Joint work by XML Query and XSL

Choose data based on content or structure

should support externally defined functions on all data

queries can be combined without limit

Transform XML structures, and create new

Operate on document hierarchy and order

Combine data from different parts of a document, or from multiple

Item = a node or an atomic value of a simple

Preserve relative hierarchy and sequence of input structures

influenced by many research groups and query

Common data model, functions and operators

For addressing in

XQuery

predeclared

A query language for any XML

XPath 2.0 + XSLT' + SQL'  (well

XPath used in several other contexts, too:

- Closure property:
  - Results of XML queries are also XML
    (well-formed document fragments)
  - queries can be combined without limit

- Extensibility:
  - should support externally defined functions on all data
types of the data model

Example Query

```xml
<xquery version="1.0">
  <cheapBooks>
    <book price="26.50">
      <title>Designing Internet applications</title>
      <publisher>Prentice Hall</publisher>
      <year>1998</year>
    </book>
    <book price="40.00">
      <title>Computing with Logic</title>
      <author>David Mairson</author>
      <publisher>Benjamin Cummings</publisher>
      <year>1999</year>
    </book>
  </cheapBooks>
</xquery>
```

Syntax “concise and easily understood”

XML-based syntax (XQueryX) also specified

A possible result

XQuery in a Nutshell

- Functional expression language
- Strongly-typed: (optional) type-checking of expressions, and
  validation of results. (We'll concentrate to processing)
  - predeclared prefix for type names:
    xs=http://www.w3.org/2001/XMLSchema
- Extends XPath 2.0
  - XQuery 1.0 and XPath 2.0 Functions and Operators, Rec. Jan.
    2007
  - XQuery = XPath 2.0 + XSLT' + SQL' (roughly)

XQuery and XPath

- XQuery (1.0) is an extension of XPath (2.0)
  - Common data model, functions and operators
  - > study some XPath first
- XPath used in several other contexts, too:
  - For uniqueness constraints in XML Schema
  - For in validation rules of Schematron
  - For pattern matching and selection in XSLT
  - For addressing in XLink and XPointer

XPath in a Nutshell

- XPath 1.0 (W3C Rec. 11/99)
  - a compact non-XML syntax for addressing
    parts of XML documents (as node-sets in
    XPath 1.0)
  - also typical operations on strings, numbers and
    truth values
- XPath 2.0 (2.0 Rec. 1/07) extends and
generalizes:
  - data manipulated as sequences of items
    - Item = a node or an atomic value of a simple
XPath 1.0 vs 2.0

- XPath 2.0 more elegant and complete than 1.0
- Also more complex (Length of specs as pages):

<table>
<thead>
<tr>
<th>Version</th>
<th>Data Model</th>
<th>~ values</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPath 1.0</td>
<td>~ 30</td>
<td>Funcs &amp; opers ~160</td>
</tr>
<tr>
<td>XPath 2.0</td>
<td>~ 80</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>~ 30</td>
<td>~340</td>
</tr>
</tbody>
</table>

Document trees

- Defined in Sect. 5 of XPath 1.0 spec
  - for XSLT/XPath 2.0 & XQuery in their joint Data Model
- Element nodes have elements, text nodes, comments and processing instructions of their (direct) content as their children
  - NB: attribute nodes are not children (but have a parent)
  - > they have no siblings either
- the string value of an document/element is the concatenation of its all text-node descendants

XPath trees: Example

XQuery/XPath/XSLT Data Model

- Documents are viewed as trees made of six types of nodes:
  - root (additional parent of document element)
  - element nodes
  - attribute nodes
  - text nodes
  - Comments and processing instructions
- Obs 1: No entity nodes
- Obs 2: No namespace nodes
  - (XPath/XSLT 1.0 contains them)

XQuery/XPath/XSLT Data Model

- Document order of nodes:
  - = the depth-first traversal order
  - Root first
  - Other nodes in the order of the first character of their XML markup in the document text
  - > an element precedes its attribute nodes, which precede any content nodes of the element
  - Implementation dependent btw nodes belonging to different trees

XQuery/XPath/XSLT Data Model

- Expressions operate on, and return sequences of
  - atomic values (of XML Schema simple types) and
  - nodes
  - an item = a singleton sequence
  - sequences are flat: no sequences as items
    - (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)

Sequence Expressions

- Constant sequences constructed by listing values
  - comma (,) is a catenation operator
    - (1, (2, 3), (1, 1) = (1, 2, 3, 1)
- Shorthands for numeric sequences:
  - 1 to 4 -> (1, 2, 3, 4)
  - 4 to 1 -> ()
  - fn:reverse(1 to 4) -> (4, 3, 2, 1)

