XML Query Languages

Yanlei Diao
University of Massachusetts Amherst
XML Path Language (XPath) 1.0

- XPath 1.0: navigates through the hierarchical structure of an XML document, applies selection conditions, and retrieves nodes
  - [http://www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)
- Input: a tree of nodes
  - Types of node: Document, Element, Attribute, Text…
- Output: an object of one of the four types
  - Node set, Boolean, Number, String
- XPath 1.0 later became a subset of XQuery 1.0 with small modifications
- This lecture focuses on XQuery 1.0
XQuery 1.0

- A programming language that can
  - retrieve fragments from XML
  - transform XML to XML data in arbitrary ways
  - [http://www.w3.org/TR/xquery/](http://www.w3.org/TR/xquery/)

- **Declarative querying + functional programming**
  - “Declarative” (the query part)
  - “Strongly typed”
  - “Functional” (almost) language

- Currently being widely used for
  - querying XML databases
  - transforming XML messages in Web services
  - routing XML messages in message brokering
  - creating logical views in data integration…
Query Language and Data Model

- A query language is “closed” w.r.t. its data model if input and output of a query conform to the model
  - SQL
    - Set of tuples in, set of tuples out
  - XPath 1.0
    - A tree of nodes (well-formed XML) in, an object (node set, boolean, number, or string) out
  - XQuery 1.0
    - Sequence of items in, sequence of items out

Compositionality of a language
- Output of Query 1 can be used as input to Query 2
XQuery Data Model

- **Instance of the data model**: a sequence composed of zero or more items
  - Empty sequence often considered as the “null value”
  - [http://www.w3.org/TR/xpath-datamodel/](http://www.w3.org/TR/xpath-datamodel/)

- **Item**: node or atomic value

- **Node**: document | element | attribute | text | namespace | processing instruction | comment

- **Atomic value**: an indivisible value, e.g., string, boolean, ID, IDREF, decimal, QName, URI, ...
Atomic values

- The values of the 19 atomic types available in XML Schema (http://www.w3.org/TR/xmlschema-2/)
  - E.g., xs:integer, xs:boolean, xs:date
- All the user defined derived atomic types
  - E.g., myNS:ShoeSize derived from xs:integer
- Untyped atomic values (non schema validated)
- Atomic values carry both type and value
  - (8, myNS:ShoeSize) is not the same as (8, xs:integer)
XML nodes

- **Types of nodes**: document | element | attribute | text | namespaces | PI | comment

- **Every node has a unique** node identifier
  - In an instance of data model, every node is unique, identical to itself but not to any other node

- **Nodes have children and an optional parent**
  - conceptual “tree”

- **Nodes are ordered based of the topological order in the tree (“document order”)**

- **Nodes can be compared directly by identity** or by document order, or by value after conversion from node to value
Sequences

- Can be heterogeneous (nodes and atomic values) 
  
  $(<a/>, 3)$

- Can contain duplicates (by value and by identity) 
  
  $(1, 1, 1)$

- Are not necessarily ordered in document order 

- Nested sequences are automatically flattened 
  
  $(1, 2, (3, 4)) = (1, 2, 3, 4)$

- Single items and singleton sequences are the same 
  
  $1 = (1)$
XQuery Expressions

XQuery Expr :=

Constants | Variable | FunctionCalls |
ComparisonExpr | ArithmeticExpr | LogicExpr |
PathExpr | FLWRExpr |
ConditionalExpr | QuantifiedExpr |
TypeSwitchExpr | InstanceofExpr | CastExpr |
UnionExpr | IntersectExceptExpr |
ConstructorExpr | ValidateExpr

Expressions can be nested with full generality!

