Querying XML

Outline of the Presentation

- What is XML?
- XML query language : the big picture
- XML data model
- XML expressions
- Complex Xquery examples
- Conclusions

A little bit of history

- *Database world*
  - 1970 relational databases
  - 1990 nested relational model and object oriented databases
  - 1995 semi-structured databases
- *Document world*
  - 1974 SGML (Structured Generalized Markup Language)
  - 1990 HTML (Hypertext Markup Language)
  - 1992 URL (Universal Resource Locator)

- *Data + documents* = *information*
  - 1996 **XML** (Extended Markup Language)
  - URI (Universal Resource Identifier)

What is XML?

- The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web.
- Base specifications:
  - XML 1.0, W3C Recommendation Feb '98
  - Namespaces, W3C Recommendation Jan '99

XML Data Example (1)

```
<book year="1967">
    <title>The politics of experience</title>
    <author>
        <firstname>Ronald</firstname>
        <lastname>Laing</lastname>
    </author>
</book>
```

Elements

- Elements and attributes
- Tree-based, nested, hierarchically organized structure

XML Data Example (2)

```
<book year="1967" xmlns:amz="www.amazon.com">
    <title>The politics of experience</title>
    <author>R.D. Laing</author>
    <amz:ref    amz:isbn="1341-1444-555"/>
    <section>
        The great and true Amphibian, whose
  nature is disposed to…..
        <title>Persons and experience</title>
  Even facts become...
     </section>  …
</book>
```

- Qualified names
- Namespaces
- Mixed content

XML vs. relational data

- Relational data
  - First killer application: banking industry
  - Invented as a mathematically clean *abstract data model*
  - Philosophy: schema first, then data
  - Never had a standard syntax for data
  - Strict rules for data normalization, flat tables
  - Order is irrelevant, textual data supported but not primary goal
- XML
  - First killer application: publishing industry
  - Invented as a *syntax for data, o*nly later an abstract data model
  - Philosophy: data and schemas should be decorrelated, data can exist with or without schema, or with multiple schemas
  - No data normalization, flexibility is a must, nesting is good
  - Order *may* be very important, textual data support a primary goal

The secrets of the XML success

- XML is a general data representation format
- XML is human readable
- XML is machine readable
- XML is internationalized (UNICODE)
- XML is platform independent
- XML is vendor independent
- XML is endorsed by the World Wide web Consortium (W3C)
- XML is not a new technology
- XML is not *only* a data representation format

XML as a family of technologies

- *XML Information Set*
- *XML Schema*
- *XML Query*
- *The Extensible Stylesheet Transformation Language (XSLT)*
- *XML Forms*
- *XML Protocol*
- *XML Encryption*
- *XML Signature*
- *Others*
- *… almost all the pieces needed for a good Web Services puzzle…*

Major application domains for XML

- Data exchange on the Web
  - *e.g.HealthCare Level Seven http://www.hl7.org/*
- Application integration on the Web
  - *e.g. ebXML http://www.ebxml.org/*
- Document exchange on the Web
  - *e.g. Encoded Archival Description Application http://lcweb.loc.gov/ead/*

XML query language

- Why a *query language* for XML ?
  - Preserve logical/physical data independence
    - The semantics is described in terms of an *abstract data model*, independent of the physical data storage
  - Declarative programming
    - Such programs should describe the "*what*", not the "*how*"
- Why a *native* query language ? Why not SQL ?
  - We need to deal with the *specificities* of XML (hierarchical, ordered , textual, potentially schema-less structure)

Brief history of XML query languages

- Research
  - 1995-1997 Semi-structured query languages (e.g. UnQL, Lorel, StruQL, YATL)
  - 1997-1998 XML query languages (e.g.XML-QL, XML-GL)
- Industry
  - 1997 Xpath 1.0
  - 1998 XSLT
- 1999 Creation of a standardization group inside the W3C

