7 Querying XML

- How to access data sources as XML?
  - Candidate Rec, November 2005
  - joint work by XML Query and XSL Working Groups
  - with XPath 2.0 and XSLT 2.0
  - influenced by many research groups and query languages
  - Quirk: XPath, XQL, XQL-QL, SQL, OQL, Lorel, ...
  - Goal: a query language for any XML-represented data: both documents and databases

SDPL 2006 7: Querying XML with XQuery 3

Functional Requirements

- Support operations (selection, projection, aggregation, sorting, etc.) on all data types:
  - Choose parts of data based on content or structure
  - Also operations on document hierarchy and order
- Structural preservation and transformation:
  - Preserve the relative hierarchy and sequence of input document structures in the query results
  - Transform XML structures and create new XML structures
- Combining and joining:
  - Combine related information from different parts of a given document or from multiple documents

SDPL 2006 7: Querying XML with XQuery 2

Functional Requirements

- Closure property:
  - The result of an XML query is also XML (usually not a valid document, but a well-formed document fragment)
  - results can be used as an input to another query
- Cross-references:
  - ability to traverse intra- and inter-document references
- Extensibility:
  - The query language should support the use of externally defined functions on all data types of the data model

SDPL 2006 7: Querying XML with XQuery 3

Example Query

```
xquery version "1.0";
<cheapBooks>
  <Title>Cheap Books</Title>
  <book price="20.50">
    <title>Designing Internet applications</title>
    <author>Michael Laverentz</author>
    <publisher>Prentice Hall</publisher>
    <year>1998</year>
  </book>
  <book price="26.50">
    <title>Computing with Logic</title>
    <author>David Maier</author>
    <publisher>Benjamin Cummings</publisher>
    <year>1999</year>
  </book>
</cheapBooks>
```

A possible result

```
<cheapBooks>
  <Title>Cheap Books</Title>
  <book price="20.50">
    <title>Designing Internet applications</title>
    <author>Michael Laverentz</author>
    <publisher>Prentice Hall</publisher>
    <year>1998</year>
  </book>
</cheapBooks>
```

SDPL 2006 7: Querying XML with XQuery 1

XQuery: Basics

- A query is an expression
- Expressions operate on, and return sequences of
  - atomic values (of XML Schema simple types) and
  - nodes (of 6 node types = XPath 1.0 – NS nodes)
  - an item = a singleton sequence
  - sequences are flat: no sequences as items
    - (1, 2, 3, 4, 5) = (1, 2, 3, 4, 5)
  - sequences are ordered, and can contain duplicates
- Unlimited combination of expressions, often with automatic type conversions (e.g. for arithmetics)

SDPL 2006 7: Querying XML with XQuery 7

XQuery: Accessing Documents

- XQuery operates on nodes accessible by input functions
  - fn:doc("URI")
    - document-node of the XML document available at URI
  - same as document("URI") in XSLT 1.0
  - fn:collection("URI")
    - sequence of nodes from URI
  - predeclared prefix for the default function namespace:
    fn=http://www.w3.org/2005/04/xpath-functions

SDPL 2006 7: Querying XML with XQuery 8
Central XQuery Expressions

- Path expressions
- Sequence expressions
- Comparison operators
- Element constructors (= XSLT templates)
- FLWOR expressions
  
- "(flower", for-let-where-order-by-return)
- Quantified expressions
  
some/every $x ... in ... satisfies ...

- and others in examples ...

Path Expressions

- Similar to XPath:

  - arbitrary (node-sequence-valued) expressions as steps
  
-  6 of 13 XPath axes: child, descendant, attribute, self, descendant-or-self, parent

  - others (except namespace) available if an optional Full Axis Feature supported

  - with document-order operators (<<, >>) sufficient for expressing queries (→ Exercises)

  - produce an ordered sequence of nodes in document order, without duplicates

Sequence Expressions

- Constant sequences constructed by listing values
  
- comma (,) used as catenation operator

  (1, 2, 3, 1)

- Also dynamically generated:

  (start, end)

- Shorthands for numeric sequences:

  1 to 4
  
- 4 to 1
  
- fn:reverse(1 to 4) → (4, 3, 2, 1)

Comparison Operators

- General comparisons bw sequences:

  =, !=, <, <=, >, >=

  - existential semantics: true iff some pair of values from operand sequences satisfy the condition

    (1, 2) = (2, 3) ; (2, 3) = (3, 4) ; (1, 2) != (3,4)

    - Same as in XPath 1.0:

      //book[@id = "Aho"]

        - books where some author is Aho

- Value comparisons bw single values:

  eq, ne, lt, le, gt, ge

  1 eq 3 is 2; 10 is 100: $books[price le 100]

Set Operations on Node (!) Sequences

- Assume variable bindings:

  $s1 = ...

