9 Querying XML Data and Documents

**XQuery, W3C XML Query Language**
- "work in progress", Working Draft 30 April 2002
- joint work by XML Query and XSL Working Groups
  - together with XPath 2.0 (← XPath 1.0 and XQuery)
  - influenced by many research groups and query languages
    - Quilt, XPath, XQL, XML-QL, SQL, OQL, Lorel, ...
  - Goal: a query language applicable to any XML-represented data: both documents and databases

**Functional Requirements**

1. 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

2. 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

3. Combining and joining:
   - Combine related information from different parts of a given document or from multiple documents

**XQuery in a Nutshell**

- Functional expression language
- Strongly-typed: (XML Schema) types may be assigned to expressions statically
- Includes XPath 2.0 (says Draft, but not all XPath axes included)
  - XQuery 1.0 and XPath 2.0 share extensive functionality:
    - XQuery 1.0 and XPath 2.0 Functions and Operators, WD 30/4/2002
- Roughly: XQuery = XPath' + XSLT' + SQL'

**XQuery: Basics**

- A query is represented as an expression
- Expressions operate on and return sequences of
  - atomic values (of XML Schema simple types) and
  - nodes (same 7 node types as in XPath/XSLT)
- an item = a singleton sequence
- sequences are flat: no sequences as items
  - e.g. (1, 2, 3), (1, 1) ≠ (1, 2, 3, 1)
- sequences are ordered, and can contain duplicates
  - Unlimited combination of expressions, often with automatic type conversions (e.g. for arithmetics)
Some Central XQuery Expressions

- path expressions
- creation and combination of sequences (union, intersect, except)
- element constructors (= XSLT templates)
- FLWR ("flower": for-let-where-return) expressions
- sort expressions
- quantified expressions (some/every... satisfies ...) testing and modifying datatypes; validation

Path Expressions

- Extend the syntax of XPath: [Expr]/.../Expr – arbitrary (node-sequence-valued) expressions as steps
- only 6 (of 13) axes: child, descendant, attribute, self, descendant-or-self, parent
- combined with order comparison operators (precedes, follows) sufficient for queries (?)
- produce ordered sequence of nodes in document order, without duplicate nodes
- dereferencing for ID/IDREF values: Figure(s) referenced by refid attribute of the first figref child: figref[1]/@refid => figure

Element Constructors

- Similar to XSLT templates:
  - start and end tag enclosing the content
  - literal fragments written directly, expressions enclosed in braces { and }
- often used inside another expression that binds variables used in the element constructor
  - See next

Example

- Generate an emp element containing an empid attribute and nested name and job elements; values of the attribute and nested elements given in variables $id, $n, and $j:
  <emp empid="{$id}"
    <name>{$n} </name>
    <job>{$j} </job>
  </emp>

FLWR ("flower") Expressions

- Constructed from for, let, where and return clauses (~ SQL select-from-where)
- Form: (forClause | LetClause)* WhereClause? return* Expr
- a FLWR expression binds values to one or more variables, and uses these variables to construct a result (in general, an ordered sequence of nodes)

Flow of data in a FLWR expression
for clauses

- for $V_i$ in Exp_i (...) associates each variable $V_i$ with an expression Exp_i that returns a list of items (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

let clauses

- let also binds one or more variables to one or more expressions
  - each variable to the entire value of its expression (without iterating through items of the sequence)
  - results in binding a single sequence for each variable
- Compare:
  - for $x$ in /library/book -> many bindings (books)
  - let $x := /library/book -> single binding (to a sequence of books)

for/let clauses

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

for/let clauses - Example 1

for $i$ in (1,2), $j$ in (3,4)
return <tuple><i>{$i}</i> <j>{$j}</j></tuple>

Result:
<tuple><i>1</i> <j>3</j></tuple>
<tuple><i>1</i> <j>4</j></tuple>
<tuple><i>2</i> <j>3</j></tuple>
<tuple><i>2</i> <j>4</j></tuple>

for/let clauses - Example 2

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

Result:
<out>
  <one/>
  <two/>
  <three/>
</out>

where clause

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

- Variables bound by a `for` clause represent a single item
  - scalar predicates, e.g. `$p/color = "Red"`
- Variables bound by a `let` clause may represent lists of nodes
  - list-oriented predicates, e.g. `avg($p/price) > 100`
  - a number of aggregation functions available:
    - `avg()`, `sum()`, `count()`, `max()`, `min()`

(also in XPath 1.0)

**return clause**

- The return clause generates the output of the FLWR expression
- instantiated once for each binding tuple
- often contains element constructors, references to bound variables, and nested subexpressions

-- Examples (from "XML Query Use Cases")