Location Paths

- XPath can select any parts of a document tree using ...
- Location paths
  - evaluated with respect to a context item
    - in XQuery typically starting from $x or doc(...)
  - Result: a sequence of nodes in document order, without duplicates
Location paths

- Consist of location steps separated by '/'
  - each step produces a sequence of items
  - steps evaluated left-to-right, each item in turn as the context item
- Complete location step:
  - AxisName: NodeTest ((PredicateExpr)*)
  - axis specifies the tree relationship between the context node and the selected nodes
  - node test restricts the type and and name of nodes
  - filtered further by 0 or more predicates

Location steps: Axes

- In total 12 axes (~ directions in tree)
  - for staying at the context node: self
  - for going downwards:
    - child, descendant, descendant-or-self
  - for going upwards:
    - parent, ancestor, ancestor-or-self
  - for moving towards start/end of the document:
    - preceding-sibling, following-sibling, preceding, following
  - "Special" axes
    - attribute; (+ namespace in Path 1.0)
- Only child, descendant, attribute, self, descendant-or-self, and parent mandatory in XQuery

Path Axes and Their Orientation

- Ordinary axes oriented away from context node (attribute and namespace axes are unordered)
  - the position() for the closest node = 1
  - for the most remote node: position() = last()
- The simplest axis, self::

XPath Axes and Their Orientation

- parent:: (exists for every node except the root)

XPath Axes and Their Orientation

- ancestor::
- ancestor-or-self::

XPath Axes and Their Orientation

- descendant::
- descendant-or-self::

XPath Axes and Their Orientation

- preceding-sibling::
- following-sibling::

XPath Axes and Their Orientation

- self::
Location paths: Node tests

- **Node tests (slightly simplified)**
  - **Name**: any element node with name *Name*
    - (on an attribute axis, any attribute node with name *Name*)
  - `*`: any element (any attribute node on an attribute axis)
  - `text()`: any text node
    - `comment()`: any comment node
    - `processing-instruction()`: any processing instruction
  - `node()`: any node of any type

Location paths: Abbreviations

- **Abbreviations in location steps**
  - `child::` can be omitted
  - `attribute::`: can be shortened to `@`
  - `parent::node()`: can be shortened to `.`
  - Predicate `[position()=n]` for testing occurrence position *n* can be shortened to `[n]`
  - `/descendant-or-self::node()/` shortened to `/`

Notes on Location Paths (1)

- Path 2.0 allows unrestricted expressions as steps
  - but steps except the last must produce nodes only
- Numeric predicates support array-style access: `$rows[$i]`
- Predicates evaluated step at a time. This often causes confusion with shorthand notations:
  - `doc("doc.xml")/title(3)`
  - third `title` child of each parent (likely none!). Why?
  - `doc("doc.xml")/descendant-or-self::node()/child::title[3]`
  - To get the third title in the doc use
    - `(doc("doc.xml")/title)[3]`

Notes on Location Paths (2)

- **References to attributes and subelements easy to use as predicates**
  - Get divisions that are of class `C` or have a `head`:
    - `doc("doc.xml")/div[@class="C" or head]`
  - Values are coerced to Booleans on demand
    - `string/sequence` ➔ true iff non-empty
    - `number` ➔ true iff not zero
  - (but a single number as predicate tests for equality with `position()`)
Set Operations on Node (!) Sequences

- Assume variable bindings:
  \[ s_1 = \{ a, b, c \} \quad s_2 = \{ d, e, c \} \]
- Then:
  \[ s_1 \cup s_2 = \{ a, b, c, d, e \} \quad s_1 \cap s_2 = \{ c \} \quad s_1 \setminus s_2 = \{ a, b \} \]

Node Comparisons

- To compare single nodes,
  - for identity:
    \[ $\text{books//chap[@id="ch1"] is $\text{books//chap[1]}$ true if the chapter with id="ch1" is indeed the first \]
  - for document order: \[ << \text{ and } >> \] $\text{books/chap[@id="ch2"]}$ is $\text{books//title: eq "Intro"}$ true if the chapter with id="ch2" appears after \[ $\text{title}=$"Intro"$/$title$]\

Comparing values of sequences and items

- General comparisons bw sequences:
  - \( =, \neq, <, >, \leq, \geq \)
  - existential semantics: true if some pair of values from operand sequences satisfy the condition
    \( (1,2) = (2,3); (2,3) = (3,4); (1,2) \neq (3,4) \)
- Where in XPath 1.0
  - \( $(a,b) = (c,d) \)
  - \( $(a,b) \neq (c,d) \)
- Value comparisons bw single values:
  - eq, ne, lt, le, gt, ge
  - \( 1 \) eq \( 2 \); \( 10 \) le \( 20 \); $\text{books[@price le 100]}$\

