

Querying XML

Susan B. Davidson
susan@cis.upenn.edu



Fall 2004

Some slide content courtesy of Zack Ives CIS 650



1

Querying XML

How do you query a directed graph? a tree?

The standard approach used by many XML, semistructured-data, and object query languages:

- Define some sort of a *template* describing traversals from the *root* of the directed graph
- In XML, the basis of this template is called an XPath



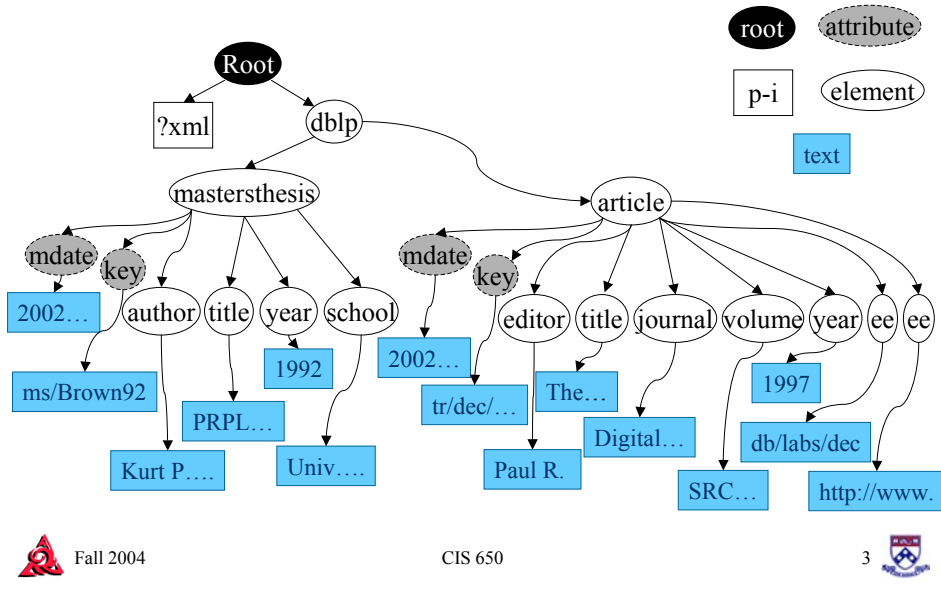
Fall 2004

CIS 650



2

XML Data Model Visualized



Sample XML

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dblp>
  <mastersthesis mdate="2002-01-03" key="ms/Brown92">
    <author>Kurt P. Brown</author>
    <title>PRPL: A Database Workload Specification Language</title>
    <year>1992</year>
    <school>Univ. of Wisconsin-Madison</school>
  </mastersthesis>
  <article mdate="2002-01-03" key="tr/dec/SRC1997-018">
    <editor>Paul R. McJones</editor>
    <title>The 1995 SQL Reunion</title>
    <journal>Digital System Research Center Report</journal>
    <volume>SRC1997-018</volume>
    <year>1997</year>
    <ee>db/labs/dec/SRC1997-018.html</ee>
    <ee>http://www.mcjones.org/System_R/SQL_Reunion_95</ee>
  </article>

```



Some Example XPath Queries

- `/dblp/mastersthesis/title`
- `/dblp/*/editor`
- `//title`
- `//title/text()`



Context Nodes and Relative Paths

XPath has a notion of a *context* node, which is analogous to a current directory

- "." represents this context node
- ".." represents the parent node
- We can express relative paths:
 - subpath/sub-subpath/../../ gets us back to the context node

➤ By default, the document root is the context node



Predicates - Selection Operations

A *predicate* allows us to filter the node set based on selection-like conditions over sub-XPath's:

```
/dblp/article[title = "Paper1"]
```

which is equivalent to:

```
/dblp/article[./title/text() = "Paper1"]
```



Axes: More Complex Traversals

Thus far, we've seen XPath expressions that go *down* the tree (and up one step)

- But we might want to go up, left, right, etc.
- These are expressed with so-called *axes*.
 - self::path-step
 - child::path-step
 - descendant::path-step
 - descendant-or-self::path-step
 - preceding-sibling::path-step
 - preceding::path-step
- The previous XPath's we saw were in "abbreviated form"

parent::path-step

ancestor::path-step

ancestor-or-self::path-step

following-sibling::path-step

following::path-step



Querying Order

- We saw in the previous slide that we could query for preceding or following siblings or nodes
- We can also query a node for its position according to some index:
 - `fn::first(), fn::last()` return index of 0th & last element matching the last step:
 - `fn::position()` gives the relative count of the current node

