Lecture 4
XML/Xpath/XQuery

Tuesday, January 30, 2007

XML Outline

- XML
  - Syntax
  - Semistructured data
  - DTDs
- Xpath
- XQuery

Additional Readings on XML

- http://www.w3.org/XML/
  - Main source on XML, but hard to read

- http://www.w3.org/XML/1999/XML-in-10-points
- www.zvon.org/xxl/XMLTutorial/General/book_en.html
  - One of many mini-tutorials
- http://www.w3.org/TR/xquery/
  - Strongly recommended reading on Xquery

Note: XML/XQuery is NOT covered in the textbook

Additional Readings on XPath/XQuery

- http://www.galaxquery.org/
  - The semi-official XQuery implementation;
  - we will use it in HW3;
  - easy to download and use
  - fully standard compliant
- http://www.xmlportfolio.com/xquery.html
  - Explains XQuery to XSLT programmers

XML

- A flexible syntax for data
- Used in:
  - Configuration files, e.g. Web.Config
  - Replacement for binary formats (MS Word)
  - Document markup: e.g. XHTML
  - Data: data exchange, semistructured data
- Roots: SGML - a very nasty language

We will study only XML as data

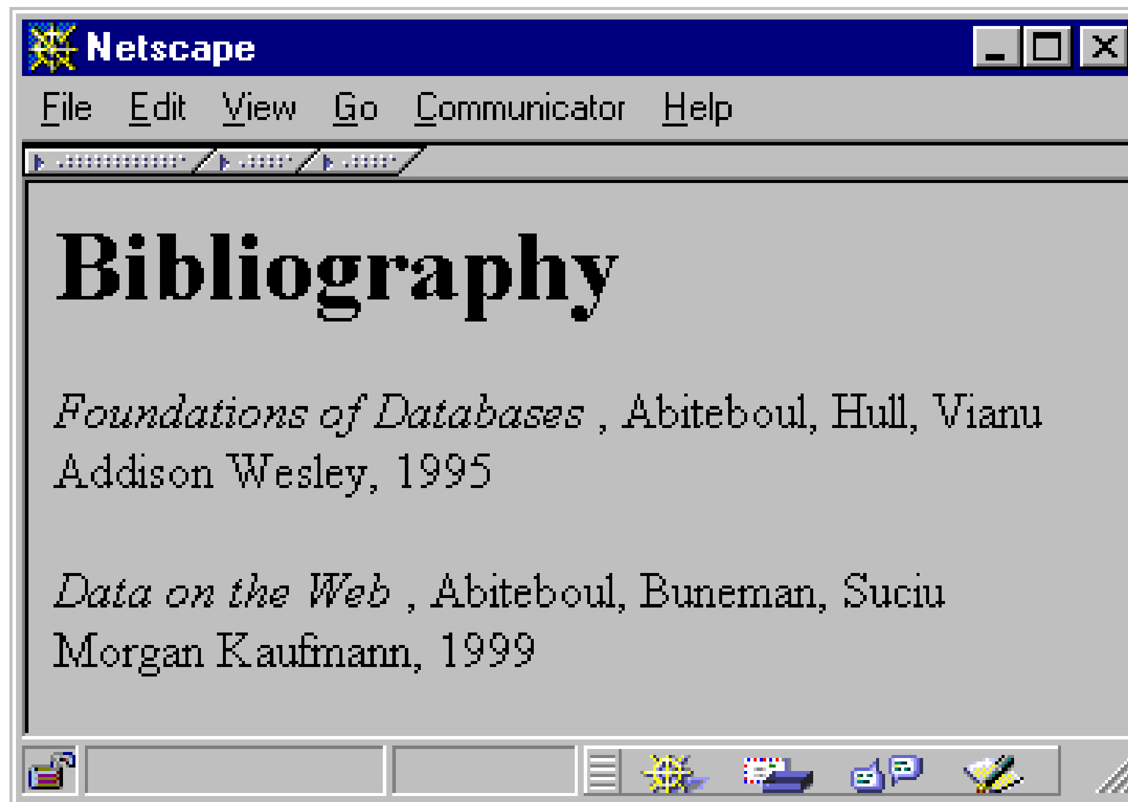
XML for Data Exchange

- Relational data does not have a syntax
  - I can't "give" you my relational database
  - Examples of syntaxes: CSV (comma-separated-values), ASN.1
- XML = syntax for data
  - But XML is not relational: *semistructured*
- Usage:
  - Export: Database → XML
  - Transport/transform XML
  - Import: XML → Databases or application

XML as Semistructured Data

- Relational databases have rigid schema
  - Schema evolution is costly
- XML is flexible: semistructured data
  - Store data in XML
- Warning: not normal form ! Not even 1NF

From HTML to XML



HTML describes the presentation

HTML

```
<h1> Bibliography </h1>
<p> <i> Foundations of Databases </i>
        Abiteboul, Hull, Vianu
        <br> Addison Wesley, 1995
<p> <i> Data on the Web </i>
        Abiteoul, Buneman, Suciu
        <br> Morgan Kaufmann, 1999
```

XML Syntax

```
<bibliography>
     <book>   <title> Foundations… </title>
              <author> Abiteboul </author>
              <author> Hull </author>
              <author> Vianu </author>
              <publisher> Addison Wesley </publisher>
              <year> 1995 </year>
     </book>
     …
</bibliography>
```

XML describes the content

XML Terminology

- tags: book, title, author, …
- start tag: <book>, end tag: </book>
- elements: <book>…</book>,<author>…</author>
- elements are nested
- empty element: <red></red> abbrv. <red/>
- an XML document: single *root element*

well formed XML document: if it has matching tags

More XML: Attributes

```
<book price = “55” currency = “USD”>
  <title> Foundations of Databases </title>
  <author> Abiteboul </author>
   …
  <year> 1995 </year>
</book>
```

Attributes v.s. Elements

```
<book price = “55” currency = “USD”>
   <title> Foundations of DBs </title>
   <author> Abiteboul </author>
    …
   <year> 1995 </year>
</book>
```

```
<book>
  <title> Foundations of DBs </title>
   <author> Abiteboul </author>
    …
   <year> 1995 </year>
   <price> 55 </price>
   <currency> USD </currency>
</book>
```

attributes are alternative ways to represent data

Comparison

Elements	**Attributes**
Ordered	Unordered
May be repeated	Must be unique
May be nested	Must be atomic

XML v.s. HTML

- What are the differences between XML and HTML ?

