XML-XQuery and Relational Mapping

Introduction to Databases

CompSci 316 Spring 2017

DUKE
COMPUTER SCIENCE

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

Today

- Finish XML
  - XQuery
  - Relational mapping from XML
  - An overview of XSLT, SAX, DOM (if we have time)

- Remaining lectures
  - Advanced topics
  - Parallel, distributed databases, Map-Reduce, NOSQL
  - Data mining and data warehousing
  - …

A tricky XPath example

- Suppose for a moment that price is a child element of book, and there may be multiple prices per book
- Books with some price in range [20, 50]
  - Wrong answer:
    /bibliography/book
    [price >= 20 and price <= 50]
  - Correct answer:
    /bibliography/book
    [price[. >= 20 and . <= 50]]

Review: XQuery
A simple XQuery based on XPath

Find all books with price lower than $50

```
<result>{
  doc("bib.xml")/bibliography/book[@price<50]
}</result>
```

- Things outside {}’s are copied to output verbatim
- Things inside {}’s are evaluated and replaced by the results
  - `doc("bib.xml")` specifies the document to query
    - Can be omitted if there is a default context document
  - The XPath expression returns a sequence of book elements
  - These elements (including all their descendants) are copied to output

Review: FLWR expressions

- Retrieve the titles of books published before 2000, together with their publisher

```
<result>{
  for $b in doc("bib.xml")/bibliography/book
  let $p := $b/publisher
  where $b/year < 2000
  return
    <book>
      { $b/title }
      { $p }
    </book>
}</result>
```

- for: loop
  - $b ranges over the result sequence, getting one item at a time
- let: "assignment"
  - $p gets the entire result of $b/publisher (possibly many nodes)
  - let isn't really assignment, but simply creates a temporary binding
- where: filtering by condition
- return: result structuring
  - Invoked in the "innermost loop," i.e., once for each successful binding of all query variables that satisfies where

Review: An equivalent formulation

- Retrieve the titles of books published before 2000, together with their publisher

```
<result>{
 for $b in doc("bib.xml")/bibliography/book[year<2000]
 return
   <book>
     { $b/title }
     { $b/publisher }
   </book>
}</result>
```

End of Lecture 22

Another formulation

- Retrieve the titles of books published before 2000, together with their publisher

```
<result>{
  for $b in doc("bib.xml")/bibliography/book,
      $p in $b/publisher
  where $b/year < 2000
  return
    <book>
      { $b/title }
      { $p }
    </book>
}</result>
```

} Nested loop

- Is this query equivalent to the previous two?
- Yes, if there is one publisher per book
- No, in general
  - Two result book elements will be created for a book with two publishers
  - No result book element will be created for a book with no publishers

Yet another formulation

- Retrieve the titles of books published before 2000, together with their publisher

```
<result>{
 let $b := doc("bib.xml")/bibliography/book
 where $b/year < 2000
 return
   <book>
     { $b/title }
     { $b/publisher }
   </book>
}</result>
```

- Is this query correct?
- No!
- It will produce only one output book element, with all titles clumped together and all publishers clumped together
- All books will be processed (as long as one is published before 2000)

Subqueries in return

- Extract book titles and their authors; make title an attribute and rename author to writer

```
<bibliography>{
 for $b in doc("bib.xml")/bibliography/book
 return
   <book title="{normalize-space($b/title)}">{
     for $a in $b/author
     return <writer>{string($a)}</writer>
   }</book>
}</bibliography>
```

  - normalize-space(*string*) removes leading and trailing spaces from string, and replaces all internal sequences of white spaces with one white space

An explicit join

- Find pairs of books that have common author(s)

```
<result>{
 for $b1 in doc("bib.xml")//book
 for $b2 in doc("bib.xml")//book
 where $b1/author = $b2/author
   and $b1/title > $b2/title
 return
  <pair>
   {$b1/title}
   {$b2/title}
  </pair>
}</result>
```

← These are string comparisons,
not identity comparisons!