Functional programming heritage.
Path Expressions

- A *path expression* consists of a sequence of steps.
- A *step* contains (axis node test predicate*)
- An *axis* controls the navigation direction in the tree:
  - **Forward axes**: ("attribute" | "child" | "descendant" | "descendant-or-self" | "self" | "following-sibling" | "following") ::
  - **Backward axes**: ("parent" | "ancestor" | "ancestor-or-self" | "preceding-sibling" | "preceding") ::
  - Given a *context node* (the evaluation context), an axis returns a sequence of nodes.
Path Expressions (cont’d)

- An node test filters the sequence of nodes that an axis selects.
  - **Name test**: e.g. publisher, * (wildcard for name test), myNS:publisher, *:publisher, myNS:*, *:*  
  - **Node kind test**: e.g. node(), text(), comment(),  
  - **Type test**: e.g., attribute(*, xs:integer)
Path Expressions (cont’d)

- A *predicate* further filters the nodes selected by the axis and retained by the node test.
  - axis :: node test [pred 1] [pred 2] … [pred n]

- **predicate** can be any expression, whose result is coerced to a boolean value, item retained if true.
  - On attribute: descendant::toy[attribute::color = “red”]
  - On text data: descendant::toy[child::text() = “pooh”]
  - On position: child::chapter[position() = 2]
  - On other elements: child::chapter[child::figure]
Semantics of Path Expressions

- Semantics of \textit{step1/step2}:
  - Evaluate \textit{step1} => sequence of nodes (o.w. runtime error)
  - For each node in this sequence:
    - Bind the \textit{evaluation context} to this node
    - Evaluate \textit{step2} with this binding => sequence of nodes
      (case 1), or sequence of atomic values (case 2)
  - Concatenate the partial sequences
  - Case 1:
    - \textit{Eliminate duplicates by node identity}
    - \textit{Sort by document order}

- Implicit iteration through steps of a path expression
  - http://www.w3.org/TR/xquery-semantics/
Non-Abbreviated Syntax

- Step in the non-abbreviated syntax:
  `axis "::" node test ("[" predicate "]")`*
- `doc("bibliography.xml")/child::bib`
- `$x/child::bib/child::book/attribute::year`
- `$x/descendant::author`
- `$x/child::chapter [child::figure]`
- `$x/child::book/child::chapter[fn:position() = 5]/child::section[fn:position() = 2]`
- `$x/child::para[attribute::type = "warning"] [fn:position() = 5]`
- `$x/child::para [fn:position() = 5] [attribute::type = "warning"]`
Abbreviated Syntax

- Axis can be missing
  - By default the child axis
    \[ \text{x/} \text{child::person} \rightarrow \text{x/person} \]

- Shorthand notations for common axes
  - Descendent:
    \[ \text{x/}\text{descendant-or-self::*/child::*}\text{chapter} \rightarrow \text{x//chapter} \]
  - Parent:
    \[ \text{x/}\text{parent::*} \rightarrow \text{x/..} \]
  - Attribute:
    \[ \text{x/}\text{attribute::*year} \rightarrow \text{x/@year} \]
  - Self:
    \[ \text{x/}\text{self::*} \rightarrow \text{x/} \]

- In a predicate, function \text{fn:position()} can be omitted
  - Positional predicate:
    \[ \text{[fn:position()} = 2] \rightarrow [2] \]
Examples of Abbreviated Syntax

- `doc("bibliography.xml")/bib`
- `$x/bib/book/year`
- `$x/chapter[figure]`
- `$x//author`
- `$x/book/chapter[5]/section[2]`
- `$x/para[@type = "warning"][5]`
- `$x/para[5][@type = "warning"]`

**Typical mistakes:**
- `$x/a/b[1]` means `$x/a/(b[1])` and not `($x/a/b)[1]`
- `$x//chapter[1]($x/descendant-or-self::*/child::*chapter[1])` is NOT the same as `$x/descendant::*chapter[1]`. 

16
Simple Iteration Expression

- **Syntax**: 
  ```
  for variable in expression1
  return expression2
  ```

- **Example**
  ```
  for $x$ in doc("bib.xml")/bib/book
  return $x/title
  ```

- **Semantics**:
  - bind the variable $x$ to each node returned by `expression1`
  - for each such binding, evaluate `expression2`
  - concatenate the resulting sequences
  - nested sequences in the “for” clause are automatically flattened
FLWOR expressions

- FLOWR is a high-level construct that
  - supports iteration and binding of variables to intermediate results
  - is useful for joins and restructuring data