  $s2 = ...

  Then:

  $s1 union $s2 = ...

  $s1 intersect $s2 = ...

  $s1 except $s2 = ... based on node idenity (node, is node)

Node Comparisons

- To compare single nodes,

  - for identity:

    $book/chap[@id="ch1"] is $book/chap[1]

    true iff the chapter with id="ch1" is indeed the first

  - for document order: << and >>


    true iff the chapter with id="ch2" appears after


Filter Expressions

- Location steps can be filtered by predicates:

  $book/chap[*] apply fn:last()//@title

  - the title of the last chapter of appendix, whichever is last

  - Other sequences, too:

    (1, 2, 3, 4) → (5, 10, 20, 20)

    - (.) generalized from XPath 1.0 shorthand for

      self::node() into the context item

Element Constructors

- Similar to XSLT templates:

  - start and end tag enclosing the content

  - literal fragments written directly, expressions enclosed in braces { and }

  - of XSLT 1.0 attribute value templates

  - often used inside another expression that binds variables used in the element constructor

    - (There is no 'current node' in XQuery)

    - See next
Example

- An `emp` element with an `empid` attribute and child elements `name` and `job`, from values in variables $\text{id}$, $\text{n}$, and $\text{j}$:
  
  `<emp empid="\{\$id\}"`
  
  `<name>{$n}</name>`
  
  `<job>{$j}</job>`
  
  `</emp>`

Also with computed names:

- element ("emp") {
  attribute ("empid"){$id},
  "name" {$n} </name>,
  "job" {$j} </job> }

Example: let clauses

```
let

– for $\text{i}$ in (1,2),
  $\text{j}$ in (1 to $\text{i}$)
return <tuple>
  <i>{$i}</i> <j>{$j}</j></tuple>
```

Result:

```
<tuple><i>1</i><j>1</j></tuple>
<tuple><i>1</i><j>2</j></tuple>
<tuple><i>2</i><j>2</j></tuple>
```

Example: for clause

```
for $\text{i}$ in (1,2),
  $\text{j}$ in (1 to $\text{i}$)
return <tuple>
  <i>{$i}</i> <j>{$j}</j></tuple>
```

Result:

```
<tuple><i>1</i><j>1</j></tuple>
<tuple><i>1</i><j>2</j></tuple>
<tuple><i>2</i><j>2</j></tuple>
```

Flow of data in a FLWOR expression

FOR/LET Clauses → Ordered list of tuples of bound variables
WHERE Clause → Pruned list of tuples of bound variables
RETURN Clause → Instance of XML Query data model

Flow of data in a FLWOR expression

Let clauses

- `let` also binds variables to expressions
  - each variable gets the entire sequence as its value (without iterating over the items of the sequence)
  - results in binding a single sequence for each variable

```
let $\text{s} := \text{<one/>}, \text{<two/>}, \text{<three/>}
return <out> ($\text{s} )</out>
```

Result:

```
<out>
  <one/>
  <two/>
  <three/>
</out>
```

for/let clauses

- A FLWOR expr may contain several `for`s and `lets`
  - each may refer to variables bound in previous clauses
  - the result of the `for/let` sequence:
    - an ordered list of tuples (monikko) of bound variables
    - number of tuples = product of the cardinalities of the node-lists returned by the expressions in the `for` clauses

```
for $\text{s}$ in <one/> , <two/> , <three/>
return <out> ($\text{s} )</out>
```

```
for $\text{s}$ in <one/> , <two/> , <three/>
return <out> ($\text{s} )</out>
```

```
for $\text{s}$ in <one/> , <two/> , <three/>
return <out> ($\text{s} )</out>
```

for clauses

- `for $\text{v}_1$ in Exp$_1$,
  $\text{v}_2$ in Exp$_2$, ...
  where
  assoc each variable $\text{v}_i$ with expression Exp$_i$
  (e.g. a path expression)
- Result: list of tuples, each containing a binding for each of the variables
- can be thought of as loops iterating over the items returned by respective expressions

```
for $\text{v}_1$ in Exp$_1$,
  $\text{v}_2$ in Exp$_2$, ...
  where
  assoc $\text{v}_1$ with Exp$_1$
  assoc $\text{v}_2$ with Exp$_2$
  ... ...
```