```
child::article[fn::position() = fn::last()]
```



XPath dereferences

- Recall that ID and IDREF can be used to create a reference between one element and another.
- This can be dereferenced in XPath. For example, to find Joe's wife you would write:
 - `/person[@name="Joe"]/@spouse ==> person`



Users of XPath

- XML Schema uses simple XPathS in defining keys and uniqueness constraints
- XQuery
- XSLT
- XLink and XPointer, hyperlinks for XML



XQuery

A strongly-typed, Turing-complete XML manipulation language

- Attempts to do static typechecking against XML Schema
- Based on an object model derived from Schema

Unlike SQL, fully compositional, highly orthogonal:

- Inputs & outputs collections (sequences or bags) of XML nodes
- Anywhere a particular type of object may be used, may use the results of a query of the same type
- Designed mostly by DB and functional language people

Attempts to satisfy the needs of data management *and* document management

- The database-style core is mostly complete (even has support for NULLS in XML!!)
- The document keyword querying features are still in the works - shows in the order-preserving default model



XQuery's Basic Form

- Has an analogous form to SQL's `SELECT..FROM..WHERE..GROUP BY..ORDER BY`
- The model: bind nodes (or node sets) to variables; operate over each legal combination of bindings; produce a set of nodes
- "FLWOR" statement:
 - for {iterators that bind variables}
 - let {collections}
 - where {conditions}
 - order by {order-conditions}
 - return {output constructor}



"Iterations" in XQuery

A series of (possibly nested) FOR statements assigning the results of XPath's to variables

```
for $root in document("http://my.org/my.xml")
  for $sub in $root/rootElement,
    $sub2 in $sub/subElement, ...
```

- Something like a template that pattern-matches, produces a "binding tuple"
- For each of these, we evaluate the WHERE and possibly output the RETURN template
- `document()` or `doc()` function specifies an input file as a URI



Two XQuery Examples

```
<root-tag> {  
  for $p in document("dblp.xml")/dblp/proceedings,  
    $yr in $p/yr  
    where $yr = "1999"  
    return <proc> {$p} </proc>  
} </root-tag>
```

```
for $i in doc ("dblp.xml")/dblp/inproceedings[author/text() = "John  
Smith"]  
return <smith-paper>  
  <title>{ $i/title/text() }</title>  
  <key>{ $i/@key }</key>  
  { $i/crossref }  
</smith-paper>
```



Joins in XQuery

Suppose we have a document of addresses, and a document of movies. Who of our contacts was involved in a movie?

```
<XML>  
{  
  for $p in document("address.xml")//person,  
    $m in document("moviedb.xml")//movie[character=$p/name],  
    return <cin-contact>  
      <who>{$p/name/text()}</who>  
      <movie>{$m/title/text()}</movie>  
      {for $e in $p/email  
        return{<where>{$e/text()}</where>}}  
    </cin-contact>  
}  
<\XML>
```



Nesting in XQuery

Nesting XML trees is perhaps the most common operation

In XQuery, it's easy - put a subquery in the return clause where you want things to repeat!

```
for $u in doc("dblp.xml")/universities
where $u/country = "USA"
return <ms-theses-99>
  { $u/title } {
    for $mt in $u/../mastersthesis
    where $mt/year/text() = "1999"
    return $mt/title }
</ms-theses-99>
```



Equality

- Equality
 - node-equal: same node
 - deep-equal: same value

```
let $first:= {<val>1</val>, 2,3}
    $second:={<val>1</val>, 2,3}
return <result>
  Node: {sequence-node-equal($first, $second)}
  Deep: {sequence-deep-equal($first, $second)}
</result>
```

```
Result:
<result>
  Node: false
  Deep: true
</result>
```



Collections & Aggregation

- In XQuery, many operations return collections
 - XPath, sub-XQueries, functions over these, ...
 - The let clause assigns the results to a variable
- Aggregation simply applies a function over a collection, where the function returns a value