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

  ```xml
  <book>
    <title>TCP/IP Illustrated</title>
    <author>
      <name>Andrew S. Tanenbaum</name>
    </author>
    <publisher>Morgan Kaufmann</publisher>
    <year>1998</year>
    <price>49.95</price>
  </book>
  ...
  ```

  <$b/title>
  </book>
`</recent-MK-books>

- List each publisher with the average price of its books

  ```xml
  <publisher>...
    <name>...
    <avgprice>...
    </publisher>
  </publisher>
  ```

- 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>
  ```
Invert the book-list structure

```xml
<author_list>
  { -- group books by authors -- }
  for $a in distinct-values(
    document("bib.xml")//author )
  return
    <author>
      <name> {$a/text()} </name>
      for $b in document("bib.xml")//book[
        author = $a]
      return $b/title
    </author>
} </author_list>
```

Make an alphabetic list of publishers; for each publisher, list books (title & price) in descending order of price

```xml
for $p in distinct-values(
  document("bib.xml")//publisher )
return
  <publisher>
    <name> {$p/text()} </name>
    { for $b in document("bib.xml")//book[
      publisher = $p]
    return
      <book> {$b/title , $b/price} </book>
    sortby (price descending)
      sortby (name)
  } </publisher>
```

Queries on Document Order

- Operators `precedes` and `follows` to express conditions on document order
- The next example involves a surgical report with elements `procedure`, `incision` and `anesthesia`
- Return a "critical sequence" that contains all contents between the 1st and 2nd incisions of the 1st procedure

Computing a "critical sequence"

```xml```
<critical_sequence> {
  let $p := //procedure[1]
  for $n in $p/node() where $n follows ($p//incision)[1] and $n precedes ($p//incision)[2]
  return $n
} </critical_sequence>
```

Filtering

- Function `filter(Expr)` in XQuery core function library
- Returns shallow copies of the nodes selected by `Expr`
- Mutual order and hierarchy btw nodes the same as in input sequence

Hierarchical Filtering

- Only selected nodes remain; other levels are collapsed
Table of contents for "cookbook.xml", consisting of section titles (at any level of nesting)

```
let $b := document("cookbook.xml")
return
  <toc> {
    filter($b//section |
      section/title |
      section/title/text() )
  } </toc>
```

Querying relational data

- Lots of data is stored in relational databases
- should be able to access also this data
- Example: suppliers and parts
  - Table S: supplier numbers (sno) and names (sname)
  - Table P: part numbers (pno) and descriptions (descrip)
  - Table SP: relationships between suppliers and the parts they supply, including the price (price) of each part from each supplier

A possible XML form for relations

<table>
<thead>
<tr>
<th>Relational data</th>
<th>XML representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>&lt;s snno snname&gt;</td>
</tr>
<tr>
<td>P</td>
<td>&lt;p pno descrip&gt;</td>
</tr>
<tr>
<td>SP</td>
<td>&lt;sp sno pno price&gt;</td>
</tr>
</tbody>
</table>

SQL vs. XQuery

- SQL: `SELECT pno` `FROM p` `WHERE descrip LIKE 'Gear%'` `ORDER BY pno;`
- XQuery: `for $p in document("p.xml")//p_tuple` `where starts-with($p/descrip, "Gear")` `return $p/pno` `sortby(.);`

Grouping

- Many relational queries involve forming data into groups and applying some aggregation function such as `count` or `avg` to each group
- in SQL: `GROUP BY` and `HAVING` clauses
- Example: Find the part number and average price for parts that have 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 $pn in distinct-values(document("sp.xml")//pno)
let $sp := document("sp.xml")//sp_tuple[pno = $pn]
where count($sp) >= 3
return <well_supplied_item>
  {$pn,
   <avgprice>{avg($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 document("sp.xml")//sp_tuple,
  $p in document("p.xml")//p_tuple[pno = $sp/pno],
  $s in document("s.xml")//s_tuple[sno = $sp/sno]
return <sp_pair>{$s/sname, $p/descrip}</sp_pair>
```

XQuery vs. XSLT

- Could we express XQuery queries with XSLT? In principle yes, always. (but could be tedious)
- Ex: Partial XSLT simulation of FLWR expressions

XQuery: for $x in Expr …
can be expressed in XSLT as:
```
<xsl:for-each select="Expr">
  <xsl:variable name="x" select="."/>
  ...
</xsl:for-each>
```

XQuery vs. XSLT: Example

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

- NB: The sketched simulation is not complete:
  - Only two things can be done in XSLT for result tree fragments produced by templates:
    - insertion in result tree, and conversion to a string (with xsl:value-of)
  - Not possible to apply further operations to results (like e.g., sorting in XQuery)

Summary

- XQuery
  - W3C working draft of an XML query language
  - vendor support??
    » http://www.w3.org/XML/Query mentions about 20 prototype implementations or products that support (or will support) some version of XQuery
  - future??
  - promising confluence of document and database research