  In class

More XML: Oids and References

```
<person id="o555">
    <name> Jane </name>
</person>

<person id="o456">
    <name> Mary </name>
    <mother idref="o555"/>
</person>
```

Are just keys/ foreign keys design by someone who didn't take 444

Don't use them: use your own foreign keys instead.

oids and references in XML are just syntax

More XML: CDATA Section

- Syntax: <![CDATA[.....any text here...]]>

- Example:

```
<example>
   <![CDATA[ some text here </notAtag> <>]]>
</example>
```

More XML: Entity References

- Syntax: &entityname;
- Example:
  <element> this is less than < </element>
- Some entities:

<	<
>	>
&	&
'	‘
"	“
&	Unicode char

More XML: Processing Instructions

- Syntax: <?target argument?>
- Example:

```
<product> <name> Alarm Clock </name>
          <?ringBell 20?>
          <price> 19.99 </price>
</product>
```

- What do they mean ?

More XML: Comments

- Syntax <!-- .... Comment text... -->

- Yes, they are part of the data model !!!

XML Namespaces

- name ::= [prefix:]localpart

```
<book xmlns:isbn="www.isbn-org.org/def">
    <title> … </title>
    <number> 15 </number>
    <isbn:number> …. </isbn:number>
</book>
```

Means nothing as URL; just a unique name

XML Namespaces

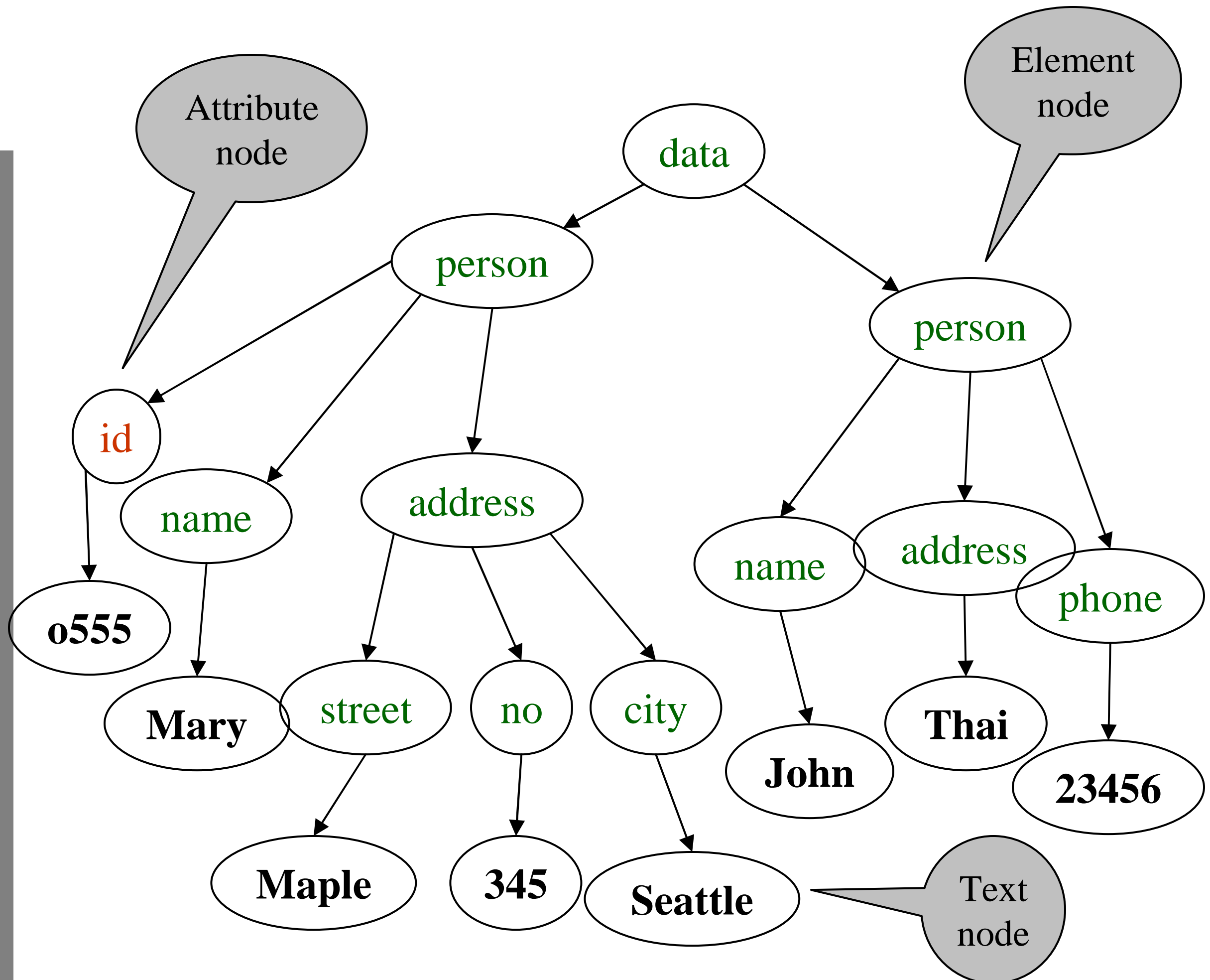
- syntactic: <number> , <isbn:number>
- semantic: provide URL for schema

```
<tag  xmlns:mystyle = “http://…”>
            …
            <mystyle:title> … </mystyle:title>
            <mystyle:number> …
   </tag>
```

Belong to this namespace

XML Semantics: a Tree !

```
<data>
  <person id="o555" >
    <name> Mary </name>
    <address>
      <street>Maple</street>
      <no> 345 </no>
      <city> Seattle </city>
    </address>
  </person>
  <person>
    <name> John </name>
    <address>Thailand
    </address>
    <phone>23456</phone>
  </person>
</data>
```



Order matters !!!