- Syntax: For-Let-Where-Order by-Return

  for $x$ in expression1  /* similar to FROM in SQL */
  [let $y := expression2 ]  /* no analogy in SQL */
  [where expression3 ]  /* similar to WHERE in SQL */
  [order by expression4 (ascending | descending)? (empty (greatest | least))? ]  /* similar to ORDER-BY in SQL */

  return expression4  /* similar to SELECT in SQL */
Example FLOWR Expression

for $x$ in doc("bib.xml")/bib/book // iterate, bind each item to $x$
let $y := $x/author // no iteration, bind a sequence to $y$
where $x/title="XML" // filter each tuple ($x, $y)$
order by $x/@year descending // order tuples
return count($y) // one result per surviving tuple

- The for clause iterates over all books in an input document, binding $x$ to each book in turn.
- For each binding of $x$, the let clause binds $y$ to all authors of this book.
- The result of for and let clauses is a tuple stream in which each tuple contains a pair of bindings for $x$ and $y$, i.e. ($x, $y$).
- The where clause filters each tuple ($x, $y$) by checking predicates.
- The order by clause orders surviving tuples. Atomization is implicitly applied to the order by expression to yield an atomic value for sorting.
- The return clause returns the count of $y$ for each surviving tuple.
FOR versus LET

for $x$ in doc("bib.xml")/bib/book
return <result> $x$ </result>

Returns:

<result> <book>...</book></result>
<result> <book>...</book></result>
<result> <book>...</book></result>
...

let $x$ in document("bib.xml")/bib/book
return <result> $x$ </result>

Returns:

<result> <book>...</book></result>
<book>...</book>
<book>...</book>
...

</result>
Distinct values: the `fn:distinct-values` function eliminates duplicates in a sequence by value

- The `for` expression evaluates to a sequence of nodes
- `fn:distinct-values` converts it to a sequence of atomic values and removes duplicates

```xml
for $a in distinct-values(doc("bib.xml")/book/author)
return <author-name> {$a} </author-name>
```

versus

```xml
for $a in doc("bib.xml")/book/author
return $a
```
More Examples of WHERE

- **Selections**
  
  ```xml
  for $b$ in doc("bib.xml")/bib/book
  where $b$/publisher = "Addison Wesley" and
  $b$/@year = "1998"
  return $b/title
  ```

  ```xml
  for $b$ in doc("bib.xml")/bib/book
  where empty($b/author)
  return $b/title
  ```

  ```xml
  for $b$ in doc("bib.xml")/bib/book
  where count($b/author) = 1
  return $b/title
  ```

  Aggregates over a sequence: count, avg, sum, min, max
Value Comparison

- Value comparison “eq”: compares single values
- “eq” applies atomization (fn:data( )) to each operand
  - Given a sequence of nodes, fn:data( ) returns an atomic value for each node which consists of:
    - a string value, i.e., the concatenation of the string values of all its Text Node descendants in document order
    - a type, e.g., xdt:untypedAtomic
  - For each operand, “eq” uses the fn:data() result if it evaluates to a singleton sequence, o.w. runtime error.

✓ for $a$ in doc(“bib.xml”) / bib / book / author
   where $a$ eq “JefferyUllman”
   return $a/..$

✗ for $b$ in doc(“bib.xml”) / bib / book
   where $b$/author eq “JefferyUllman”
   return $b$/author
General Comparison

- General comparison operators (=, !=, <, >, <=, >=): existentially quantified comparisons, applied to operand sequences of any length
- Atomization (fn:data()) is applied to each operand to get a sequence of atomic values
- Comparison is true if one value from a sequence satisfies the comparison

```xml
for $b in doc("bib.xml")/bib/book
where $b/author = "JefferyUllman"
return $b/author
```

```xml
(1, 2) = (2, 3) ?
```
Node Comparison

- **Node comparison by *identity***

```xml
for $b in doc("bib.xml")/bib/book
where $b/author[last eq "Ullman"] is $b/author[first eq "Jeffery"]
return $b/title
```

- **Node comparison by *document order***

```xml
for $b in doc("bib.xml")/bib/book
where $b/author[. eq "JefferyUllman"] << $b/author[. eq "JenniferWidom"]
return $b/title
```
String Operations