Existentially quantified expressions

(some $*var* in *collection* satisfies *condition*)

  - Can be used in where as a condition

- Find titles of books in which XML is mentioned in some section

```
<result>{
  for $b in doc("bib.xml")//book
  where (some $section in $b//section satisfies
          contains(string($section), "XML"))
  return $b/title
}</result>
```

Universally quantified expressions

(every $*var* in *collection* satisfies *condition*)

  - Can be used in where as a condition

- Find titles of books in which XML is mentioned in every section

```
<result>{
  for $b in doc("bib.xml")//book
  where (every $section in $b//section satisfies
         contains(string($section), "XML"))
  return $b/title
}</result>
```

Aggregation

- List each publisher and the average prices of all its books

```
<result>{
  for $pub in distinct-values(doc("bib.xml")//publisher)
  let $price := avg(doc("bib.xml")//book[publisher=$pub]/@price)
  return
    <publisherpricing>
      <publisher>{$pub}</publisher>
      <avgprice>{$price}</avgprice>
    </publisherpricing>
}</result>
```

- distinct-values(*collection*) removes duplicates by value
  - If the collection consists of elements (with no explicitly declared types), they are first converted to strings representing their "normalized contents"
- avg(*collection*) computes the average of *collection* (assuming each item in *collection* can be converted to a numeric value)

Conditional expression

- List each publisher and, only if applicable, the average prices of all its books

```
<result>{
 for $pub in distinct-values(doc("bib.xml")//publisher)
 let $price := avg(doc("bib.xml")//book[publisher=$pub]/@price)
 return
  <publisherpricing>
   <publisher>{$pub}</publisher>
   { if ($price)
    then <avgprice>{$price}</avgprice>
    else () }
  </publisherpricing>
}</result>
```

Empty list ≈ nothing

- Use anywhere you’d expect a value, e.g.:
  - `let $foo := if (…) then … else …`
  - `return <bar blah="{ if (…) then … else … }"/>`

Sorting with “order by”

Replaces “sort by” used earlier since August 2002 (http://www.w3.org/TR/2002/WD-xquery-20020816/)

Since June 2006

- A new order by clause is added to FLWR
  - Which now becomes FLWOR

- Example: list all books in order by price from high to low; for books with the same price, sort by first author and then title

```
<result>{
  for $b in doc("bib.xml")//book[@price>100]
  stable order by
    number($b/price) descending,
    $b/author[1],
    $b/title empty least
  return $b
}</result>
```

Preserve input order

Order as number, not string

Override default (ascending)

Empty value considered smallest

Summary

- Many, many more features not covered in class
- XPath is very mature, stable, and widely used
  - Has good implementations in many systems
  - Is used in many other standards
- XQuery is also fairly popular
  - Has become the SQL for XML
  - Has good implementations in some systems

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: ← *Focus of this lecture*
    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*)

Node/edge-based: example

```
<bibliography>
 <book ISBN="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>
 </book>…
</bibliography>
```

Attribute

eid	*attrName*	*attrValue*
e1	ISBN	ISBN-10
e1	price	80

Text

tid	*value*
t0	Foundations of Databases
t1	Abiteboul
t2	Hull
t3	Vianu
t4	Addison Wesley
t5	1995

Element

eid	*tag*
e0	bibliography
e1	book
e2	title
e3	author
e4	author
e5	author
e6	publisher
e7	year

ElementChild

eid	*pos*	*child*
e0	1	e1
e1	1	e2
e1	2	e3
e1	3	e4
e1	4	e5
e1	5	e6
e1	6	e7
e2	1	t0
e3	1	t1
e4	1	t2
e5	1	t3
e6	1	t4
e7	1	t5

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 e1.eid = c1.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 = e3.eid
                AND e3.tag = 'author'
                AND e3.eid = c3.eid AND c3.child = t.tid
                AND t.value = 'Abiteboul')
    AND e2.eid = a.eid
    AND a.attrName = 'price';
    ```

some author of e2
is ’Abiteboul’