```
let $allpapers := doc("dblp.xml")/dblp/article
return <article-authors>
  <count> {fn:count(fn:distinct-values($allpapers/authors))} </count>
{
  for $paper in doc("dblp.xml")/dblp/article
  let $pauth := $paper/author
  return <paper> {$paper/title}
    <count> {fn:count($pauth)} </count>
  </paper>
} </article-authors>
```



Sorting in XQuery

- SQL allows you to sort its output, with a special ORDER BY clause (which we haven't discussed)
- XQuery borrows this idea
- In XQuery, what we order is the sequence of "result tuples" output by the return clause:

```
for $x in doc("dblp.xml")/proceedings
order by $x/title/text()
return $x
```



What if order doesn't matter?

- By default:
 - SQL is unordered
 - XQuery is ordered everywhere!
 - But unordered queries are much faster to answer
- XQuery has a way of telling the DBMS to avoid preserving order:
 - for \$x in fn:unordered(mypath) ...
 - Some of us feel the default is "wrong"...



Distinct-ness

- XQuery has a notion that DISTINCT-ness happens as a function over a collection
 - But since we have nodes, we can do duplicate removal according to value or node
 - Can do `fn:distinct-values(collection)` to remove duplicate values, or `fn:distinct-nodes(collection)` to remove duplicate nodes

```
for $years in fn:distinct-  
values(doc("dblp.xml")//year/text())  
return $years
```



Querying & Defining Metadata

- Can't do this in SQL..
- Can get a node's name by querying node-name():

```
for $x in document("dblp.xml")/dblp/*  
return node-name($x)
```
- Can construct elements and attributes using computed names:

```
for $x in document("dblp.xml")/dblp/*,  
  $year in $x/year,  
  $title in $x/title/text(),  
element node-name($x) {  
  attribute {"year-" + $year} { $title }  
}
```



XQuery: Beyond FLWR

- XQuery has many built-in functions and predicates, such as
 - count(), sum(), min(), max(), position(), first(...), last() which work over sequences
 - index-of() finds the position of a node in a sequence
 - Distinct-values(), distinct-nodes() remove duplicates
 - Set operations: union, intersection
- If-then-else statements and function definition ("define function name (params) returns result") are also included



XQuery Summary

- Very flexible and powerful language for XML
 - Clean and orthogonal: can always replace a collection with an expression that creates collections
 - DB and document-oriented (we hope)
 - The core is relatively clean and easy to understand



XSL(T): The Bridge Back to HTML

- XSL (XML Stylesheet Language) is actually divided into two parts:
 - XSL:FO: formatting for XML
 - XSLT: a special transformation language
- We'll ignore for now XSL:FO
- XSLT is actually able to convert from XML → HTML, which is how many people do their formatting today
 - Products like Apache Cocoon generally translate XML → HTML on the server side



A Different Style of Language

- XSLT is based on a series of *templates* that match different parts of an XML document
 - There's a policy for what rule or template is applied if more than one matches (it's not what you'd think!)
 - XSLT templates can invoke other templates
 - XSLT templates can be nonterminating (beware!)
- XSLT templates are based on XPath "match"es, and we can also apply other templates (potentially to "select"ed XPath)s
 - Within each template, we describe what should be output



An XSLT Stylesheet

```
<xsl:stylesheet version="1.1">
  <xsl:template match="/dblp">
    <html><head>This is DBLP</head>
    <body>
      <xsl:apply-templates />
    </body>
  </html>
</xsl:template>
<xsl:template match="inproceedings">
  <h2><xsl:apply-templates select="title" /></h2>
  <p><xsl:apply-templates select="author" /></p>
</xsl:template>
...
</xsl:stylesheet>
```



What XSLT Can and Can't Do

- XSLT is great at converting XML to other formats
 - XML → diagrams in SVG; HTML; LaTeX
 - ...
- XSLT doesn't do joins (well), it only works on one XML file at a time, and it's limited in certain respects
 - It's not a query language
 - ... But it's a very good formatting language
- Most web browsers (post Netscape 4.7x) support XSLT and XSL formatting objects
- But most real implementations use XSLT with something like Apache Cocoon



Wrapping Up

We've seen three XML manipulation formalisms:

- XPath: the basic language for "projecting and selecting" (evaluating path expressions and predicates) over XML
- XQuery: a statically typed, Turing-complete XML processing language
- XSLT: a template-based language for transforming XML documents

- Each is extremely useful for certain applications!