XQuery

General Xquery requirements

- Non-procedural, declarative query language
- Human readable syntax
- Protocol independent
- Standard error conditions
- Should not preclude updates

Xquery in a nutshell

- Side effect free, functional language
  - A query is a *prologue* + an *expression* to evaluate
  - Expressions are compiled and evaluated in an environment populated by the query prologue
  - The result of the query is the result of the evaluation of the expression
- Strongly typed
  - Every expression has a type
- Statically typed
  - The type of the result of an expression can be detected statically
- Formal semantics based on XML Abstract Data Model

Xquery type system

- Xquery’s has a powerful (yet complex!) type system
- Xquery types are imported from XML Schemas
- The type system can:
  1. detect statically errors in the queries
  2. infer the type of the result of valid queries
  3. ensure statically that the result of a given query is of a given (expected) type if the input dataset is guaranteed to be of a given type

XML Data Model

- Common for Xpath 2.0 and XQuery 1.0
- Same goal as the relational data model for SQL
    - table -> SQL -> tables
    - *XML trees* -> Xquery -> *XML trees*
- Models well-formed XML data (*untyped*), as well as schema-valid XML data (*typed*)
- Xquery and XSLT are *closed* with respect to the data model

XML Data Model

- Instance of the data model:
  - a *sequence* composed of zero or more *items*
- Items
  - *nodes* or *atomic values*
- Nodes

  document | element | attribute | text | namespaces | PI | comment
- Atomic values
  - Instances of all XML Schema atomic types

  `string, boolean, ID, IDREF, decimal, QName, URI, ...`
  - untyped atomic values

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)

Atomic values

- The values of the 19 *atomic types* available via XML Schema Part II (e.g.: `xs:integer, xs:boolean`, `xs:date`)
- All the *user defined derived atomic types* (e.g ShoeSize)
- Atomic values carry their type together with the value
  - (8, myNS:ShoeSize) is not the same as (8, xs:integer)
- Constructing atomic values in Xquery:

1. Xquery constants
   - xs:string: “125.0” or ‘125.0’
   - xs:integer: 150
   - xs:decimal: 125.0
   - xs:double: 125.e2
2. Special Xquery operators

   `xf:true(),xf:date(“2002-5-20”)`, etc.
3. Via schema validation of a document

XML nodes

- 7 types of nodes:
  - document | element | attribute | text | namespaces | PI | comment
- Every node has a unique node identifier
- Nodes have children and an optional parent
  - conceptual "tree"
- Nodes are ordered based of the topological order in the tree ("document order")

Example of well formed XML data

```
<book year="1967" xmlns="www.amazon.com">
    <title>The politics of experience</title>
    <author>R.D. Laing</author>
</book>
```

- 3 element nodes, 1 attribute node, 1 NS node, 2 text nodes
  - name(book element) = {www.amazon.com}:book
- In the absence of schema validation
  - type(book element) = xs:anyType
  - type(author element) = xs:anyType
  - type(year attribute) = xs:anySimpleType
  - typed-value(author element) = “R.D. Laing”
  - typed-value(year attribute) = “1967”

XML schema example

```
<type name="book-type">
    <sequence>
        <attribute name="year" type="xs:integer">
        <element name="title" type="xs:string">
        <sequence minoccurs="0">
            <element name="author" type="xs:string>
        </sequence>
    </sequence>
</type>
<element name="book" type="book-type">
```

Schema validated XML data

```
<book year="1967" xmlns="www.amazon.com">
    <title>The politics of experience</title>
    <author>R.D. Laing</author>
</book>
```

- After schema validation
  - type(book element) = myNs:book-type
  - type(author element) = xs:string
  - type(year attribute) = xs:integer
  - typed-value(author element) = “R.D. Laing”
  - typed-value(year attribute) = 1967
- Schema validation impacts the data model representation and therefore the Xquery semantics

