Introduction to Data Management
CSE 344

Lecture 14: XPath, XQuery
Announcements

• Homework 4 due tonight, 11pm
• Homework 5 due next Thursday, 11 pm

• Midterm next Monday, in MEB 238
  – Review this Sunday
  – Past exams are up on the web
  – You are allowed 1 sheet of notes

• Readings for today’s lecture: sec. 11.1-11.3, 12.1-12.2
Today

• XML schema

• Querying XML
  – XPath
  – XQuery
XML Schema
Document Type Definitions (DTD)

- An XML document may have a DTD
- XML document:
  - **Well-formed** = if tags are correctly closed
  - **Valid** = if it has a DTD and conforms to it
- Validation is useful in data exchange
  - Use [http://validator.w3.org/check](http://validator.w3.org/check) to validate

Superseded by XML Schema (Book Sec. 11.4)
- Very complex: DTDs still used widely
Example DTD

```
<!DOCTYPE company [ 
  <!ELEMENT company  ((person|product)*)> 
  <!ELEMENT person  (ssn, name, office, phone?)> 
  <!ELEMENT ssn        (#PCDATA)> 
  <!ELEMENT name       (#PCDATA)> 
  <!ELEMENT office     (#PCDATA)> 
  <!ELEMENT phone      (#PCDATA)> 
  <!ELEMENT product    (pid, name, description?)> 
  <!ELEMENT pid        (#PCDATA)> 
  <!ELEMENT description (#PCDATA)> 
]> 
```
Example DTD

Example of valid XML document:

```xml
<company>
  <person>
    <ssn>123456789</ssn>
    <name>John</name>
    <office>B432</office>
    <phone>1234</phone>
  </person>

  <person>
    <ssn>987654321</ssn>
    <name>Jim</name>
    <office>B123</office>
  </person>

  <product>...</product>

  ...
</company>
```
DTD: The Content Model

- Content model:
  - Complex = a regular expression over other elements
  - Text-only = #PCDATA
  - Empty = EMPTY
  - Any = ANY
  - Mixed content = (#PCDATA | A | B | C)*

<!ELEMENT tag (CONTENT)>
DTD: Complex Content

Sequence

<!ELEMENT name (firstName, lastName)>

Optional

<!ELEMENT name (firstName?, lastName)>

Kleene star

<!ELEMENT person (name, phone*)>

Alternation

<!ELEMENT person (name, (phone|email))>

XML

<name>
  <firstName> . . . . . </firstName>
  <lastName> . . . . . </lastName>
</name>

<person>
  <name> . . . . . </name>
  <phone> . . . . . </phone>
  <phone> . . . . . </phone>
  <phone> . . . . . </phone>
  <phone> . . . . . </phone>
</person>
DTD: Attributes

From “sample-xml-with-dtd.xml”

```xml
<!DOCTYPE bib [
  <!ELEMENT bib (book* )>
  <!ELEMENT book (title, (author+ | editor+ ), publisher?, price )>
  <!ATTLIST book year CDATA #REQUIRED >
  ...
]

<bib>
  <book year="1994">
    ...
  </book>

```
DTD: Text

Two options:

• **#PCDATA** ("Parsed Character Data") = the text inside elements
• **CDATA** ("Character Data") = the text inside attributes
• There is no #CDATA and no PCDATA
Querying
Querying XML Data

• **XPath** = simple navigation
• **XQuery** = the SQL of XML
  – Every XPath expression is also an XQuery, but XQuery can do (much) more

• **XSLT** = recursive traversal
  – will not discuss in class
Sample Data for Queries

```xml
<bib>
  <book>
    <publisher> Addison-Wesley </publisher>
    <author> Serge Abiteboul </author>
    <author> <first-name> Rick </first-name> <last-name> Hull </last-name> </author>
    <author> Victor Vianu </author>
    <title> Foundations of Databases </title>
    <year> 1995 </year>
  </book>
  <book price="55">
    <publisher> Freeman </publisher>
    <author> Jeffrey D. Ullman </author>
    <title> Principles of Database and Knowledge Base Systems </title>
    <year> 1998 </year>
  </book>
</bib>
```
Data Model for XPath

XPath returns a sequence of items. An item is either:

- A value of primitive type, or
- A node (doc, element, or attribute)
XPath: Simple Expressions

\[ /\text{bib/book/year} \]

Result:  \(<\text{year}> 1995 </\text{year}>\)
         \(<\text{year}> 1998 </\text{year}>\)

\[ /\text{bib/paper/year} \]

Result:  empty (there were no papers)

What’s the difference?
XPath: Restricted Kleene Closure

//author

Result: <author> Serge Abiteboul </author>
  <author> <first-name> Rick </first-name> 
  <last-name> Hull </last-name>
</author>

<author> Victor Vianu </author>
<author> Jeffrey D. Ullman </author>

/bib//first-name

Result: <first-name> Rick </first-name>
XPath: Attribute Nodes

/bib/book/@price

Result: “55”