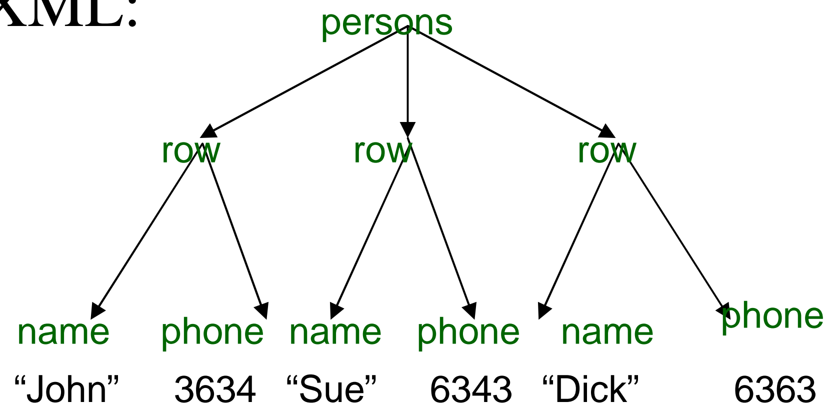
XML Data

- XML is self-describing
- Schema elements become part of the data
  - Reational schema: persons(name,phone)
  - In XML <persons>, <name>, <phone> are part of the data, and are repeated many times
- Consequence: XML is much more flexible
- XML = semistructured data

Mapping Relational Data to XML Data

The canonical mapping:

XML:



Persons

Name	Phone
John	3634
Sue	6343
Dick	6363

<persons>
<row> <name>John</name>
<phone> 3634</phone></row>
<row> <name>Sue</name>
<phone> 6343</phone>
<row> <name>Dick</name>
<phone> 6363</phone></row>
</persons>

Mapping Relational Data to XML Data

Application specific mapping

Persons

Name	Phone
John	3634
Sue	6343

Orders

PersonName	Date	Product
John	2002	Gizmo
John	2004	Gadget
Sue	2002	Gadget

XML

```
<persons>
  <person>
      <name> John </name>
      <phone> 3634 </phone>
      <order>  <date> 2002 </date>
               <product> Gizmo </product>
       </order>
       <order> <date> 2004 </date>
               <product> Gadget </product>
       </order>
  </person>
 <person>
      <name> Sue </name>
      <phone> 6343 </phone>
      <order>  <date> 2004 </date>
               <product> Gadget </product>
       </order>
  </person>
</persons>
```

XML is Semi-structured Data

- Missing attributes:

```
<person>   <name> John</name>
               <phone>1234</phone>
 </person>

<person>  <name>Joe</name>
</person>
```

no phone !

- Could represent in a table with nulls

name	phone
John	1234
Joe	-

XML is Semi-structured Data

- Repeated attributes

```
<person> <name> Mary</name>
              <phone>2345</phone>
              <phone>3456</phone>
</person>
```

Two phones !

- Impossible in tables:

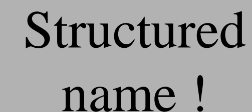
name	phone	
Mary	2345	3456

???

XML is Semi-structured Data

- Attributes with different types in different objects

```
<person> <name>  <first> John </first>
                  <last> Smith </last>
         </name>
         <phone>1234</phone>
</person>
```



- Nested collections (no 1NF)
- Heterogeneous collections:
  - <db> contains both <book>s and <publisher>s

Document Type Definitions DTD

- part of the original XML specification
- 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

DTD

Goals:

- Define what tags and attributes are allowed
- Define how they are nested
- Define how they are ordered

Superseded by XML Schema

- Very complex: DTDs still used widely

Very Simple 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)>
]>
```

Very Simple DTD

Example of valid XML document:

```
<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

<!ELEMENT tag (*CONTENT*)>

content model

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

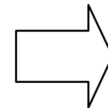
DTD: Regular Expressions

DTD

XML

sequence

```
<!ELEMENT name
              (firstName, lastName))>
```



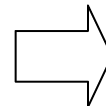
```
<name>
   <firstName> . . . . . </firstName>
   <lastName> . . . . . </lastName>
</name>
```

optional

```
<!ELEMENT name (firstName?, lastName))>
```

Kleene star

```
<!ELEMENT person   (name, phone*))>
```



```
<person>
   <name> . . . . . </name>
   <phone> . . . . . </phone>
   <phone> . . . . . </phone>
   <phone> . . . . . </phone>
   . . . . . .
</person>
```

alternation

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

XSchema
OPTIONAL MATERIAL

- Generalizes DTDs
- Uses XML syntax
- Two parts: structure and datatypes
- Very complex
  - criticized
  - alternative proposals: Relax NG

DTD v.s. XML Schemas

DTD:

```
<!ELEMENT paper (title,author*,year, (journal|conference))>
```

XML Schema:

```
<xs:element name="paper" type="paperType"/>
<xs:complexType name="paperType">
    <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="author" minOccurs="0"/>
        <xs:element name="year"/>
        <xs: choice> < xs:element name="journal"/>
                      <xs:element name="conference"/>
        </xs:choice>
    </xs:sequence>
</xs:element>
```

Example

A valid XML Document:

```
<paper>
  <title> The Essence of XML </title>
  <author> Simeon</author>
  <author> Wadler</author>
  <year>2003</year>
  <conference> POPL</conference>
</paper>
```

Elements v.s. Types

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name"
                  type="xs:string"/>
      <xs:element name="address"
                  type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
<xs:element name="person"
            type="ttt">
<xs:complexType name="ttt">
  <xs:sequence>
    <xs:element name="name"
                type="xs:string"/>
    <xs:element name="address"
                type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Both say the same thing; in DTD:

```
<!ELEMENT person (name,address)>
```

- Types:
  - Simple types (integers, strings, ...)
  - Complex types (regular expressions, like in DTDs)

- Element-type Alternation:
  - An **element** has a **type**
  - A **type** is a regular expression of **elements**

Local v.s. Global Types

- Local type:

  <xs:element name=“person”>

  [define locally the person’s type]

  </xs:element>

- Global type:

  <xs:element name=“person” type=“ttt”/>

  <xs:complexType name=“ttt”>

  [define here the type ttt]

  </xs:complexType>

Global types: can be reused in other elements

Local v.s. Global Elements

- Local element:

```
<xs:complexType name=“ttt”>
    <xs:sequence>
        <xs:element name=“address” type=“...”/>...
    </xs:sequence>
</xs:complexType>
```

- Global element:

```
<xs:element name=“address” type=“...”/>

<xs:complexType name=“ttt”>
    <xs:sequence>
        <xs:element ref=“address”/>  ...
    </xs:sequence>
</xs:complexType>
```

Global elements: like in DTDs

Regular Expressions

Recall the element-type-element alternation:

<xs:complexType name=“....”>
[regular expression on elements]
</xs:complexType>

Regular expressions:

- <xs:sequence> A B C </...> = A B C
- <xs:choice> A B C </...> = A | B | C
- <xs:group> A B C </...> = (A B C)
- <xs:... minOccurs=“0” maxOccurs=“unbounded”> ..</...> = (...)*
- <xs:... minOccurs=“0” maxOccurs=“1”> ..</...> = (...)?

