is no way to produce a Cartesian product, which rules out joins as well.

In XQuery, every FLWOR can apply new filtering steps to refine the results of previous ones, and nothing stops the programmer from storing a result set in a variable and iterating over it as many times as they want, including with nested loops.

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)

The basic unit of work in xpath is a sequence: an ordered list of elements, possibly with duplicates. Xpath queries return sequences, as do FLOWR expressions, and sequence literals can be specified with a tuple syntax similar to the one used in python. Unlike python, however, sequences in xquery are always flat: putting two or more sequences together returns a new sequence that contains the elements of its inputs. This is true even if you try to make a sequence as an element inside another sequence (which can happen if a FLWOR returns sequences rather than elements).

Sequences may contain elements, attributes, or atomic values like strings and numbers. Sequences can even be mixed, containing elements of different types, though this is usually not a good idea (it’s much more useful to group mixed sequences into new xml nodes that can be easily processed later with additional xpath queries).
RETURN gotchas

- Must use a sequence to return multiple values
  - Broken: for ... return $x, $y
  => parses as (for ... return $x), $y
  - Correct: for ... return ($x, $y)

- Need { } when inside templates:
  - Broken: return <i>$title</i>
  - Correct: return <i>{$title}</i>

- Only need { } when inside templates
  - Broken: return (<i>{$title}</i>, {$rest})
  - Correct: return (<i>{$title}</i>, $rest)

What the slide says. These are easy to trip over, but just as easy to fix once you know what the problem is.
If-then-else expressions

- Syntax is very C-like:
  
  \[
  \text{if ($expr) then $expr else $expr}
  \]

- BUT, like FLWOR, it is an expression!

```xml
for $b in //book
  return
    if ($b/publisher = 'Tor')
      then <book>{$b/(title|author)}</book>
    else ()
```

xquery includes an if/then conditional syntax, like most languages. However, it’s not a statement but rather an expression. Instead of saying “if $condition is true, *do* x else *do* y” if/then in xquery says “if $condition is true, *return* x else *return* y” (where return has its usual xquery meaning). This is a lot like the ?: syntax in C or Java.
Advanced predicates on node sets

- So far, two ways of predicating 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 (∃): some $x$ in $y$ satisfies $expr$
  – Universal (∀): 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 $m < 2.0$]

Quantification, what joy. Remember from xpath, how any predicates comparing scalars to sequences return a new sequence containing only those elements that match the scalar comparison? This is “existential quantification” – any element matching is enough to make the expression true.

XQuery defaults to existential quantification as well, but also allows the user to explicitly request either existential or universal quantification. Universal quantification means that every element must match or else the expression is false.

Note that, either way, xquery follows Xpath in treating “empty” as “false” and “non-empty” as “true.”
Most comparisons are like the ones in any other language, but there are some special cases to be aware of.

First, XQuery introduces a second set of comparators that only operate on objects, and return an error if they encounter a sequence as either operand.

Second, XQuery gives a way to work with identities of elements, rather than their values. Identity is based on a node’s position in the original document. We can test whether two variables contain exactly the same element (the self axis in XPath), and also test whether an element comes before/after some other element (the preceding and following axes in XPath).
“"=" 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

A couple of examples to think about...
Set operations

• XPath defines only union ("|")
• XQuery adds union, intersect, except operators
  – stolen shamelessly from SQL
• Also, duplicate elimination: distinct-values()
• Compares nodes but returns values
  – Attributes and children (recursively) must also match
  => “match” usually means “same node in the source doc”
  – Returned output is sequence of strings

Unfortunately, the function distinct-values does exactly what its name suggests: it converts each sequence element into text and then deduplicates the resulting sequence of strings. This is not very useful in practice: think of all those <book> elements containing seemingly-identical <author><first-name>Isaac</first-name><last-name>Asimov</last-name></author> elements; if we select them with an xpath query and try to use them as input to a later operation, we either have to be content with working on strings from then on or live with the duplicates.

Later on, we’ll define a function that returns distinct members of any sequence (nodes, attributes, or atomic values) instead.
Now we’re getting to the good stuff: we can define our own functions to encapsulate common operations and allow recursion.

The implications are exciting, but the syntax itself is not complex, so there’s little more to say about it here. The function here defines a recursive node-counter: the count of a node is 1 + the counts of its children (leaf nodes therefore have count 1).