@price means that price has to be an attribute
XPath: Wildcard

//author/*

Result: <first-name> Rick </first-name>  
<last-name> Hull </last-name>

* Matches any element
@* Matches any attribute
XPath: Text Nodes

Result: Serge Abiteboul
        Victor Vianu
        Jeffrey D. Ullman

Rick Hull doesn’t appear because he has first-name, last-name

Functions in XPath:
- `text()` = matches the text value
- `node()` = matches any node (= * or @* or `text()`)
- `name()` = returns the name of the current tag
XPath: Predicates

/bib/book/author[first-name]

Result:  
<author>  
  <first-name> Rick </first-name>  
  <last-name> Hull </last-name>  
</author>
XPath: More Predicates

/\bib/\book/\author[first-name][address[./zip][city]]/last-name

Result: <last-name> … </last-name>
   <last-name> … </last-name>

How do we read this?
First remove all qualifiers (predicates):

/\bib/\book/\author/last-name

Then add them one by one:

/\bib/\book/\author[first-name][address]/last-name
XPath: More Predicates

/bib/book[@price < 60]

/bib/book[author/@age < 25]

/bib/book[author/text()]
XPath: Position Predicates

- `/bib/book[last()]`: The last book
XPath: More Axes

. means *current node*  
\[ /\text{bib/book}[./\text{review}] \]

\[ /\text{bib/book}[./\text{review}] \]  
Same as  
\[ /\text{bib/book}[\text{review}] \]

\[ /\text{bib/author}/../\text{first-name} \]  
Same as  
\[ /\text{bib/author/first-name} \]
XPath: More Axes

.. means parent node

`/bib/author/.. /author/zip`  
Same as  
`/bib/author/zip`

`/bib/book[.//review/..//comments]`

Same as

`/bib/book[.//*[comments][review]]`

Hint: don’t use ..
A Few Extra Examples

Run these examples on the sample xml posted on course website
(Instructions for running queries in HW5)

Each line is a separate example:

- doc("sample-xml.xml")//book/price
- doc("sample-xml.xml")//book[editor]/price
- doc("sample-xml.xml")//book[price/text() > 100]/title
XPath: Summary

- `bib` matches a `bib` element
- `*` matches any element
- `/` matches the root element
- `/bib` matches a `bib` element under root
- `bib/paper` matches a `paper` in `bib`
- `bib//paper` matches a `paper` in `bib`, at any depth
- `//paper` matches a paper at any depth
- `paper|book` matches a `paper` or a `book`
- `@price` matches a `price` attribute
- `bib/book/@price` matches price attribute in book, in `bib`
- `bib/book[@price<"55"]/author/last-name` matches...
- `bib/book[@price<"55" or @price>"99"]/author/last-name` matches...
XQuery
XQuery

- Standard for high-level querying of databases containing data in XML form
- Based on Quilt, which is based on XML-QL
- Uses XPath to express more complex queries
  - Every XPath expression is itself a (simple) XQuery or can be part of a more complex query

- Reference: sec. 12.2
FLWR ("Flower") Expressions

FOR ...  
LET...  
WHERE...  
RETURN...

- Zero or more
- Zero or more
- Zero or one
- Exactly one
FOR-WHERE-RETURN

Find all book titles published after 1995:

```xml
FOR $x$ IN doc("bib.xml")/bib/book
WHERE $x$/year/text() > 1995
RETURN $x$/title
```

Result:
```
<title>
  Principles of Database and Knowledge Base Systems
</title>
```
FOR-WHERE-RETURN

Equivalently

RETURN $x$

And even shorter:

COERCION

The query:

```
FOR $x IN doc("bib.xml")/bib/book[year > 1995] /title
RETURN $x
```

Is rewritten by the system into:

```
RETURN $x
```
FOR-WHERE-RETURN

• Find all book titles and the year when they were published:

FOR $x$ IN doc("bib.xml")/ bib/book
RETURN <answer>
  <title>{ $x/title/text() } </title>
  <year>{ $x/year/text() } </year>
</answer>

Result:
<answer> <title> Foundations … </title> <year> 1995 </year> </answer>
<answer> <title> Principles … </title> <year> 1998 </year> </answer>
FOR-WHERE-RETURN

Lesson: Always put `{ }` around Xpath expressions

- Notice the use of `{“ and “}`
- What is the result without them?

```
FOR $x IN doc("bib.xml")/ bib/book
RETURN <answer>
    <title> $x/title/text() </title>
    <year> $x/year/text() </year>
</answer>
```

```xml
<answer> <title> $x/title/text() </title> <year> $x/year/text() </year> </answer>
<answer> <title> $x/title/text() </title> <year> $x/year/text() </year> </answer>
```