Local Names

name has different meanings in **person** and in **product**

```
<xs:element name="person">
  <xs:complexType>
       . . . . .
     <xs:element name="name">
          <xs:complexType>
             <xs:sequence>
                <xs:element name="firstname" type="xs:string"/>
                <xs:element name="lastname" type="xs:string"/>
          </xs:sequence>
     </xs:element>
      . . . .
 </xs:complexType>
</xs:element>


<xs:element name="product">
  <xs:complexType>
       . . . . .
     <xs:element name="name"   type="xs:string"/>

  </xs:complexType>
</xs:element>
```

Subtle Use of Local Names

```
<xs:element name="A" type="oneB"/>


<xs:complexType name="onlyAs">
  <xs:choice>
     <xs:sequence>
         <xs:element name="A" type="onlyAs"/>
         <xs:element name="A" type="onlyAs"/>
     </xs:sequence>
     <xs:element name="A" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

```
<xs:complexType name="oneB">
  <xs:choice>
     <xs:element name="B" type="xs:string"/>
     <xs:sequence>
         <xs:element name="A" type="onlyAs"/>
         <xs:element name="A" type="oneB"/>
     </xs:sequence>
     <xs:sequence>
         <xs:element name="A" type="oneB"/>
         <xs:element name="A" type="onlyAs"/>
     </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Arbitrary deep binary tree with A elements, and a single B element

Note: this example is not legal in XML Schema (why ?)
Hence they cannot express all regular tree languages

Attributes in XML Schema

```
<xs:element name=“paper” type=“papertype”>
  <xs:complexType name=“papertype”>
      <xs:sequence>
          <xs:element name=“title” type=“xs:string”/>
          . . . . . .
      </xs:sequence>
      <xs:attribute name=“language" type="xs:NMTOKEN" fixed=“English"/>
  </xs:complexType>
</xs:element>
```

Attributes are associated to the *type*, not to the element

Only to *complex types*; more trouble if we want to add attributes to *simple types*.

“Mixed” Content, “Any” Type

```
<xs:complexType mixed="true">
  . . . .
```

- Better than in DTDs: can still enforce the type, but now may have text between any elements

```
<xs:element name="anything" type="xs:anyType"/>
  . . . .
```

- Means anything is permitted there

“All” Group

```
<xs:complexType name="PurchaseOrderType">
  <xs:all>  <xs:element name="shipTo" type="USAddress"/>
            <xs:element name="billTo" type="USAddress"/>
            <xs:element ref="comment" minOccurs="0"/>
            <xs:element name="items"  type="Items"/>
  </xs:all>
  <xs:attribute name="orderDate" type="xs:date"/>
</xs:complexType>
```

- A restricted form of & in SGML
- Restrictions:
  - Only at top level
  - Has only elements
  - Each element occurs at most once
- E.g. “comment” occurs 0 or 1 times

Derived Types by Extensions

```
<complexType name="Address">
    <sequence>  <element name="street" type="string"/>
                <element name="city"   type="string"/>
    </sequence>
</complexType>

<complexType name="USAddress">
  <complexContent>
      <extension base="ipo:Address">
          <sequence>  <element name="state" type="ipo:USState"/>
                      <element name="zip"   type="positiveInteger"/>
          </sequence>
      </extension>
  </complexContent>
</complexType>
```

Corresponds to inheritance

Derived Types by Restrictions

```
<complexContent>
    <restriction base="ipo:Items“>
     … [rewrite the entire content, with restrictions]...
    </restriction>
 </complexContent>
```

- (*): may restrict cardinalities, e.g. (0,infty) to (1,1); may restrict choices; other restrictions…

Corresponds to set inclusion

Simple Types

- String
- Token
- Byte
- unsignedByte
- Integer
- positiveInteger
- Int (larger than integer)
- unsignedInt
- Long
- Short
- ...

- Time
- dateTime
- Duration
- Date
- ID
- IDREF
- IDREFS

Facets of Simple Types

- Facets = additional properties restricting a simple type
- 15 facets defined by XML Schema

Examples

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

- maxInclusive
- maxExclusive
- minInclusive
- minExclusive
- totalDigits
- fractionDigits

Facets of Simple Types

- Can further restrict a simple type by changing some facets
- Restriction = subset

Not so Simple Types

- List types:

```
<xs:simpleType name="listOfMyIntType">
  <xs:list itemType="myInteger"/>
</xs:simpleType>
```

```
<listOfMyInt>20003 15037 95977 95945</listOfMyInt>
```

