XML- XPath and XQuery
Introduction to Databases
CompSci 316 Spring 2017

XML- XPath and XQuery
Introduction to Databases
CompSci 316 Spring 2017

Announcements (Wed., Apr. 12)
• Homework #4 due Monday, April 24, 11:55 pm
  • 4.1, 4.2, 4.3, X1 is posted
  • Please start early
  • There may be another extra credit problem

• Projects
  • keep working on them and write your final report
  • Demo in the week of April 24

• Google Cloud use?
  • If anyone is planning to use or using google cloud for HW/project, please send me an email

Relational Mapping

Approaches to XML processing

• Text files/messages
• Specialized XML DBMS
  • Tamino (Software AG), BaseX, eXist, Sedna, ...
  • Not as mature as relational DBMS
• Relational (and object-relational) DBMS
  • Middleware and/or extensions
  • IBM DB2’s pureXML, PostgreSQL’s XML type/functions...

Mapping XML to relational

• Store XML in a column
  • Simple, compact
  • CLOB (Character Large Object) type + full-text indexing, or better, special XML type + functions
  • Poor integration with relational query processing
  • Updates are expensive
• Alternatives?
  • Schema-oblivious mapping:
    well-formed XML → generic relational schema
    1. Node/edge-based mapping for graphs
    2. Interval-based mapping for trees
    3. Path-based mapping for trees
  • Schema-aware mapping:
    valid XML → special relational schema based on DTD

1. Node/edge-based: schema

• Element(eid, tag)
• Attribute(eid, attrName, attrValue) Key: (eid, attrName)
  • Attribute order does not matter
• ElementChild(eid, pos, child) Keys: (eid, pos), (child)
  • pos specifies the ordering of children
  • child references either Element(eid) or Text(tid)
• Text(tid, value)
  • tid cannot be the same as any eid
  • Need to “invent” lots of id’s
  • Need indexes for efficiency, e.g., Element(tag), Text(value)
2. Interval-based: schema

- **Element(left, right, level, tag)**
  - left is the start position of the element
  - right is the end position of the element
  - level is the nesting depth of the element
  - Key is left
- **Text(left, right, level, value)**
  - Key is left
- **Attribute(left, attrName, attrValue)**
  - Key is (left, attrName)

Where did ElementChild go?
- $e_1$ is the parent of $e_2$ iff:
  
  
  

Node/edge-based: example

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>Foundations of Databases</td>
</tr>
<tr>
<td>author</td>
<td>Hull, Abiteboul</td>
</tr>
<tr>
<td>publisher</td>
<td>Addison Wesley</td>
</tr>
<tr>
<td>year</td>
<td>1995</td>
</tr>
<tr>
<td>price</td>
<td>80.00</td>
</tr>
</tbody>
</table>

Node/edge-based: simple paths

- **//title**
  - SELECT eid FROM Element WHERE tag = 'title';
- **//section/title**
  - SELECT e2.eid FROM Element e1, ElementChild c, Element e2 WHERE e1.tag = 'section' AND e2.tag = 'title' AND e1.eid = c.eid AND c.child = e2.eid;
  - Path expression becomes joins!
    - Number of joins is proportional to the length of the path expression

Node/edge-based: complex paths

- **//bibliography/book(author=’Abiteboul’)@price**
  - SELECT a.attrValue FROM Element e1, ElementChild c1, Element e2, Attribute a WHERE e1.tag = 'bibliography' AND c1.eid = e1.eid AND c1.child = e2.eid AND e2.tag = 'book'
  - AND EXISTS (SELECT * FROM ElementChild c2, Element e3, ElementChild c3, Text t WHERE e2.eid = c2.eid AND c2.child = c3.eid AND e3.tag = 'author' AND c3.eid = c.eid AND c3.child = t.eid AND t.value = 'Abiteboul')
  - AND e2.eid = a.eid
  - AND a.attrName = 'price';

Some author of e2 is 'Abiteboul'

