Today's Lecture

- Document Object Model (XML API's)
- An XML query language: XQuery

- These slides were adapted from slides developed at the University of Pennsylvania (by Peter Buneman and Susan Davidson)

Part III
The Document Object Model (DOM)

Programming with XML
XML Parsers

- traditional: build (main memory) data structure (DOM)
- event based: SAX (Simple API for XML)
  - http://www.megginson.com/SAX
  - write handler for start tag and for end tag

DOM - the Document Object Model

- Interface to parsed XML
- “… language-neutral…” interface (IDL)

- “With the Document Object Model, programmers can build documents, navigate their structure, and add, modify, or delete elements and content. Anything found in an HTML or XML document can be accessed, changed, deleted, or added using the Document Object Model…”

http://www.w3.org/DOM/
**DOM representation -- a tree of nodes**

```
<partdb>
  <part id="a123" units="mks">
    <name>widget</name>
    <weight>0.454</weight>
  </part>
</partdb>
```

**Node interface**

```java
public interface Node {
    ...
    public String getNodeName();
    ...
    public NodeList getChildNodes();
    ...
    public NamedNodeMap getAttributes();
    ...
}
```
public interface NodeList {
    public Node item(int index);
    public int getLength();
}

public interface NamedNodeMap {
    public Node getNamedItem(String name);
    ...
}

A common form of data extraction

Find the names and telephones of all employees in Math
Top-level traversal in DOM

```java
public class Test {
    public static void main(String args[]) throws Exception {
        Parser parser = new Parser( args[0] );
        Document doc = parser.readStream( new FileInputStream( args[0] ) );
        NodeList nodes = doc.getDocumentElement().getChildNodes();
        for (int i=0; i<nodes.getLength(); i++) {
            Node n = nodes.item(i);
            // coercion
            // select Math depts
        }
    }
}
```

Selecting the math departments

```java
NodeList ndl = n.getChildNodes();
for(int idl=0; idl<ndl.getLength(); idl++) {
    Node nd = ndl.item(idl);
    if (nd.tagName.equals("dept") &&
        (((CharacterData) nd.getFirstChild()).getData().equals("Math"))) // coercion
    {
        // inner code
        return;
    }
}
```
The inner code

```java
NodeList nnl = n.getChildNodes();
for(int ii=0; ii < nnl.getLength(); ii++)
{
  Node nn = nodes.item(i); // coercion
  if (nn.tagName = "name")
    Nodelist ncl = n.getElementsByTagName("tel");
    for(int iii = 0; iii < ncl.getLength(); iii++)
    {
      Node nc = ncl.item(iii);
      System.out.print((CharacterData) nn.firstChild).data // coercion
      System.out.println((CharacterData) nc.firstChild).data // coercion
    }
}
```

Comments on our code

- Already quite cumbersome. Compare with an “equivalent” semistructured database query:

  ```sql
  select {name: $N, tel: $T}
  where {name: $N, dept: "Math", contact-info.tel: $T} <- DB
  ```

- Code may fail wherever there is a coercion, or give “empty” results.
- Code is already inefficient (double iterations over the same set of nodes)
- Need for types !!!
Constructing data

```
<doc1>
  <employee>
    <name> John Doe </name>
    <contact-info>
      <address> ... </address>
      <tel> 123 7456 </tel>
      <email> jd@abc.edu </email>
    </contact-info>
    <dept> Math </dept>
  </employee>
  ...
</doc1>
```

```
<doc2>
  <employee>
    <name> John Doe </name>
    <tel> 123 7456 </tel>
  </employee>
  <employee>
    <name> Jane Doe </name>
    <tel> 234 5678 </tel>
  </employee>
  ...
</doc2>
```

Constructing Data using the DOM

```java
Document d = new DocumentImplementation ...
Element root = d.createElement("doc2")
// set root of document
// top level loop
{
  Element emp = d.createElement("employee")
  root.appendChild(emp)
  // innermost loop
  {
    Element name = d.createElement("name")
    // set s to appropriate character string
    name.appendChild(createCDATASection(s))
    emp.appendChild(name)
    ...
  }
  ...
}
```

All node constructors are methods of the document implementation

Could also use an existing node or a "cloned" node
A join

The names of employees and their department buildings

Implementing a Join in the DOM??

```java
nl1 = r1.getChildNodes() //r1 is root of doc1
for (int i1 = 1; i1 < n1.getLength; i1++)
{
    nl2 = r2.getChildNodes() //r2 is root of doc2
    for (int i2 = 1; i2 < n2.getLength; i2++)
        ...
}
```

- Even if we can get both documents in core, this is not the most efficient method
- If not?
- This is a typical database query!
**SAX, a low-level alternative to DOM**

- **SAX** = Simple API for XML
- Event-driven, supported by most XML parsers
- Reads a stream of tokens and triggers events, e.g.,
  - startDocument
  - startElement
  - endElement
  - endDocument
- The programmer has to write a document handler that captures these events and does something with the tokens
- Simpler than DOM: programmer must do more storage management
- Less likely than DOM to fail on large documents!!

---

**Conclusions so far**

- We need types for our XML documents, and we would like to be able to use them as a static check of our program correctness.
- We need database technology for large “documents” and -- perhaps -- for simplicity of code.
Part IV: Query Languages

Why a query language? Extracting, Restructuring, Integration, Browsing…

XQuery
http://www.w3.org/TR/xquery/
http://www.xml.com/pub/a/2002/10/16/xquery.html

XPATH (part of a query language)
http://www.w3.org/TR/xpath

XSLT
http://www.w3.org/TR/xslt
and many others….