XML queries

- An Xquery unit:
  - a *prolog* + an *expression*
- Role of the prolog:
  - Populate the context where the expression is compiled and evaluated
- Prologue contains:
  - namespace definitions
  - schema imports
  - default element and function namespace
  - function definitions
  - collations declarations
  - function library imports
  - global and external variables definitions
  - etc

Xquery expressions

Xquery Expr := Constants | Variable | FunctionCalls

PathExpr |

ComparisonExpr | ArithmeticExpr | LogicExpr |

FLWRExpr | ConditionalExpr | QuantifiedExpr |

TypeSwitchExpr | InstanceofExpr | CastExpr |

UnionExpr | IntersectExceptExpr |

ConstructorExpr

Expressions can be nested with full generality !

Constants

Xquery grammar has built-in support for:

- Strings: "125.0" or '125.0'
- Integers: 150
- Decimal: 125.0
- Double: 125.e2

- 19 other *atomic types* available via XML Schema
- Values can be constructed
  - with constructors in F&O doc: `xf:true(), xf:date("2002-5-20")`
  - by casting
  - by schema validation

Variables

- $ + QName
- bound, not assigned
- created by `let`, `for`, `some/every`, `typeswitch` expressions, function parameters
- example:

```
let $x := ( 1, 2, 3 )
return count($x)
```

- above scoping ends at conclusion of `return` expression

Constructing sequences

```
(1, 2, 2, 3, 3, <a/>, <b/>)
```

- "," is the sequence concatenation operator
- Nested sequences are flattened:

```
(1, 2, 2, (3, 3)) => (1, 2, 2, 3, 3)
```

- range expressions: `(1 to 3) => (1, 2, 3)`

Combining sequences

- Union, Intersect, Except
- Work only for sequences of nodes, not atomic values
- Eliminate duplicates and reorder to document order

```
$x := <a/>, $y := <b/>, $z := <c/>

($x, $y) union ($y, $z) => (<a/>, <b/>, <c/>)
```

- F&O specification provides other functions & operators; eg `xf:distinct-values()` and `xf:distinct-nodes()` particularly useful

Arithmetic expressions

```
1 + 4                  $a div 5
5 / 6                  $b mod 10
1 - (4 * 8.5)          -55.5
<a>42</a> + 1          <a>baz</a> + 1
validate {<a xsi:type="xs:integer">42</a> }+ 1
validate {<a xsi:type="xs:string">baz</a> }+ 1
```

- Apply the following rules:
  - *atomize* all operands. if either operand is (), => ()
  - if an operand is untyped, cast to `xs:double` (if unable, => `error`)
  - if the operand types differ but can be *promoted* to common type, do so (e.g.: `xs:integer` can be promoted to `xs:decimal`)
  - if operator is consistent w/ types, apply it; result is either atomic value or `error`
  - if type is not consistent, throw type exception

Logical expressions

```
expr1 and expr2
expr1 or expr2
```

- returns `true, false`
- *two value logic,* not three value logic like SQL !
- Rules:
  - first compute the *Boolean Effective Value (BEV)* for each operand:
    - if (), “”, NaN, 0, return `false`
    - if the operand is of type boolean, its BEV is its value;
    - else return `true`
  - then use standard two value Boolean logic on the two BEV's as appropriate
- false and error => false *or* error ! (non-deterministically)

Comparisons

Value	for comparing single values	`eq, ne, lt, le, gt, ge`
General	above + *some* semantics and atomization	`=, !=, <=, <, >, >=`
Node	for testing identity of single nodes	`is, isnot`
Order	testing relative position of one node vs. another (in document order)	`<<, >>`

Value and general comparisons

• `<a>42</a>  eq  42`	`true`
• `<a>42</a>  eq  42.0`	`true`
• `<a>42</a>  eq  <b>42</b>`	`true`
• `<a>42</a> eq <b>   42</b>`	`false`
• `<a>baz</a>  eq  42`	`type error`
• `() eq 42`	`()`
• `(<a>42</a>, <b>43</b>)  =  42`	`true`

