Module 5

Introduction to XQuery
XML is now everywhere

- Google search (warning: unreliable numbers)
  - 285,000,000 for XML
  - 1,000,000 for XQuery
  - 11,000,000 for XSLT
  - 12,000,000 for XML Schema
  - 60,000,000 for .NET
  - 200,000,000 for Java
  - 64,000,000 for SQL

- The highest Google number among all the technology buzzwords that I searched (except RSS)
Sources of XML data

1. Inter-application communication data (WS, Rest, etc)
2. Mobile devices communication data
3. Logs
4. Blogs (RSS)
5. Metadata (e.g. Schema, WSDL, XMP)
6. Presentation data (e.g. XHTML)
7. Documents (e.g. Word)
8. Views of other sources of data
   - Relational, LDAP, CSV, Excel, etc.
9. Sensor data
Some vertical application domains for XML

- HealthCare Level Seven [http://www.hl7.org/]
- Geography Markup Language (GML)
- Systems Biology Markup Language (SBML) [http://sbml.org/]
- XBRL, the XML based Business Reporting standard [http://www.xbrl.org/]
- Global Justice XML Data Model (GJXDM) [http://it.ojp.gov/jxdm]
- ebXML [http://www.ebxml.org/]
- e.g. Encoded Archival Description Application [http://lcweb.loc.gov/ead/]
- Digital photography metadata XMP
- An XML grammar for sensor data (SensorML)
- Real Simple Syndication (RSS 2.0)

Basically everywhere.
Processing the XML data

• Huge amount of XML information, and growing
• We need to “manage” it, and then “process” it
  • Store it efficiently
  • Verify the correctness
  • Filter, search, select, join, aggregate
  • Create new pieces of information
  • Clean, normalize the data
  • Update it
  • Take actions based on the existing data
  • Write complex execution flows
• No conceptual organization like for relational databases (applications are too heterogeneous)
Frequent solutions to XML data management

1. Map it to *generic* programming APIs (e.g. DOM, SAX, StaX)
2. *Manually* map it to *non-generic* APIs
3. *Automatically* map it to *non-generic* structures
4. Use *XML extensions* of existing languages
5. Shredding for relational stores
6. *Native XML processing through XSLT and XQuery*
1. Mapping to generic structures

- Represent the data:
  - Original UNICODE form or
  - Some binary representation (e.g. FastInfoset)

- Store it:
  - Directly on a file system or
  - On a “transacted” file system (e.g. SleepyCat, or a relational database)

- Map the XML data to generic XML programmatic APIs
  - E.g. Dom, Sax, Stax (JSR 173), XMLReader

- Use the native programming languages (e.g. Java, C#) to manipulate the data

- Re-serialize it at the end
1. Manual mapping to generic structures (example)

```xml
<purchaseOrder>
  <lineItem>
    ....
  </lineItem>
  <lineItem>
    ....
  </lineItem>
</purchaseOrder>

<book>
  <author>...</author>
  <title>....</title>
  ....
</book>
```

Class DomNode{
  public String getNodeName();
  public String getNodeValue();
  public void setValue(nodeValue);
  public short getType();
}

Hard coded mappings
2. Manual mapping to non-generic structures

```xml
<purchaseOrder>
  <lineItem>
    ..... 
  </lineItem>
  <lineItem>
    ..... 
  </lineItem>
</purchaseOrder>

Class PurchaseOrder{
  public List getLineItems();
  .......
}

Class Book{
  public List getAuthor();
  public String getTitle();
  ......
}
```

Hard coded mappings
3. Automatic mapping to non-generic structures

```xml
<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">
  Automatic mapping
e.g. XMLBeans

Class Book-type{
  public integer getYear();
  public string getTitle();
  public List getAuthors();
  .......
}
```
4. XML extensions of existing procedural languages

- Examples:
  - C-omega, ECMAscript, PHP extensions, Phyton extensions, etc.