CSE 344 - Winter 2015
Nesting

• For each author of a book by Freeman, list all books he/she published:

```xml
FOR $b IN doc("bib.xml")/bib,
  $a IN $b/book[publisher/text()="Freeman"]/author
RETURN <result>
  { $a,
    FOR $t IN $b/book[author/text()=$a/text()]/title
    RETURN $t
  }
</result>
```

In the **RETURN** clause comma concatenates XML fragments
Result

<result>
  <author>Jeffrey D. Ullman</author>
  <title>Principles of Database …</title>
</result>
Aggregates

Find all books with more than 3 authors:

```
FOR $x IN doc("bib.xml")/bib/book
WHERE count($x/author)>3
RETURN $x
```

count = a function that counts
avg = computes the average
sum = computes the sum
distinct-values = eliminates duplicates
Aggregates

Same thing:

FOR $x$ IN doc("bib.xml")/bib/book[count(author)>3]
RETURN $x$
Eliminating Duplicates

Print all authors:

```
FOR $a IN distinct-values($b/book/author/text())
RETURN <author> { $a } </author>
```

Note: distinct-values applies ONLY to values, NOT elements
The LET Clause

Find books whose price is larger than average:

FOR $b$ in doc("bib.xml")/bib
LET $a := \text{avg}(\text{\$b/book/price/text()})$
FOR $x$ in $b/book$
WHERE $x/price/text() > \$a$
RETURN $x$

LET enables us to declare variables
SQL and XQuery Side-by-side

Product(pid, name, maker, price)  Find all product names, prices, sort by price

SQL

```
SELECT x.name,  
      x.price  
FROM Product x  
ORDER BY x.price
```

XQuery

```
FOR $x in doc("db.xml")/db/Product/row  
ORDER BY $x/price/text()  
RETURN <answer>  
    { $x/name, $x/price }  
</answer>
```
XQuery’s Answer

<answer>
  <name> abc </name>
  <price> 7 </price>
</answer>

<answer>
  <name> def </name>
  <price> 23 </price>
</answer>

... Notice: this is NOT a well-formed document!
(WHY ???)
Producing a Well-Formed Answer

<myQuery>
{  FOR $x$ in doc("db.xml")/db/Product/row
    ORDER BY $x$/price/text()
    RETURN <answer>
      {  $x$/name,  $x$/price }
    </answer>
  }
</myQuery>
XQuery’s Answer

<myQuery>
  <answer>
    <name> abc </name>
    <price> 7 </price>
  </answer>
  <answer>
    <name> def </name>
    <price> 23 </price>
  </answer>
  ....
</myQuery>

Now it is well-formed!
SQL and XQuery Side-by-side

Product(pid, name, maker, price)
Company(cid, name, city, revenues)

Find all products made in Seattle

```sql
SELECT x.name
FROM Product x, Company y
WHERE x.maker=y.cid
and y.city="Seattle"
```

```xquery
FOR $r in doc("db.xml")/db,
   $x in $r/Product/row,
   $y in $r/Company/row
WHERE
   $x/maker/text()=$y/cid/text()
and $y/city/text() = "Seattle"
RETURN { $x/name }
```

```xquery
FOR $y in /db/Company/row[city/text()="Seattle"],
   $x in /db/Product/row[maker/text()=$y/cid/text()]
RETURN { $x/name }
```

SQL

XQuery

Cool XQuery
<product>
  <row> <pid> 123 </pid>
    <name> abc </name>
    <maker> efg </maker>
  </row>
  <row> .... </row>
  ...
</product>
<product>
  ....
</product>
  ....
SQL and XQuery Side-by-side

For each company with revenues < 1M count the products over $100

SELECT y.name, count(*)
FROM Product x, Company y
WHERE x.price > 100 and x.maker=y.cid and y.revenue < 1000000
GROUP BY y.cid, y.name

FOR $r in doc("db.xml")/db,
    $y in $r/Company/row[revenue/text()<1000000]
RETURN
    <proudCompany>
        <companyName>{ $y/name/text() } </companyName>
        <numberOfExpensiveProducts>
            { count($r/Product/row[maker/text()=$y/cid/text()][price/text() > 100]) }
        </numberOfExpensiveProducts>
    </proudCompany>
SQL and XQuery Side-by-side

Find companies with at least 30 products, and their average price

```
SELECT y.name, avg(x.price)
FROM Product x, Company y
WHERE x.maker=y.cid
GROUP BY y.cid, y.name
HAVING count(*) > 30
```

```
FOR $r in doc("db.xml")/db,
   $y in $r/Company/row
LET $p := $r/Product/row[maker/text()=$y/cid/text()]
WHERE count($p) > 30
RETURN
   <theCompany>
      <companyName> { $y/name/text() } </companyName>
      <avgPrice> { avg($p/price/text()) } </avgPrice>
   </theCompany>
```
XML Summary

• Stands for eXtensible Markup Language
  1. Advanced, **self-describing file format**
  2. Based on a flexible, **semi-structured data model**

• Query languages for XML
  – XPath
  – XQuery