- Union types
- Restriction types

END OF OPTIONAL MATERIAL

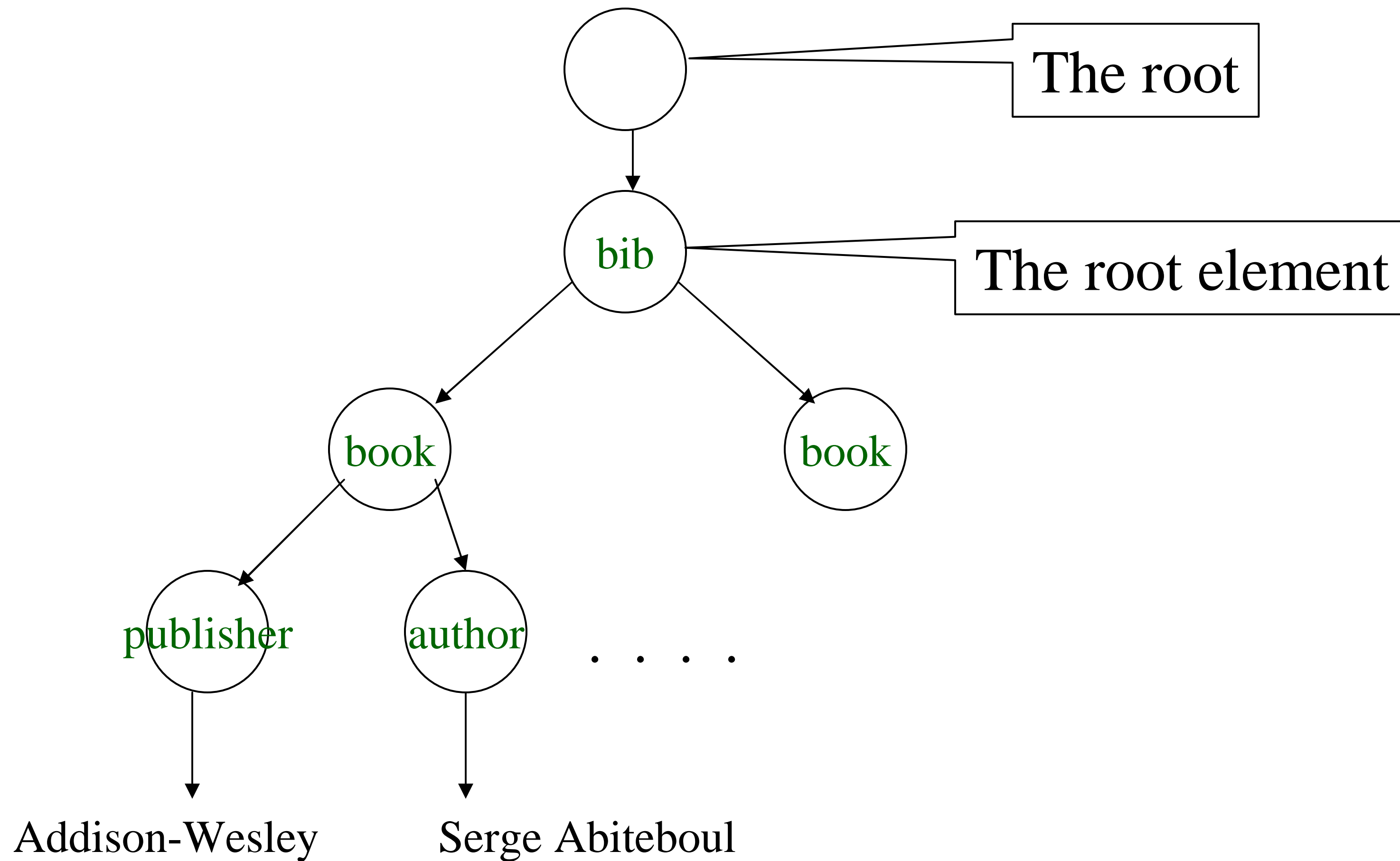
Querying XML Data

- XPath = simple navigation through the tree
- XQuery = the SQL of XML
- XSLT = recursive traversal
  - will not discuss in class

Sample Data for Queries

```
<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: Simple Expressions

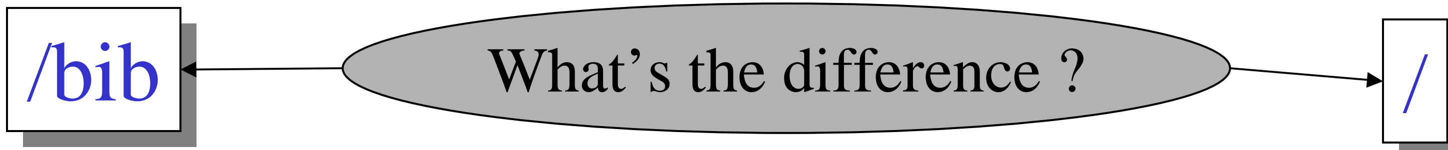
/bib/book/year

Result: <year> 1995 </year>

<year> 1998 </year>

/bib/paper/year

Result: empty (there were no papers)



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>

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

```
/bib//first-name
```

Xpath: Attribute Nodes

```
/bib/book/@price
```

Result: “55”

@price means that price is 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

/bib/book/author/text()

Result: Serge Abiteboul
Victor Vianu
Jeffrey D. Ullman

Rick Hull doesn't appear because he has firstname, lastname

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[firstname]

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

Xpath: More Predicates

/bib/book/author[firstname][address[.//zip][city]]/lastname

Result: <lastname> … </lastname>

<lastname> … </lastname>

How do we read this ?

First remove all qualifiers (predicates):

/bib/book/author /lastname

Then add them one by one:

/bib/book/author[firstname][address]/lastname

etc

Xpath: More Predicates

```
/bib/book[@price < 60]
```

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

```
/bib/book[author/text()]
```

Xpath: More Axes

. means *current node*

/bib/book[.//review]

/bib/book[./review] Same as /bib/book[review]

/bib/author/. /firstname Same as /bib/author/firstname

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]
```

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

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/lastname matches…

XQuery

- Based on Quilt, which is based on XML-QL
- Uses XPath to express more complex queries

NOTE: will skip many slides in class; please study at home

FLWR ("Flower") Expressions

FOR ...

LET...

WHERE...

RETURN...

FOR-WHERE-RETURN

Find all book titles published after 1995:

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

Result:

```
<title> abc </title>
<title> def </title>
<title> ghi </title>
```

FOR-WHERE-RETURN

Equivalently (perhaps more geekish)

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

And even shorter:

```
document("bib.xml")/bib/book[year/text() > 1995] /title
```

FOR-WHERE-RETURN

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

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

Result:

<answer> <title> abc </title> <year> 1995 </ year > </answer>
<answer> <title> def </title> < year > 2002 </ year > </answer>
<answer> <title> ghk </title> < year > 1980 </ year > </answer>

FOR-WHERE-RETURN

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

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

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

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

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

Nesting

For each author of a book by Morgan Kaufmann, list all books she published:

```
FOR $b IN document("bib.xml")/bib,
      $a IN $b/book[publisher /text()="Morgan Kaufmann"]/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>Jones</author>
        <title> abc </title>
        <title> def </title>
  </result>
  <result>
        <author> Smith </author>
        <title> ghi </title>
  </result>
```

Aggregates

Find all books with more than 3 authors:

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

Aggregates

Print all authors who published more than 3 books – be aware of duplicates !

```
FOR $b IN document("bib.xml")/bib,
      $a IN distinct-values($b/book/author/text())
WHERE count($b/book[author/text()=$a])>3
RETURN <author> { $a } </author>
```

Aggregates

Find books whose price is larger than average:

```
FOR $b in document("bib.xml")/bib
LET $a:=avg($b/book/price/text())
FOR $x in $b/book
WHERE $x/price/text() > $a
RETURN $x
```