- Most of them define:
  - A way of importing XML data into their native type system
  - A rich API for XML data manipulation
  - A way of navigating/searching/querying the XML data via their extensions (Xpath based or Xpath inspired)
5. Native XML processing
XSLT and XQuery

- **Most promising alternative for the future.**
- The **only** alternative such that:
  - the data is modeled only once
  - is well integrated with XML Schema type system
  - it preserves the logical/physical data independence
  - the code deals with non-generic structures
  - Code can be optimized automatically

- **Data is stored:**
  - in plain file systems or in sophisticated data stores (e.g. XML extensions of relational stores)

- **Missing pieces, under development**
  - E.g. no procedural logic
Why XQuery?

Why a “query” language for XML?
- Need to process XML data
- 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)

Why another XML processing language? Why not XSLT?
- The template nature of XSLT was not appealing to the database people. Not declarative enough.
What is XQuery?

- A programming language that can express arbitrary XML to XML data transformations
  - Logical/physical data independence
  - “Declarative”
  - “High level”
  - “Side-effect free”
  - “Strongly typed” language
- “An expression language for XML.”
- Commonalities with functional programming, imperative programming and query languages
- The “query” part might be a misnomer (***)
XQuery family of standards

• **XQuery 1.0**: An XML Query Language: an XML-aware syntax for querying collections of structured and semi-structured data both locally and over the Web.

• **XSL Transformations (XSLT) Version 2.0**: transforms data model instances (XML and non-XML) into other documents, including into XSL-FO for printing.

• **XML Path Language (XPath) 2.0**: expression syntax for referring to parts of XML documents.

• **XQuery 1.0 and XPath 2.0 Functions and Operators**: the functions you can call in XPath expressions and the operations you can perform on XPath 2.0 data types.

• **XQuery 1.0 and XPath 2.0 Data Model (XDM)**: representation and access for both XML and non-XML sources.

• **XSLT 2.0 and XQuery 1.0 Serialization**: how to output the results of XSLT 2.0 and XML Query evaluation in XML, HTML or as text.

• **XML Syntax for XQuery 1.0 (XQueryX)**: an XML-aware syntax for querying collections of structured and semi-structured data both locally and over the Web.

• **XQuery 1.0 and XPath 2.0 Formal Semantics**: the type system used in XQuery and XSLT 2 via XPath defined precisely for implementers.
XQuery, Xpath, XSLT

- XQuery 1.0
  - FLWOR expressions
  - Node constructors
  - Validation

- XSLT 2.0
  - uses

- Xpath 2.0
  - extends
  - 2007

- Xpath 1.0
  - extends, almost backwards compatible
  - 1999

- XSLT 1.0
  - uses
Roadmap for today

- XQuery Data Model (XDM)
- XQuery type system
- Xquery environment
- XQuery basic constructs
  - variables
  - constants
  - function calls, function library
  - arithmetic operations
  - boolean operations
  - path expressions
  - conditionals
The need for an abstract XML data model

- XML 1.0 specification only talks about characters
- We cannot have a programming language processing “characters” (one by one)
- An XML abstract/logical data model !?
- Unfortunately too many of those
  - Infoset, PSVI, DOM, **XDM**, etc
XML Data Model (XDM)

- Abstract (I.e. logical) data model for XML data
- Same role for XQuery as the relational data model for SQL
- Purely logical --- no standard storage or access model (in purpose)
- XQuery is closed with respect to the Data Model

Diagram:
- PSVI
- Infoset
- XML Data Model
- XQuery
- Xpath 2.0
- XSLT 2.0
XML Data model life cycle

- **parse**
- **validate**
- **serialize**

- `.xml`
- `.xsd`

- XQuery Data Model
- XSLT 2.0
- XQuery
- Xpath 2.0

application-dependent
XML Data Model

- Instance of the data model:
  - a **sequence** composed of zero or more **items**
  - The **empty sequence** often considered as the “null value”
- 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
- **Typed** (I.e. schema validated) and **untyped** (I.e. non schema validated) nodes and values

Remember Lisp?
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 in XML Schema*
  - E.g. xs:integer, xs:boolean, xs:date
- All the *user defined derived atomic types*
  - E.g myNS:ShoeSize
- xs:untypedAtomic
- Atomic values carry their type together with the value
  - (8, myNS:ShoeSize) is not the same as (8, xs:integer)
XML nodes

- 7 types of nodes:
  - document | element | attribute | text | namespaces | PI | comment

- Every node has a unique **node identifier**
  - Scope of node identifier uniqueness is implementation dependent