Conditional expressions

- Syntax :

```
if  ( expression1 )
    then expression2 else expression3
```

- Example :

```
if ( $book/@year <1980 )
        then “old book”
        else “new book”
```

XPath expressions

- Express navigation in a XML tree
- Xpath 2.0 and Xquery 1.0 are designed jointly
- Share the data model, type system and built in Functions and Operators library
- Xpath 2.0 syntactically backwards compatible with Xpath 1.0
- Xpath 2.0 *almost* semantically backwards compatible with Xpath 1.0

Xpath expressions

- General syntax:

  expression ‘/’ step

- Two syntaxes: abbreviated or not
- Step in the non-abbreviated syntax:

  axis ‘::’ nodeTest

- Axis control the navigation direction in the tree
  - attribute, child, descendant, descendant-or-self, parent, self
- Node test by:
  - Name (e.g. publisher, myNS:publisher, *: publisher, myNS:* , *:*)
  - Kind (e.g. node(), comment(), text())

Examples of path expressions

- `document(“bibliography.xml”)/child::bib`
- `$x/child::bib/child::book/attribute::year`
- `$x/parent::*`
- `$x/child::*/descendent::comment()`

Xpath abbreviated syntax

- Axis can be missing
  - By default the child axis

```
$x/child::person  ->    $x/person
```

- Short-hands for common axes
  - Descendent-or-self

```
$x/descendant::comment()-> $x//comment()
```

  - Parent

```
$x/parent::*    ->     $x/..
```

  - Attribute

```
$x/attribute::year   ->   $x/@year
```

  - Self

```
$x/self::*     ->     $x/.
```

Xpath filter predicates

- Syntax:

  expression1 [*expression2*]

- [ ] is an overloaded operator

- Filtering by predicate :

```
//book [author/firstname = “ronald”]
//book [@price <25]
//book [count(author [@gender=“female”] )>0 ]
```

- Filtering by position :

```
/book[3]
/book[3]/author[1]
/book[3]/author[1 to 2]
```

Simple iteration expression

- Syntax :

```
for  variable in expression1
return expression2
```

- Example

```
for $x in document("bib.xml")/bib/book
return  $x/title
```

- Semantics :
    - bind the variable to each root node of the forest returned by *expression1*
    - for each such binding evaluate *expression2*
    - concatenate the resulting sequences
    - nested sequences are automatically flattened

Local variable declaration

- Syntax :

```
let variable := expression1
return expression2
```

- Example :

```
let $x :=document(“bib.xml”)/bib/book
return count($x)
```

- Semantics :
  - bind the *variable* to the result of the *expression1*
  - add this binding to the current environment
  - evaluate and return *expression2*

FLWR expressions

- Syntactic sugar that combines FOR, LET, IF

- Example

```
for $x in //bib/book                          /* similar to FROM in SQL */
 let $y := $x/author                          /* no analogy in SQL */
 where $x/title="The politics of experience"
                                              /* similar to WHERE in SQL */
 return count($y)                             /* similar to SELECT in SQL */
```

FLWR expression semantics

- FLWR expression:

```
for $x in //bib/book
let $y := $x/author
where $x/title="Ulysses"
return count($y)
```

- Equivalent to:

```
for $x in //bib/book
return  (let $y := $x/author
         return
                 if ($x/title="Ulysses" )
                 then count($y)
                 else  ()
         )
```

More FLWR expression examples

- Selections

```
for $b in document("bib.xml")//book
where $b/publisher = “Springer Verlag" and
      $b/@year = "1998"
return $b/title
```