Node/edge-based: descendent-or-self

- //book//title
  - Requires SQL3 recursion

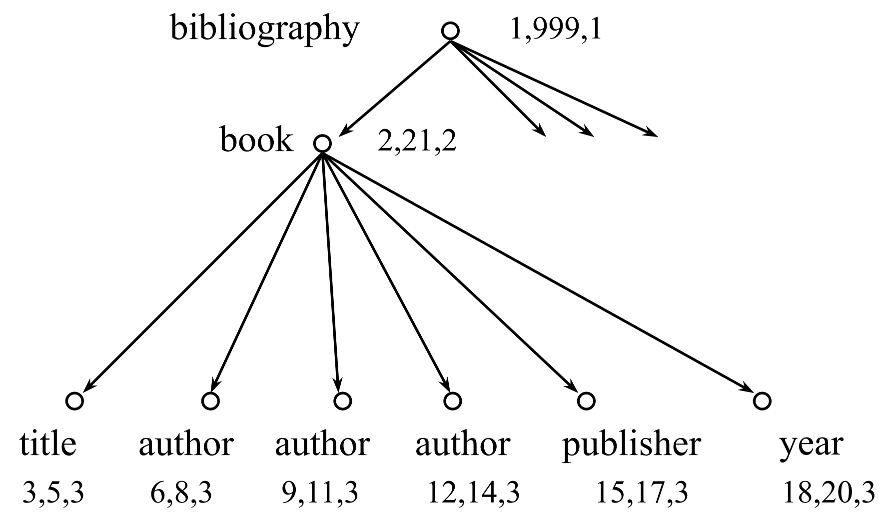
  -
    ```
    WITH RECURSIVE ReachableFromBook(id) AS
    ((SELECT eid FROM Element WHERE tag = 'book')
     UNION
     (SELECT c.child
      FROM ReachableFromBook r, ElementChild c
      WHERE r.eid = c.eid))
    SELECT eid
    FROM Element
    WHERE eid IN (SELECT * FROM ReachableFromBook)
    AND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

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*)

Interval-based: example