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

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

- node-kind : xs:string
- node-name : xs:Qname ?
- parent : node() ?
- string-value : xs:string
- typed-value : xs:anyAtomicType *
- type-name : xs:Qname ?
- children : node() *
- attributes : attribute() *
- namespaces : node() *
Example of well formed XML data

```xml
<book year="1967">
  <title>The politics of experience</title>
  <author>R.D. Laing</author>
</book>
```

- 3 element nodes, 1 attribute node, 5 text nodes
  - name(book element) = {-}:book
- In the absence of schema validation
  - type(book element) = xs:untyped
  - type(author element) = xs:untyped
  - type(year attribute) = xs:untypedAtomic
  - typed-value(author element) = ("R.D. Laing", xs:untypedAtomic)
  - typed-value(year attribute) = ("1967", xs:untypedAtomic)
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>
    </element>
    </sequence>
  </sequence>
</type>

<element name="book" type="book-type">
Schema validated XML data

<book year="1967">
  <title>The politics of experience</title>
  <author>R.D. Laing</author>
</book>

- After schema validation
  - type(book element) = {uri}:book-type
  - type(author element) = xs:string
  - type(year attribute) = xs:integer
  - typed-value(author element) = ("R.D. Laing", xs:string)
  - typed-value(year attribute) = (1967, xs:integer)

- Schema validation impacts the data model representation and therefore the XQuery semantics!!
Lexical and binary aspect of the data

- Every node holds (logically) redundant information:
  - `<a xsi:type="xs:integer">001</a>`
  - `dm:string-value () “001” as xs:string`
  - `dm:typed-value ()
    - “001” as an xs:untyped before validation
    - 1 as an xs:integer after validation`

- Implementations can store:
  - The **string value**
    - Retrieve the typed value dynamically based on the type, every time is needed
  - The **typed value**
    - Retrieve an acceptable lexical value for that type every time this is required
  - Both

- In case of unvalidated data the two are the same
Typed vs. untyped XML Data

• **Untyped data** (non XML Schema validated)
  
  `<a>3</a>      eq     3
  <a>3</a>      eq     “3”

• **Typed data** (after XML Schema validation)
  
  `<a xsi:type="xs:integer">3</a>      eq     3
  `<a xsi:type="xs:string">3</a>        eq     3
  `<a xsi:type="xs:integer">3</a>      eq     “3”
  `<a xsi:type="xs:string">3</a>        eq     “3”
XML data equivalence

- XQuery has multiple notions of data “equality”
  - “=”, “eq”, “is”, “fn:deep-equal()”

- Expected properties:
  - Transitivity, reflexivity and symmetry
  - Necessary for grouping, indexing and hashing

- Additional property:
  - if (data₁ equal data₂) then (f(data₁) equal f(data₂))
  - Necessary for memoization, caching

- None of the equality relationships above (except “is”) satisfies those properties

- The “is” relationship only applies to nodes

- Careful implementations for indexes, hashing, caches
Document order

<book year="1967" price="45.32">
  <title>The politics of experience</title>
  <author>R.D. Laing</author>
</book>

- How many nodes here?
- What is the order between nodes?
Document order

<book(n1) year(n2) =“1967” price(n3)=“45.32”>(n4)
  <title(n5)>(n6) The politics of experience</title>(n7)
  <author(n8)>(n9) R.D. Laing</author>
</book>

- How many nodes here? 9
- What is the order between nodes?
  - n1 before all the others
  - order of n2 and n3 non-deterministic
  - n2 and n3 are before n4,n5,n6,n7,n8,n9
  - n4<n5<n6<n7<n8<n9 (top-down, left to right among the children)
XQuery type system

- XQuery has a powerful (and complex!) type system
- XQuery types are imported from XML Schemas
- Every XML data model instance has a dynamic type
- Every XQuery expression has a static type
- Pessimistic static type inference

The goal of the type system is:

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
XQuery type system components