Flattening

- "Flatten" the authors, i.e. return a list of (author, title) pairs

```
FOR $b IN document("bib.xml")/bib/book,
      $x IN $b/title/text(),
      $y IN $b/author/text()
RETURN <answer>
            <title> { $x } </title>
            <author> { $y } </author>
       </answer>
```

Result:

```
<answer>
   <title> abc </title>
   <author> efg </author>
</answer>
<answer>
   <title> abc </title>
   <author> hkj </author>
</answer>
```

Re-grouping

- For each author, return all titles of her/his books

```
FOR $b IN document("bib.xml")/bib,
      $x IN $b/book/author/text()
RETURN
  <answer>
    <author> { $x } </author>
    { FOR $y IN $b/book[author/text()=$x]/title
      RETURN $y }
  </answer>
```

Result:

```
<answer>
   <author> efg </author>
    <title> abc </title>
    <title> klm </title>
    . . . .
</answer>
```

What about duplicate authors ?

Re-grouping

- Same, but eliminate duplicate authors:

```
FOR $b IN document("bib.xml")/bib
LET $a := distinct-values($b/book/author/text())
FOR $x IN $a
RETURN
  <answer>
    <author> $x </author>
    { FOR $y IN $b/book[author/text()=$x]/title
      RETURN $y }
  </answer>
```

Re-grouping

- Same thing:

```
FOR $b IN document("bib.xml")/bib,
      $x IN distinct-values($b/book/author/text())
RETURN
  <answer>
    <author> $x </author>
    { FOR $y IN $b/book[author/text()=$x]/title
      RETURN $y }
  </answer>
```

Another Example

Find book titles by the coauthors of “Database Theory”:

```
FOR $b IN document("bib.xml")/bib,
        $x IN $b/book[title/text() = “Database Theory”],
        $y IN $b/book[author/text() = $x/author/text()]
RETURN <answer> { $y/title/text() } </answer>
```

Result:

<answer> abc </ answer >

< answer > def </ answer >

< answer > abc </ answer >

< answer > ghk </ answer >

Question:
Why do we get duplicates ?

Distinct-values

Same as before, but eliminate duplicates:

```
FOR $b IN document("bib.xml")/bib,
        $x IN $b/book[title/text() = “Database Theory”]/author/text(),
        $y IN distinct-values($b/book[author/text() = $x] /title/text())

RETURN <answer> { $y } </answer>
```

distinct-values = a function that eliminates duplicates

Result:

<answer> abc </ answer >
< answer > def </ answer >
< answer > ghk </ answer >

Need to apply to a collection of *text* values, not of *elements – note how query has changed*

SQL and XQuery Side-by-side

Product(pid, name, maker, price)

Find all product names, prices, sort by price

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

SQL

```
FOR $x in document(“db.xml”)/db/Product/row
ORDER BY $x/price/text()
RETURN <answer>
          { $x/name,  $x/price }
       </answer>
```

XQuery

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 document("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

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

SQL

```
FOR $r in document("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

Cool XQuery

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

```
<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 document("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 document("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>
```

An element

A collection

Sorting in XQuery

```
<publisher_list>
 { FOR $b IN document("bib.xml")//book[year = “97”]
   ORDER BY $b/price/text()
   RETURN <book>
                  {  $b/title ,
                     $b/price
                  }
               </book>
 }
</publisher_list>
```

If-Then-Else

```
FOR $h IN //holding
RETURN <holding>
         { $h/title,
             IF $h/@type = "Journal"
                  THEN $h/editor
             ELSE $h/author
         }
       </holding>
```

Existential Quantifiers

```
FOR $b IN //book
WHERE SOME $p IN $b//para SATISFIES
                contains($p, "sailing")
                AND contains($p, "windsurfing")
RETURN { $b/title }
```

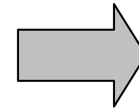
Universal Quantifiers

```
FOR $b IN //book
WHERE EVERY $p IN $b//para SATISFIES
                contains($p, "sailing")
RETURN { $b/title }
```

Duplicate Elimination

- distinct-values(list-of-text-values)
- How do we eliminate duplicate “tuples” ?

```
<row> <a>3</a> <b>100</b> </row>
<row> <a>8</a> <b>500</b> </row>
<row> <a>3</a> <b>100</b> </row>
<row> <a>3</a> <b>200</b> </row>
<row> <a>8</a> <b>500</b> </row>
```



```
<row> <a>3</a> <b>100</b> </row>
<row> <a>8</a> <b>500</b> </row>
<row> <a>3</a> <b>200</b> </row>
```

FOR v.s. LET

FOR

- Binds *node variables* → iteration

LET

- Binds *collection variables* → one value

FOR v.s. LET

```
FOR $x IN /bib/book
RETURN <result> { $x } </result>
```

Returns:

<result> <book>...</book></result>
<result> <book>...</book></result>
<result> <book>...</book></result>
...