FLWOR ("flower") Expressions

- Constructed from `for`, `let`, `where`, `order by` and `return` clauses (~SQL `select-from-where`)
- Syntax: `(ForClause | LetClause | WhereClause)

  WhereClause?
  OrderByClause?
  "return" Expr
- FLWOR binds variables to values, and uses these bindings to construct a result (an ordered sequence of nodes)
where clause

- binding tuples generated by for and let clauses are filtered by an optional where clause
  - tuples with a true condition are used to instantiate the return clause
- the where clause may contain several predicates connected by and, or, and fn:not ()
  - usually refer to the bound variables
- sequences treated as Booleans (similarly to node-sets in XPath 1.0): empty ~ false, non-empty ~ true

Example: for + return

```xml
for $i in (1,2), $j in (1 to $i)
return <tuple>
  <i>{$i}</i> <j>{$j}</j></tuple>
```

Result:

```xml
<tuple><i>1</i><j>1</j></tuple> <tuple><i>2</i><j>1</j></tuple> <tuple><i>2</i><j>2</j></tuple>
```

Examples (modified from "XML Query Use Cases")

- Assume: a document named "bib.xml" containing a list of books:

```xml
<book><title><author><publisher><year><price></book>
```

List Morgan Kaufmann book titles since 1998

```xml
<recent-MK-books>
  for $b in doc("bib.xml")/book
  where $b/publisher = "Morgan Kaufmann"
  and $b/year >= 1998
  return <book year="{$b/year}">
    ($b/title)
  </book>
</recent-MK-books>
```

Result could be...

```xml
<recent-MK-books>
  <book year="1999">
    <title>TCP/IP Illustrated</title>
  </book>
  <book year="2000">
    <title>Advanced Programming in the Unix environment</title>
  </book>
</recent-MK-books>
```

Publishers with avg price of their books:

```xml
for $p in fn:distinct-values(
  fn:doc("bib.xml")//book/publisher
) let $a := avg( doc("bib.xml")//book[
  publisher = $p]/price )
return <publisher>
  <name>($p)</name>
  <avgprice>{$a}</avgprice>
</publisher>
```
Invert the book-list structure

```xml
<author_list>
  { $a : group books by authors }
  for $a in distinct-values(
    doc("bib.xml")//author )
  return $a
</author_list>
```

List of publishers alphabetically, and their books in descending order of price

```xml
for $p in distinct-values(
  doc("bib.xml")//publisher )
order by $p
return $p
</publisher>
```

Queries on Document Order

- `<<` and `>>`:
  - `x << y` is true if node `x` precedes node `y` in document order; `y >> x` similarly
- Consider a surgical report with
  - procedure elements containing
    - incision sub-elements
- Return a "critical sequence" of contents between the first and the second incisions of the first procedure

```xml
let $n/* = doc("report.xml")
for $b in $n/*: incision
if $b[1]/incision
let $p = doc("report.xml")//procedure[1]
for $n in $p/node()
  where $n >> ($p//incision)[1] and $n << ($p//incision)[2]
  return $n
</critical_sequence>
```

User-defined functions: Example

```xml
declare function xs:precedes($a as node(),
  $b as node()) as xs:boolean
  { $a << $b and ( ($a is no ancestor of $b:) or
    empty($a//node()) intersect $b ) };
```

User-defined functions: Example

- Now, "critical sequence" without ancestors of incision:
  ```xml
  let $sp = doc("report.xml")//procedure[1]
  for $n in $sp/node()
    where $n >> ($sp//incision)[1] and
      local:precedes($n, ($sp//incision)[2])
    return $n
  </critical_sequence>
  ```

Recursive Transformations

- Example: "Table-of-contents" for nested sections
  - NB if-then-else (in ordinary expressions, too)
  ```xml
  declare namespace my=http://my.name.org;
  declare function my:toc($n as element() ) as element()
  { if (name($n)="sect")
    then <sect ve>
      for $c in $n/* return my:toc($c )
    </sect>
  else if (name($n)="title") then $n
  else { check child elements, if any ;}
      for $c in $n/* return my:toc($c) ;
  ```