- Joins

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

Quantified expressions

- Syntax:

  some *variable* in *expression1* satisfies *expression2*

  every *variable* in *expression1* satisfies *expression2*

- Examples:
  - `some $x in  //book satisfies $x/price >200`
  - `//book[some $x in author satisfies $x/@gender=“female”]`
  - ```
    for $x in //book
    where  every $y in $x/author
           satisfies $y/@gender=“female”
    return $x/title
    ```
    ```

Node constructors

- In XQuery, we can either return nodes we find using path expressions (selection), or we can construct new nodes
  - elements
  - attributes
  - documents
  - processing instructions
  - comments
  - text

- XML and non-XML syntax to construct elements and attributes

Literal vs. evaluated element content

```
<result>
    literal text content
</result>


<result>
       {  $x/name     }  {-- evaluated content --}
</result>


<result>
     some content here {$x/name} and some more  here
</result>
```

- Braces "`{}`" used to delineate evaluated content

Same works for attributes

Operators on datatypes

```
expression instanceof sequenceType
```

- returns true if its first operand is an instance of the type named in its second operand

```
expression castable as sequenceType
```

- returns true if first operand can be casted as the given sequence type

```
cast as sequenceType {expression }
```

- used to convert a value from one datatype to another

```
treat as sequenceType {expression }
```

- treats an expr as if its datatype is a subtype of its static type (down cast)

```
typeswitch
```

- case-like branching based on the type of an input expression

Complex Xquery example

```
<bibliography>
      { for $x in //book[@year=2001]
        return
            <book  title="{$x/title}">
                  { if(empty($x/author))
                 then $x/editor
                    else  $x/author
                  }
           </book>
       }
  </bibliography>
```

XSLT-like transformations

```
<HTML>
<TABLE>
{
for $b in document("data/xmp-data.xml")//book
return
     <TR>
          <TD>{$b/title}</TD>
          <TD>{$b/author/last}</TD>
     </TR>
}
</TABLE>
</HTML>
```

Joins in XQuery

```
<books-with-prices>
   {for  $a in document('amaxon.xml')/book,
         $b in document('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>
```

Left-outer joins in XQuery

```
<books-with-prices>
  {
   for  $a in document('amaxon.xml')/book
   return
      <book>
       {$a/title}
        <price-amazon>{$a/price}</price-amazon>
       {  for $b in document('bn.xml')/book
          where $b/isbn=$a/isbn
          return
            <price-bn>{$b/price}</price-bn>
         }
     </book>
    }
 </books-with prices>
```

Full-outer joins in Xquery

```
let $allISBNs:=distinct-value(
                 document(‘amazon.xml’)/book/isbn
                     union document(‘bn.xml’)/book/isbn ))
return
 <books-with-prices>
     {for  $isbn in $allISBNs
      return
        <book>
         {for $a in document(‘amazon.xml’)/book[isbn=$isbn]
            return
              <price-amazon>{$b/price}</price-amazon>
         }
         {for $b in document(“bn.xml”)/book [isbn=$isbn]
           return  <price-bn>{$b/price}</price-bn>
         }
        </book>
      }
 </books-with prices>
```

Group-by and Having

```
for $a in distinct-value(//book/author/lastname)
let $books:=//book[some $y in author/lastname=$a]
where count($books)>10
return <result  lastname="{$a}">
                    {$books[1 to 10]}
          </result>
```

Content exchanger

```
define function swizzle (xs:anyElement $x)
returns xs:anyElement
{
    element {name($x)}
    {
     for $attr in $x/@*
        return element {name($attr)}{$attr/data()},
     for $elem in $x/*
        return attribute {name($elem)}{$elem/data()}
    }
}
swizzle( <a b="1“><c>empty</c></a> )

    =>   <a c="empty"><b>1<b/></a>
```

XML query language summary

- Declarative
- Expressive power
  - Major functionality of SQL, OQL , Xpath, XSLT
- Query the many kinds of data XML contains
- Very versatile: transformation language, query language, integration language, etc
- Can be implemented in many environments
  - Traditional databases, XML repositories, XML programming libraries, etc.
  - Queries may combine data from many sources

Conclusion

- Expressive, concise
- Implementable, optimizable
- Many existing implementations
- Short term future:
  - *Update language for XML data*