```
1<bibliography>
  2<book ISBN="ISBN-10" price="80.00">
    3<title>4Foundations of Databases</title>5
    6<author>7Abiteboul</author>8
    9<author>10Hull</author>11
   12<author>13Vianu</author>14
   15<publisher>16Addison Wesley</publisher>17
   18<year>191995</year>20
  </book>21…
</bibliography>999
```



☞Where did *ElementChild* go?

- e_1 is the parent of e_2 iff:

  $[e_1.left, e_1.right] \supset [e_2.left, e_2.right]$, and $e_1.level = e_2.level - 1$

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?

Element

pathid	*left*	*right*	…
1	1	999	…
2	2	21	…
3	3	5	…
4	6	8	…
4	9	11	…
4	12	14	…
…	…	…	…

Path

pathid	*path*
1	/bibliography
2	/bibliography/book
3	/bibliography/book/title
4	/bibliography/book/author
…	…

Label-path encoding: queries

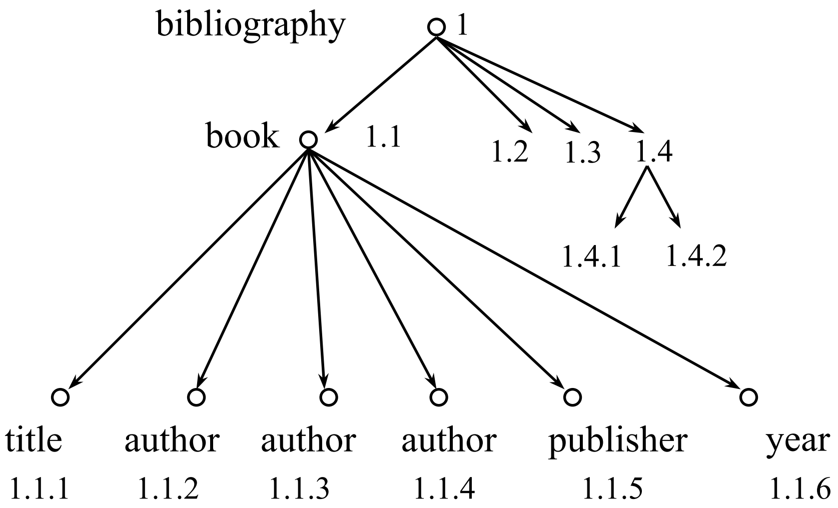
- Simple path expressions with no conditions
  //book//title
  - Perform string matching on *Path*
  - Join qualified *pathid*’s with *Element*
- //book[publisher='Prentice Hall']/title
  - Evaluate //book/title
  - Evaluate //book/publisher[text()='Prentice Hall']
  - Must then ensure title and publisher 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



Element(<u>dewey_pid</u>, tag)
Text(<u>dewey_pid</u>, value)
Attribute(<u>dewey_pid</u>, <u>attrName</u>, attrValue)

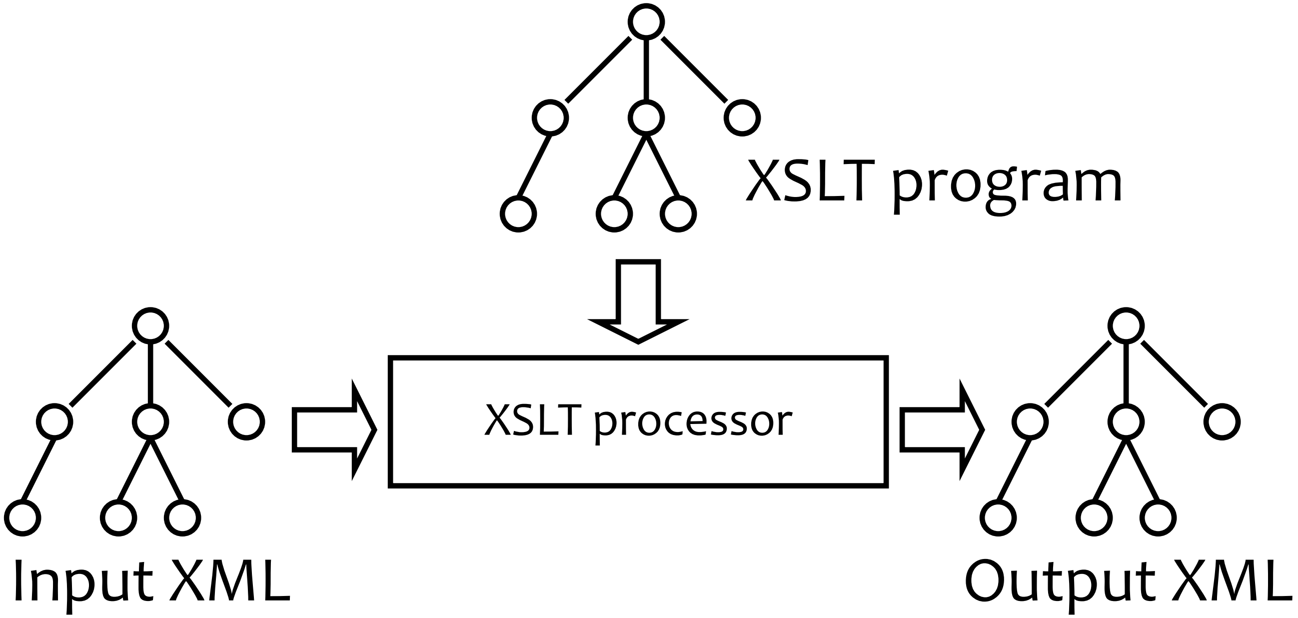
Dewey-order encoding: queries

- 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



Actually, output does not need to be in XML in general

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
  - <xsl:value-of>, <xsl:copy-of>, <xsl:element>, <xsl:attribute>, <xsl:text>, etc.

XSLT example

- Find titles of books authored by “Abiteboul”

```
<?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>
```

Standard header of an XSLT document

- 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

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

- Example XML fragment

```
<book ISBN="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>
<book ISBN="ISBN-20" price="40.00">
  <title>A First Course in Databases</title>
  <author>Ullman</author>
  <author>Widom</author>
  <publisher>Prentice-Hall</publisher>
  <year>2002</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

```
<?xml version="1.0"?>                       ⟶ startDocument
<bibliography>                  ⟶ startElement
  <book ISBN="ISBN-10" price="80.00">  ⟶ startElement
    <title>Foundations of Databases</title>
    …                   ↳ startElement
  </book>           ⟶ endElement
  …
</bibliography>                 ⟶ endElement
                                ↳ endDocument
```

Whitespace may come up as characters or ignorableWhitespace, depending on whether a DTD is present

A simple SAX example (cont’d)

```
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)
```

Tag name

A map from attribute names to values

Characters read

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.

Summary

- XPath
  - Powerful and building block to other query forms
- XQuery
  - SQL-Like query for XML
- Relational mapping
  - 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
- XSLT
  - stylesheet like language
- SAX and DOM
  - Parsing XML