```
LET $x := /bib/book
RETURN <result> { $x } </result>
```

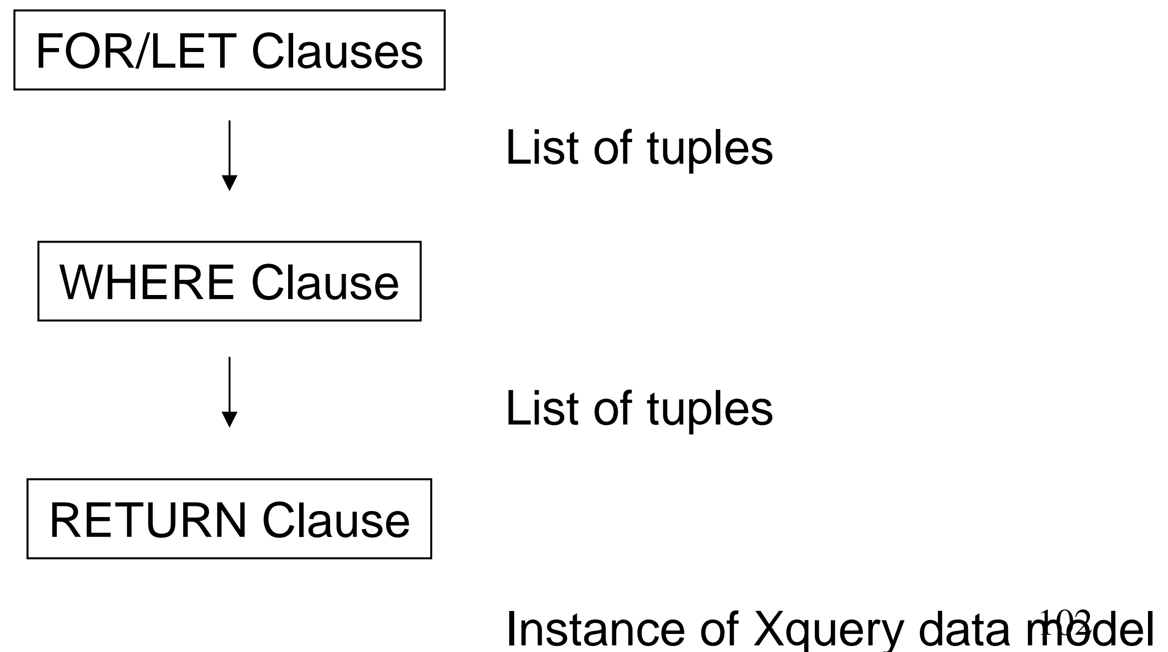
Returns:

<result> <book>...</book>
<book>...</book>
<book>...</book>
...
</result>

XQuery

Summary:

- FOR-LET-WHERE-RETURN = FLWR



Collections in XQuery

- Ordered and unordered collections
  - /bib/book/author/text() = an *ordered* collection: result is in *document order*
  - distinct-values(/bib/book/author/text()) = an unordered collection: the output order is implementation dependent
- <u>LET</u> $a := /bib/book → $a is a collection
- $b/author → a collection (several authors...)

<u>RETURN</u> <result> { $b/author } </result>

Returns:

```
<result> <author>...</author>
         <author>...</author>
         <author>...</author>
         ...
</result>
```

Collections in XQuery

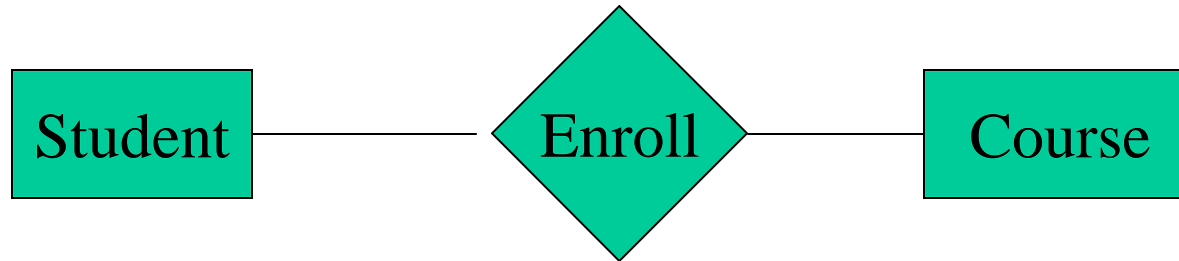
What about collections in expressions ?

- $b/price → list of n prices
- $b/price * 0.7 → list of n numbers
- $b/price * $b/quantity → list of n x m numbers ??
- $b/price * ($b/quant1 + $b/quant2) ≠
  $b/price * $b/quant1 + $b/price * $b/quant2 !!

Two Applications

- XML publishing:
  - The data is in a relational database
  - The users want to see an XML view
- XML storage
  - We have XML data
  - But want to process it in a relational engine

XML Publishing



- Relational schema:

  Student(sid, name, address)

  Course(cid, title, room)

  Enroll(sid, cid, grade)

XML Publishing

```
<xmlview>
   <course>  <title> Operating Systems </title>
             <room> MGH084 </room>
             <student> <name> John </name>
                       <address> Seattle </address >
                       <grade> 3.8 </grade>
             </student>
             <student> …</student>
       …
   </course>
   <course>  <title> Database </title>
             <room> EE045 </room>
             <student> <name> Mary </name>
                       <address> Shoreline </address >
                       <grade> 3.9 </grade>
             </student>
             <student> …</student>
       …
   </course>
   …
</xmlview>
```

Group by courses: redundant representation of students

Other representations possible too

XML Publishing

First thing to do: design the DTD:

```
<!ELEMENT xmlview (course*)>
<!ELEMENT course (title, room, student*)>
<!ELEMENT student (name,address,grade)>
<!ELEMENT name    (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT grade   (#PCDATA)>
<!ELEMENT title   (#PCDATA)>
```

Now we write an XQuery to export relational data → XML
Note: result is is the right DTD

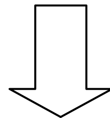
```
<xmlview>
{ FOR $x IN /db/Course/row
  RETURN
    <course>
       <title>   { $x/title/text() } </title>
       <room> { $x/room/text() } </room>
       { FOR  $y IN /db/Enroll/row[cid/text() = $x/cid/text()]
              $z IN /db/Student/row[sid/text() = $y/sid/text()]
         RETURN <student> <name>    { $z/name/text() } </name>
                          <address> { $z/address/text() } </address>
                          <grade>   { $y/grade/text() } </grade>
                </student>
       }
    </course>
}
</xmlview>
```

XML Publishing

Query: find Mary's grade in Operating Systems

XQuery

```
FOR $x IN /xmlview/course[title/text()="Operating Systems"],
      $y IN $x/student/[name/text()="Mary"]
RETURN <answer> { $y/grade/text() } </answer>
```



Can be done automatically

SQL