Node/edge-based: descendental-or-self

- **//book/title**
  - Requires SQL3 recursion
  - WITH RECURSIVE ReachableFromBook(id) AS (SELECT eid FROM Element WHERE tag = 'book')
  - UNION
  - (SELECT c.child FROM ReachableFromBook r, ElementChild c WHERE r.eid = c.eid)
  - SELECT eid FROM Element WHERE eid IN (SELECT * FROM ReachableFromBook) AND tag = 'title';

Interval-based: example

Where did ElementChild go?
- $e_1$ is the parent of $e_2$ iff:
  
  
  

4/12/17
Interval-based: queries

- \//section/title
  - SELECT e2.left
    FROM Element e1, Element e2
    WHERE e1.tag = 'section' AND e2.tag = 'title'
    AND e1.left < e2.left AND e2.right < e1.right
    AND e1.level = e2.level - 1;

  - Path expression becomes "containment" joins!
  - Number of joins is proportional to path expression length

- \//book//title
  - SELECT e2.left
    FROM Element e1, Element e2
    WHERE e1.tag = 'book' AND e2.tag = 'title'
    AND e1.left < e2.left AND e2.right < e1.right;

  - No recursion!

Summary so far

Node/edge-based vs. interval-based mapping

- Path expression steps
  - Equality vs. containment join
  - Descendent-or-self
    - Recursion required vs. not required

3. Path-based mapping

Label-path encoding: paths as strings of labels

- Element(pathid, left, right, ...), Path(pathid, path)
  - path is a string containing the sequence of labels on a path starting from the root
  - Why are left and right still needed?

<table>
<thead>
<tr>
<th>Element</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
</tbody>
</table>

Label-path encoding: queries

- Simple path expressions with no conditions
  - \//book/title
    - Perform string matching on Path
    - Join qualified pathid's with Element
  - \//book[publisher='Prentice Hall']//title
    - Evaluate //book/title
    - Evaluate //book/publisher[text()='Prentice Hall']
    - Must then ensure title and publisher belong to the same book (how?)

  - Path expression with attached conditions needs to be broken down, processed separately, and joined back

Another Path-based mapping

Dewey-order encoding

- Each component of the id represents the order of the child within its parent

<table>
<thead>
<tr>
<th>Element(dewey_pid, tag)</th>
<th>Text(dewey_pid, value)</th>
<th>Attribute(dewey_pid, attrName, attrValue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bibliography</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>book</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>title</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>author</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>author</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>author</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>publisher</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>year</td>
<td></td>
</tr>
</tbody>
</table>

Dewey-order encoding: queries

- Examples:
  - //title
  - //section/title
  - //book/title
  - //book[publisher='Prentice Hall']//title
  - Works similarly as interval-based mapping
    - Except parent/child and ancestor/descendant relationship are checked by prefix matching
An overview of XSLT, SAX, and DOM

XSLT
- XML-to-XML rule-based transformation language
  - Used most frequently as a stylesheet language
  - An XSLT program is an XML document itself

XSLT program
- An XSLT program is an XML document containing
  - Elements in the <xsl>: namespace
  - Elements in user namespace
- Roughly, result of evaluating an XSLT program on an input XML document = the XSLT document where each <xsl>: element is replaced with the result of its evaluation
- Basic ideas
  - Templates specify how to transform matching input nodes
  - Structural recursion applies templates to input trees recursively
  - Uses XPath as a sub-language

XSLT elements
- Element describing transformation rules
  - <xsl:template>
- Elements describing rule execution control
  - <xsl:apply-templates>
  - <xsl:call-template>
- Elements describing instructions
  - <xsl:if>, <xsl:for-each>, <xsl:sort>, etc.
- Elements generating output

XSLT example
- Find titles of books authored by “Abiteboul”

```xml
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="2.0">
  <xsl:template match="book[author='Abiteboul']">
    <booktitle>
      <xsl:value-of select="title"/>
    </booktitle>
  </xsl:template>
</xsl:stylesheet>
```
- Not quite; we will see why later