XQuery -- likely to gain acceptance…

…and, like other things in W3C, not necessarily the best.

Ingredients:
• A concrete syntax:
  http://www.w3.org/TR/xquery
• Based on XPath: http://www.w3.org/TR/xpath.html
• A formal semantics and “algebra”:
  http://www.w3.org/TR/query-semantics/
• Some test cases: http://www.w3.org/TR/xmlquery-use-cases
XQuery: basic syntax

- FLWR: For, Let, Where, Return
- FOR: loops through each node in a sequence, binding that node to a variable
  
  For $x$ in document("company.xml")//department
- LET: binds a variable to an entire sequence of nodes
  
  Let $y := $x//employee
- WHERE: evaluates an expression each time the statement attempts to RETURN information
- Each of these clauses may use XPath expressions.

XPath: some basics

- "/" denotes traversal to a child
  
  - . /title (where "." denotes the current context)
- "/" denotes traversal to a descendant
  
  - . //book
- "[…]" denotes restrictions on a node; they can be boolean expressions (and, or) and include XPath functions
  
  
  - . //person[@id="1234"]: person with id attribute with value ‘1234’
  
  - . //person/child[1]: first child subelement of person
XPath basics, cont.

- XPath has 13 different axes of navigation, e.g. child, descendant, attribute, self, parent, ancestor, preceding, preceding-sibling, etc.
- An axis is followed by a node kind test, e.g.
  - text(), which returns the PCDATA associated with a node
  - node(), which returns the node
- For example
  - /child::bib/child::book/child::text() is the same as /bib/book/text()
  - $title/parent::node(), which returns the parent of the element bound to $title

XQuery dereferences

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

  An IDREF attribute
  Filter the resulting sequence with this name test
Address Book Revisited

<addrBook>
  <person SSN="111-22-3333">
    <name>Caesar</name>
    <greet>Caesar Imperator</greet>
    <addr>The Capitol</addr>
    <addr>Rome, OH 98765</addr>
    <tel>(321) 786 2541</tel>
    <fax>(321) 786 2542</fax>
    <tel>(321) 786 2543</tel>
    <email>jc@forum.rome.org</email>
  </person>
</addrBook>

Pattern Matching

Find Caesar’s SSN and e-mail address:

```
<XML>
  for $person in document("address.xml")/person
  let $e:=$person/email
  where $person/name='Caesar'
  return <person>{$person/@SSN} {$e}</person>
</XML>
```

We could also have written:

```
<XML>
  for $person in document("address.xml")/person[name='Caesar'], $id:=$person/@SSN
  return <person>{$id} {$person/email}</person>
</XML>
```

Result:

```
<XML>
  <person @SSN="111-22-3333">
    <email>jc@forum.rome.org</email>
  </person>
</XML>
```
Manipulating sequences

Print the all telephone numbers listed for Caesar:

```xml
<XML>
{ 
for $person in document("address.xml")//person[name='Caesar']
let $t:= $person/tel
return  <person> {$t} </person>
}
</XML>
```

Result:
```xml
<XML>
<person>
  <tel> (321) 786 2541 </tel>
  <tel> (321) 786 2543 </tel>
</person>
</XML>
```

Manipulating sequences, cont.

Print the first telephone number listed for Caesar:

```xml
<XML>
{ 
for $person in document("address.xml")//person[name='Caesar']
let $t:= $person/tel[1]
return  <person> {$t} </person>
}
</XML>
```

Result:
```xml
<XML>
<person>
  <tel> (321) 786 2541 </tel>
</person>
</XML>
```
Constructing New XML Data

Whom can we contact electronically?

```xml
<XML>
{
for $person in document("address.xml")//person,
  $g = person/greet
let $e = $person/email
where count($e) = 1
return <e-contact>
  <who>
    {$g/text()}
  </who>
  <where>
    {$e/text()}
  </where>
</e-contact>
}<XML>
```

<table>
<thead>
<tr>
<th>XML</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;e-contact&gt;</td>
<td>&lt;who&gt;Caesar Imperator&lt;/who&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;where&gt;<a href="mailto:jc@forum.rome.org">jc@forum.rome.org</a>&lt;/where&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/e-contact&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;e-contact&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;who&gt;Brutus&lt;/who&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;where&gt;<a href="mailto:mb@philippi.com">mb@philippi.com</a>&lt;/where&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/e-contact&gt;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>&lt;/e-contact&gt;</td>
</tr>
</tbody>
</table>

Winter 2003  Data Restructuring

Joins

Who of our contacts was involved in a movie?

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

Winter 2003
Joins (cont’d)

Result:
<XML>
  <cine-contact>
    <who>Caesar Imperator</who>
    <where>jc@forum.rome.org</where>
    <movie>Asterix and Cleopatra</movie>
  </cine-contact>
  ...
</XML>

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
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>

XQuery use cases

http://support.x-hive.com/xquery/index.html
Conclusions

- We need programming interfaces (e.g. DOM) as well as query languages for XML.
- XQuery builds on XPath and borrows features from many other XML query language proposals (e.g. Quilt, XML-QL, etc), and may eventually emerge as the standard for querying documents.
- XML storage and publishing?