- Functions for string matching
  ```
  fn:contains(xs:string, xs:string)
  fn:starts(ends)-with(xs:string, xs:string)
  fn:substring-before(after)(xs:string, xs:string)
  fn:matches($input as xs:string, $pattern as xs:string)
  ...
  ```

- Again, atomization (fn:data()) is applied to each function parameter to get an atomic value.

```xml
for $a in doc("bib.xml")//author
where contains($a, "Ullman")
return $a

<author>
  <name>Jeffery Ullman</name>
</author>
```
Order By

- Order by: applies *atomization* (fn:data( )) to the *sort by expression* to yield an *atomic value* for sorting
  - runtime error if the sort by expression evaluates to a sequence of size > 1

```
for $a in doc("bib.xml")//author
order by $a
return ($a, $a/../../../title)
```

```
for $b in doc("bib.xml")/bib/book
order by $b/author[1]
return <book> {$b/title} {$b/author} </book>
```
Joins in FOR, LET, WHERE

Joins

for $b$ in doc("bib.xml")//book,
    $p$ in doc("publishers.xml")//publisher
where $b$/publisher = $p$/name
return ($b$/title, $p$/name, $p$/address)

for $d$ in doc("depts.xml")/depts/deptno
let $e$ := doc("emps.xml")/emps/emp[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>
Node Constructors

- **Node constructor**: constructs new nodes
  - elements, attributes, text, documents, processing instructions, comments
  - element constructor is the most commonly used

- **Direct element constructor**: a form of element constructor in which the element name is a constant

```xml
<book isbn="isbn-0060229357">
  <title>Harold and the Purple Crayon</title>
  <author>  <first>Crockett</first>
              <last>Johnson</last>
  </author>
</book>
```
Element Construction in RETURN

- **Literal versus Evaluated Element Content**
  
  ```
  for $p$ in doc("report.xml")//person
  return ( <result>
    literal text content /* literal */
  </result> ,
  <result>
    { $person/name } /* {evaluated content} */
  </result> ,
  <result>
    { $person/name/text()}
  </result> ,
  <result>
    Yesterday {$person/name} visited us.
  </result> )
  ```

  - Braces 
    "{}" are used to delimit *enclosed expressions*, which will be evaluated and replaced by their value.
  
  - If return contains more than 1 item, use “()” and the *comma operator*. 

More on Element Construction

- The start tag of an element may contain attributes. The value of an attribute can be:
  - a string of characters enclosed in quotes, e.g.,
    `<book year="2005">`
  - evaluated using enclosed expressions specified in `{ }`
    ```
    <bib>
    { for $b in doc("bib.xml")/bib/book
      where $b/publisher = "Addison-Wesley"
      and $b/@year > 1991
      return <book year="{$b/@year }">{ $b/title } </book>
    }
    </bib>
    ```

- The largest “wrapping” tag creates well-formed XML.
Nested FLWOR

<authorlist>
{
  for $a in distinct-values(doc("bib.xml")/book/author)
  order by $a
  return
    <author>
      <name> {$a} </name>
      <books>
      {
        for $b in doc("bib.xml")/book[author = $a]
        order by $b/title
        return $b/title
      }
      </books>
    </author>
}  
</authlist>

The nested FLOWR effectively implements “group books by author”. No Group By in XQuery!
If-Then-Else Expressions

- For each book that has at least one author, list the title and first two authors, and an empty "et-al" element if the book has additional authors.

```xml
<bib>
{ for $b in doc("bib.xml")//book
  where count($b/author) > 0
  return
    <book> { $b/title }
    { for $a in $b/author[position()<=2]
      return $a
    }
    { if (count($b/author) > 2)
      then <et-al/> else ()
    }
  </book> } }
</bib>
```
XQuery Implementation

- **Open Source**
  - Saxon  (Michael Kay)
  - Galax  (AT&T, Mary Fernandez)

- **Commercial**
  - IBM, Microsoft, Oracle (with DB products)
  - BEA System (WebLogic Integration)
  - Some freelancers

- Visit:  http://www.w3c.org/xquery
Questions