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: \( fn=http://www.w3.org/2005/04/xpath-functions \)

XQuery over XPath

- A query is an expression
- XQuery adds to XPath expressions
  - Element constructors (= XSLT templates)
  - FLWOR expressions
    - \( (\text{“flower”}, \text{for}-\text{let}-\text{where}-\text{order} \text{by}-\text{return}) \)

Central XQuery Expressions

- Path expressions
- Sequence expressions
- Comparison operators
- Quantified expressions
  - \( (\text{some/every} S \text{var in ... satisfies ...}) \)
- Element constructors (= XSLT templates)
- FLWOR expressions
  - \( (\text{“flower”}, \text{for}-\text{let}-\text{where}-\text{order} \text{by}-\text{return}) \)
  - and others, in examples ...

Element Constructors

- Similar to XSLT templates:
  - start and end tag enclosing the content
  - literal fragments written directly, expressions enclosed in braces \( \{ \) and \( \} \)
  - XSLT 1.0 attribute value templates
  - often used inside another expression that binds variables used in the element constructor
    - \( ($\text{There is no current node’ in XQuery}) \)
    - See next

Example

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

Also computed constructors:

\[
\text{element ("emp") \{
  \text{attribute ("empid")=$\{id\}$,}
  \text{name =$\{n\}$,}
  \text{job =$\{j\}$}
\}}\]
FLWOR ("flower") Expressions

- Constructed from for, let, where, order by and return clauses (~SQL select-from-where)
- Syntax: (ForClause | LetClause)^
  - WhereClause?
  - OrderByClause?
  - "return" Expr
- FLWOR binds variables to values, and uses these bindings to construct a result (an ordered sequence of items)

Flow of data in a FLWOR expression

for clauses

- for $V_1$ in Exp_1, ...
- associates each variable $V_i$ 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

Example: for clause

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

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

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
- Compare:
  - for $b$ in doc("bib.xml")/book
    - many bindings (to single books)
  - let $b1$ := doc("bib.xml")/book
    - a single binding (to sequence of books)

Example: let clauses

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

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

for/let clauses

- A FLWOR expr may contain several fors 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 sequences returned by the for expressions

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() (similarly to node-sets in XPath 1.0): empty ~ false, non-empty ~ true
**where clause**

- for binds variables to single items
  - value comparisons, e.g. $color eq "red"
- let to whole sequences → general comparisons, e.g. $colors = "red"
  - for some $c in $colors
    - satisfies $c eq "red"
  - 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 FLWOR expression
- instantiated once for each binding tuple
- often contains element constructors, references to bound variables, and nested sub-expressions

**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>
```

**Positional variables: 'at'**

- For items, can also get their position in the seq:
  ```xml
  for $at at $i in ("a", "b", "c")
  return concat($i, ".", $at, ":")
  ```
  ```xml
  ~> 1.a.2.b.3.c;
  ```
- Could pair items by their position:
  ```xml
  let $boys := doc("kids.xml")//boy,
  $girls := doc("kids.xml")//girl
  for $b at $i in $boys
    where $i le count($girls)
  return <pair><$b, $girls[$i]></pair>
  ```

**Examples (modified from "XML Query Use Cases")**

- Assume: a document named "bib.xml" containing a list of books:
  ```xml
  <book>
    <title>Illustrated TCP/IP</title>
  </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")//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>{ (<group book=list by authors :>
  for $a$ in distinct-values(
    doc("bib.xml")/author
  ) return
  <author>
    <name>[$a]</name>,
    for $b$ in doc("bib.xml")/book[
      author = $a$
    ] return $b/title
  </author>
  )</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
<publisher>
  <name>[$p]</name>,
  for $b$ in doc("bib.xml")/book[
    publisher = $p$
  ] order by $b$/price descending
</publisher>
```

Queries on Document Order

■ Operators $<<$ and $>>$:
  - $x << y = true$ 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.

Computing a "critical sequence"

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

User-defined functions: Example

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

User-defined functions: Example

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

Recursive Transformations

■ Example: “Table-of-contents” for nested sections
  - NB if-then-else (in ordinary XPath 2.0 expressions, too)
    declare namespace my:http://my.own.ns.org;
    declare function my:toc( $n$ as element() ) as element()
    ( if (name($n")="sect")
      then $sect$ {
        for $c$ in $n$/
          return my:toc($c$ )
      } elseif (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?
A recent W3C XML query language, also capable of general XML processing
Vendor support??

- http://www.w3.org/OLR/query
  - mentions ~ 50 prototypes or products (2004: ~ 30, 2005: ~ 40; free, commercial... Oracle, IBM)
- Future?? Interesting confluence of document and database research, and highly potential for XML-based data integration