- Atomic types
  - `xs:untypedAtomic`
  - All 19 primitive XML Schema types
  - All user defined atomic types
- Empty, None
- Type constructors (simplification!)
  - Elements: `element name {type}`
  - Attributes: `attribute name {type}`
  - Alternation: `type1 | type2`
  - Sequence: `type1, type2`
  - Repetition: `type*`
  - Interleaved product: `type1 & type2`
- `type1 intersect type2` ?
- `type1 subtype of type2` ?
- `type1 equals type2` ?
XML queries

- An XQuery basic structure:
  - 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 processing

External Processing
Data Model Generation

XML

(XM)
Parse and optionally validate

(DM1)

Infoset/PSVI

(DM2)
Generate Data Model

(Data Model Instances)

(DM3)
Other/Direct Generation of Data Model

Query Processing
Static analysis phase

XQuery

(SQ1) Parse query

Op-Tree

(SQ2) Initialize from environment

(SQ3) Process query prolog

(SQ4) Resolve names

(SQ5) Normalize

Dynamic evaluation phase

(DQ1) Access Op-Tree

(DQ2) Provide access to Static context

(DQ3) Initialize from environment

(DQ4) Access and create Execution Engine

(DQ5) Access and change

Schema Import Processing

XSD

(SI1) Generate

In-scope schema definitions

(SI2) Other/Direct Generation from environment

* Only if static typing enabled
** Dynamic type check if static typing not enabled
*** Need not be well-formed XML
XQuery expressions

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

Expressions can be nested with full generality!

Functional programming heritage (ML, Haskell, Lisp)
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: `fn:true()`, `fn:date("2002-5-20")`
  - by casting
  - by schema validation
Variables

- $ + Qname (e.g. $x, $ns:foo)
- bound, not assigned
- **XQuery does not allow variable assignment**
- 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
A built-in function sampler

- `fn:document(xs:anyURI) => document?`
- `fn:empty(item*) => boolean`
- `fn:index-of(item*, item) => xs:unsignedInt?`
- `fn:distinct-values(item*) => item*`
- `fn:distinct-nodes(node*) => node*`
- `fn:union(node*, node*) => node*`
- `fn:except(node*, node*) => node*`
- `fn:string-length(xs:string?) => xs:integer?`
- `fn:contains(xs:string, xs:string) => xs:boolean`
- `fn:true() => xs:boolean`
- `fn:date(xs:string) => xs:date`
- `fn:add-date(xs:date, xs:duration) => xs:date`

See Functions and Operators W3C specification
Atomization

- \( \text{fn:}\text{data(item*)} \rightarrow \text{xs:anyAtomicType*} \)
- Extracting the “value” of a node, or returning the atomic value
- Implicitly applied:
  - Arithmetic expressions
  - Comparison expressions
  - Function calls and returns
  - Cast expressions
  - Constructor expressions for various kinds of nodes
  - order by clauses in FLWOR expressions
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

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

($x, y) \text{ union } (y, z) \Rightarrow (<a/>, <b/>, <c/>)$

- F&O specification provides other functions & operators; eg. \texttt{fn:distinct-values()} and \texttt{fn:distinct-nodes()} particularly useful
Arithmetic expressions

1 + 4  $a \div 5$
5 \div 6  $b \mod 10$
1 - (4 \times 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">42</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:double*)
  - 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 \texttt{and} expr2

expr1 \texttt{or} expr2 \quad \texttt{fn:not()} as a function

- return true, false

- Different from SQL
  - \texttt{two} value logic, not \texttt{three} value logic

- Different from imperative languages
  - \texttt{and}, \texttt{or} are commutative in Xquery, but not in Java.
  - if ((\$x castable as xs:integer) and ((\$x cast as xs:integer) eq 2) ) ..... 

- Non-deterministic
  - false and error => false \texttt{or} error ! (non-deterministically)

- Rules:
  - first compute the \textit{Boolean Effective Value (BEV)} for each operand:
    - if (), "", NaN, 0, then return false
    - if the operand is of type xs:boolean, return it;
    - If operand is a sequence with first item a node, return true
    - else raises an error
  - then use standard two value Boolean logic on the two BEV's as appropriate
## Comparisons