One annoying detail: unless you tell it otherwise, the xquery engine assumes your new function belongs in the ‘fn:’ namespace (since that’s where all functions live by default). However, the fn namespace is restricted, meaning that only built-in functions are normally allowed there. You can either declare your own namespace and put the function inside it, or just put it in namespace “local”.

User-defined functions

- Example
  
  declare function local:count-nodes($e as element()) as integer
  {
    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

- Now we’re getting to the good stuff: we can define our own functions to encapsulate common operations and allow recursion.

- The implications are exciting, but the syntax itself is not complex, so there’s little more to say about it here. The function here defines a recursive node-counter: the count of a node is 1 + the counts of its children (leaf nodes therefore have count 1).

- One annoying detail: unless you tell it otherwise, the xquery engine assumes your new function belongs in the ‘fn:’ namespace (since that’s where all functions live by default). However, the fn namespace is restricted, meaning that only built-in functions are normally allowed there. You can either declare your own namespace and put the function inside it, or just put it in namespace “local”
Duplicate elimination

- XQuery often produces duplicates
  \[\text{//book[publisher='Tor']/author/last-name}\]
- Solution: distinct-values()?
  - “Atomizes” inputs (elements become strings)
  \[\Rightarrow\text{Good for ints, strings; useless for elements}\]
  \[\Rightarrow\text{distinct-values((1, 2, 3, 1, 2)) = (1, 2, 3)}\]
- Solution: XPath’s index-of() function
  \[- \text{index-of}((x_1,x_2, \ldots), y) \text{ returns position()} \text{ of } x_i \text{ where } x_i=y\]
  \[\Rightarrow\text{index-of((a,b,c,b,a), a) = (1,5)}\]
  \[\Rightarrow\text{index-of ((15, 40, 25, 40, 10), 18) = ()}\]
  \[\Rightarrow\text{index-of ("a", "dog", "and", "a", "duck"), "a") = (1, 4)}\]

As promised, now that we have user-defined functions, our first order of business is to define a general deduplication routine.

The key is a built-in xquery function called index-of, which takes a sequence and an item, and returns all positions in the sequence that compare equal to the item. There could be zero, one, or many such matches.
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 reliable for document root!
  – `/a/b[index-of(/a/b,.)[1]]` fails in strange ways
  => ALWAYS assign non-root sequences to a variable first

Armed with index-of, all we need to do is arrange to call index-of on every element of a sequence, matching against that same sequence. Then, we only return the element if it is the first (or last) of its kind. Since only one element can be the first of its kind in a given sequence, the result is a sequence containing no duplicates.

PLEASE pay attention to that warning!!! Always always always assign a sequence to a variable before deduplicating it. If you don’t, this trick can fail in strange ways as queries become more complex, and it’s always a real pain to debug.
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, .)[1]]}
  </authors>`

- Even better: encapsulate it in a function!
  `declare function local:distinct($a) { $a[index-of($a, .)[1]] };`  
  `<authors>{
    local:distinct(/book[publisher='Tor']/author)
  }</authors>`

- Can also define “duplicate” more narrowly
  `let $b := /book-list/book[publisher='Tor']
  return $b[index-of($b/author/last-name, ./author/last-name)[1]]`
  - Returns one (arbitrary) book for each author last name
Example: XML Doc

<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.)

<bib>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last> <first>Serge</first></author>
    <author><last>Buneman</last> <first>Peter</first></author>
    <author><last>Suciu</last> <first>Dan</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>CITI</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>

For brevity, later slides assume bib.xml is root

More complex FLWOR

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

LET $b:=/bib/book$
FOR $a$ IN local:distinct($b/publisher="Morgan Kaufmann"/author$)
RETURN <result>
  {$a}$
  {FOR $t$ IN $b[author=$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 local:distinct(/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 Barnes&Noble

```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 Amazon sells, give the price B&N charges

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

**Query5**: For each book either Amazon or B&N sells, list both prices

```xquery
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

**Query6**: Make a list of holdings, ordered by title. For journals, include the editor, and for all other holdings, include the author

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

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

```
FOR $b IN //book
WHERE SOME $p IN $b//paragraphs SATISFIES contains($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

```
FOR $b IN //book
WHERE EVERY $p IN $b//paragraphs SATISFIES contains($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

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

User-specified Functions in XQuery

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

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

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