<xsl:template>
- <xsl:template match="book[author='Abiteboul']">
  - <booktitle>
  - <xsl:value-of select="title"/>
  - </booktitle>
  </xsl:template>
- • <xsl:template match="match_expr"> is the basic XSLT construct describing a transformation rule
  - match_expr is an XPath-like expression specifying which nodes this rule applies to
  - <xsl:value-of select="xpath_expr"> evaluates xpath_expr within the context of the node matching the template, and converts the result sequence to a string
  - <booktitle> and </booktitle> simply get copied to the output for each node matched
Template in action

```xml
<xsl:template match="book[author='Abiteboul']">
  <booktitle>
    <xsl:value-of select="title"/>
  </booktitle>
</xsl:template>
```

- Example XML fragment
```
<book ISBN="#10" price="#80.00">
  <title>Foundations of Databases</title>
  <author>Abiteboul</author>
  <author>Hull</author>
  <author>Vianu</author>
  <publisher>Addison Wesley</publisher>
  <year>1995</year>
  <section>…</section>…
</book>
```

- Template applies
```
<booktitle>
  Foundations of Databases
</booktitle>
```

- Template does not apply; default behavior is to process the node recursively and print all text nodes
```
A First Course in Databases
Ullman
Widom
Prentice-Hall
2002
… …
```

Removing the extra output

- Add the following template:
```
<xsl:template match="text()|@*"/>
```

- This template matches all text and attributes

- XPath features
  - `text()` is a node test that matches any text node
  - `@*` matches any attribute
  - `|` means “or” in XPath

- Body of the rule is empty, so all text and attributes become empty string
```
This rule effectively filters out things not matched by the other rule
```

Other features of XSLT

- Loop and condition
- White space control, insertion of newline
- Calling templates with parameters
- Debugging and exiting the program
```
<xsl:message>, <xsl:message terminate="yes">
```

- Defining variables, keys, functions

SAX & DOM

- Both are API’s for XML processing
- SAX (Simple API for XML)
  - Started out as a Java API, but now exists for other languages too
- DOM (Document Object Model)
  - Language-neutral API with implementations in Java, C++, python, etc.

SAX processing model

- Serial access
  - XML document is processed as a stream
  - Only one look at the data
  - Cannot go back to an early portion of the document
- Event-driven
  - A parser generates events as it goes through the document (e.g., start of the document, end of an element, etc.)
  - Application defines event handlers that get invoked when events are generated

A simple SAX example

- Print out text contents of title elements
```
import sys
import xml.sax
from StringIO import StringIO
class PathHandler(xml.sax.ContentHandler):
  def startDocument(self):
      …
  def startElement(self, name, attrs):
      …
xml.sax.parse(sys.stdin, PathHandler())
```
SAX events

Most frequently used events:
- `startDocument`
- `endDocument`
- `startElement`
- `endElement`
- `characters`

- Whenever the parser has processed a chunk of character data (without generating other kinds of events)
- Warning: The parser may generate multiple characters events for one piece of text

DOM processing model

- XML is parsed by a parser and converted into an in-memory DOM tree
- DOM API allows an application to
  - Construct a DOM tree from an XML document
  - Traverse and read a DOM tree
  - Construct a new, empty DOM tree from scratch
  - Modify an existing DOM tree
  - Copy subtrees from one DOM tree to another etc.

A simple SAX example (cont’d)

```python
def startDocument(self):
    self.outBuffer = None

def startElement(self, name, attrs):
    if name == 'title':
        self.outBuffer = StringIO()

def endElement(self, name):
    if name == 'title':
        print self.outBuffer.getvalue()
        self.outBuffer = None

def characters(self, content):
    if self.outBuffer is not None:
        self.outBuffer.write(content)
```

Summary

- XML data can be “shredded” into rows in a relational database
- XQueries can be translated into SQL queries
  - Queries can then benefit from smart relational indexing, optimization, and execution
- With schema-oblivious approaches, comprehensive XQuery-SQL translation can be easily automated
  - Different data mapping techniques lead to different styles of queries
- Schema-aware translation is also possible and potentially more efficient, but automation is more complex