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Existential quantification + automatic type coercion</td>
<td>=, !=, &lt;=, &lt;, &gt;, &gt;=</td>
</tr>
<tr>
<td>Node</td>
<td>for testing identity of single nodes</td>
<td>is, isnot</td>
</tr>
<tr>
<td>Order</td>
<td>testing relative position of one node vs. another (in document order)</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td>Value</td>
<td>for comparing single values</td>
<td>eq, ne, lt, le, gt, ge</td>
</tr>
</tbody>
</table>
Value and general comparisons

- `<a>42</a>` eq “42”      true
- `<a>42</a>` eq 42          error
- `<a>42</a>` eq “42.0”     false
- `<a>42</a>` eq 42.0       error
- `<a>42</a>` = 42          true
- `<a>42</a>` = 42.0        true
- `<a>42</a>` eq `<b>42</b>` true
- `<a>42</a>` eq `<b>42</b>` false
- `<a>baz</a>` eq 42       error
- `()` eq 42               `()`
- `()` = 42                false
- `(a42, b43)` = 42.0      true
- `(a42, b43)` = “42”      true
- `ns:shoesize(5) eq ns:hatsize(5)` true
- `(1,2) = (2,3)`          true
Algebraic properties of comparisons

- General comparisons not reflexive, transitive
  - \((1,3) = (1,2)\) (but also \(!=, <, >, <=, >=\) !!!)
  - Reasons
    - implicit existential quantification, dynamic casts
- Negation rule does not hold
  - \(\text{fn:not}($x = $y)\) is not equivalent to $x != $y
- General comparison not transitive, not reflexive
- Value comparisons are *almost* transitive
  - Exception:
    - \(\text{xs:decimal}\) due to the loss of precision

Impact on grouping, hashing, indexing, caching !!!
XPath expressions

- An expression that defines the set of nodes where the navigation starts + a series of selection steps that explain how to navigate into the XML tree

- A step:
  - \textit{axis}`: ` nodeTest

- Axis control the navigation direction in the tree
  - \textit{attribute}, \textit{child}, \textit{descendant}, \textit{descendant-or-self}, \textit{parent}, \textit{self}
  - The other Xpath 1.0 axes (\textit{following}, following-sibling, preceding, preceding-sibling, ancestor, ancestor-or-self) are optional in XQuery

- Node test by:
  - \textbf{Name} (e.g. publisher, myNS:publisher, `*`: publisher, myNS:`*`, `*:*`)  
  - \textbf{Kind of item} (e.g. node(), comment(), text())
  - \textbf{Type test} (e.g. element(ns:PO, ns:PoType), attribute(`*, xs:integer)
Examples of path expressions

- `document("bibliography.xml")/child::bib`
- `$x/child::bib/child::book/attribute::year`
- `$x/parent::*`
- `$x/child::*/descendent::comment()`
- `$x/child::*/element(*, ns:PoType)`
- `$x/attribute::attribute(*, xs:integer)`
- `$x/ancestors::document(schema-element(ns:PO))`
- `$x/(child::element(*, xs:date) | attribute::attribute(*, xs:date))`
- `$x/f(.)`
Xpath abbreviated syntax

- Axis can be missing
  - By default the child axis
    \$x/\textit{child::}\textit{person} -> \$x/\textit{person}

- Short-hands for common axes
  - Descendent-or-self
    \$x/\textit{descendant-or-self::/*/child::}comment() -> \$x/\textit{comment()}
  - Parent
    \$x/\textit{parent::*} -> \$x/..
  - Attribute
    \$x/\textit{attribute::*\textit{year}} -> \$x/@\textit{year}
  - Self
    \$x/\textit{self::*} -> \$x/.
Xpath filter predicates

- Syntax:
  \[expression1 \ [ expression2 \ ]\]

- [ ] is an overloaded operator

- Filtering by position (if numeric value):
  
  - /book[3]
  - /book[3]/author[1 to 2]

- Filtering by predicate:
  
  - //book [author/firstname = “ronald”]
  - //book [@price <25]
  - //book [count(author [@gender=“female”]) > 0]

- Classical Xpath mistake
  
  - $x/a/b[1]$ means $x/a/(b[1])$ and not ($x/a/b)[1]$
Conditional expressions

if ( $book/@year <1980 )
then ns:WS(<old>{$x/title}</old>)
else ns:WS(<new>{$x/title}</new>)

- Only one branch allowed to raise execution errors
- Impacts scheduling and parallelization