Querying relational data

- Lots of data is stored in relational databases
- Should be able to access also them
- Example: Tables for Parts and Suppliers
  - P (pno, descrip) : part numbers and descriptions
  - S (sno, sname) : supplier numbers and names
  - SP (sno, pno, price) : who supplies which parts and for what price?
Possible XML representation of relations

Grouping

- Many queries involve grouping data and applying aggregation function like `count` or `avg` to each group
- in SQL: `GROUP BY` and `HAVING` clauses
- Example: Find the part number and average price for parts with at least 3 suppliers

Grouping: SQL

```
SELECT pno, avg(price) AS avgprice
FROM sp
GROUP BY pno
HAVING count(*) >= 3
ORDER BY pno;
```

Grouping: XQuery

```
for $p in distinct-values(
doc("sp.xml")//p_tuple)
let $sp := doc("sp.xml")//p_tuple[pno=$p]
where count($sp) >= 3
order by $p
return
  <well_supplied_item>
    <pno>{$p/pno}</pno>,
    <avgprice>{$avg($sp/price)}</avgprice>
  </well_supplied_item>
```

Joins

- Example: Return a “flat” list of supplier names and their part descriptions, in alphabetic order

```
for $sp in doc("sp.xml")//p_tuple,
$sp in doc("s.xml")//s_tuple[sno = $sp/sno],
$p in doc("p.xml")//p_tuple[pno = $sp/pno],
order by $s/p/desc, $s/sname
return <sp_pair>(
  $s/sname ,
  $sp/desc
)</sp_pair>
```

XQuery vs. XSLT 1.0

- Could we express XQuery queries with XSLT?
  - In principle yes, always. (but could be tedious)
- Ex: Partial XSLT simulation of FLWOR expressions
- XQuery: `for $x in Expr... rest of query can be expressed in XSLT as:
  ```<xsl:for-each select="Expr">
    <xsl:variable name="x" select="/.
    ... translation of the rest of the query
  </xsl:for-each>```

XQuery vs. XSLT 1.0

```
XQuery: let $y := Expr...
  corresponds directly to:
  <xsl:variable name="y" select="Expr" />
and where Condition... rest
  → <xsl:if test="Condition">
    ... translation of the rest
  </xsl:if>
```

Selecting in SQL vs. XQuery

- SQL: `SELECT pno
  FROM p
  WHERE descrip LIKE 'Gear%'
  ORDER BY pno;`

- XQuery:
  ```
  for $p in doc("p.xml")//p_tuple
  where starts-with($p/desc, "Gear")
  order by $p/pno
  return $p/pno
  ```
XQuery vs. XSLT 1.0

XQuery:  
```
return ElemConstructor
```
can be simulated with a corresponding XSLT template:
- static fragments as such
- enclosed expressions in element content, e.g. $(s/sname)
```
<xsl:copy-of select="$s/sname" />
```

XSLT for XQuery FLWOR Expressions

- Only two things, roughly, can be done with XSLT 1.0
  - result tree fragments produced by templates:
    - with `<xsl:copy-of select="$X" />`
    - and conversion to a string
    - with `<xsl:value-of select="$X" />`
  - Not possible to apply other operations to results (like, e.g., sorting in XQuery):
    - for $y in <a><key>($x/code)</key></a>, ...
    - order by $y/key

XQuery vs. XSLT 1.0: Example

```
XQuery: for $b in doc("bib.xml")//book
  where $b/publ = "MK" and $b/year > 1998
  return <book year="($b/year)">($b/title) </book>
```

```
<xsl:template match="/"/>
<xsl:for-each select="document(‘bib.xml’)//book">
<xsl:variable name="b" select="." />
<xsl:if test="$b/publ = ‘MK’ and $b/year > 1998’">
  <book year="($b/year)">
    <xsl:copy-of select="$b/title" />
  </book>
</xsl:if>
</xsl:for-each>
</xsl:template>
```

XQuery: Summary

- W3C XML query language (draft), also capable of general XML processing
- Recommendation expected within one year(?)
- Vendor support??
  - http://www.w3.org/XML/Query
  - mentions about 50 prototypes or products (2004 - 20; 2005 - 40)
- Future?? Interesting confluence of document and database research, and highly potential for XML-based data integration