```
SELECT Enroll.grade
FROM Student, Enroll, Course
WHERE Student.name="Mary" and Course.title="OS"
    and Student.sid = Enroll.sid and Enroll.cid = Course.cid
```

XML Publishing

How do we choose the output structure ?

- Determined by agreement with partners/users
- Or dictated by committees
  - XML dialects (called *applications*) = DTDs

- XML Data is often nested, irregular, etc
- No normal forms for XML

XML Storage

- Most often the XML data is small
  - E.g. a SOAP message
  - Parsed directly into the application (DOM API)
- Sometimes XML data is large
  - need to store/process it in a database
- The XML storage problem:
  - How do we choose the schema of the database ?

XML Storage

Three solutions:

- Schema derived from DTD
- Storing XML as a graph: “Edge relation”
- Store it as a BLOB
  - Simple, boring, inefficient
  - Won’t discuss in class

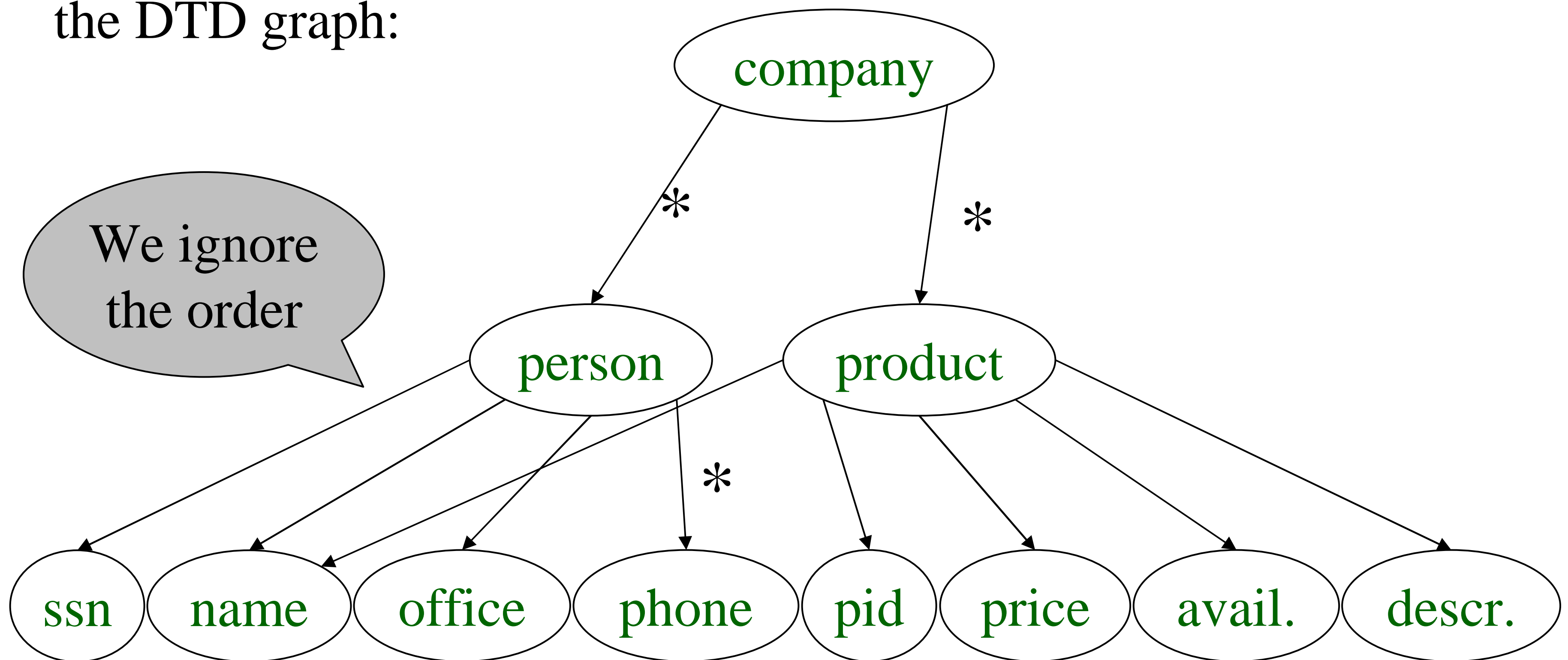
Designing a Schema from DTD

Design a relational schema for:

```
<!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, ((price,availability)|description))>
  <!ELEMENT pid     (#PCDATA)>
  <!ELEMENT description    (#PCDATA)>
]>
```

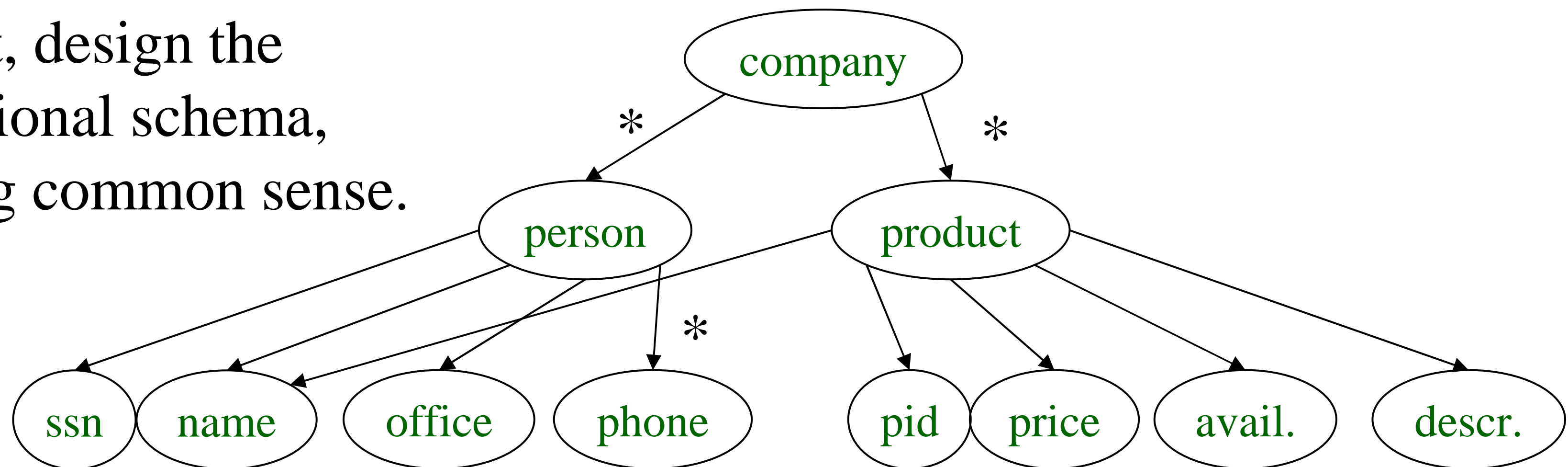
Designing a Schema from DTD

First, construct the DTD graph:



Designing a Schema from DTD

Next, design the relational schema, using common sense.



Person(<u>ssn</u>, name, office)
Phone(<u>ssn, phone</u>)
Product(<u>pid</u>, name, price, avail., descr.)

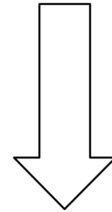
This is a simple exercise of transforming to 1NF

Which attributes may be NULL ? (Look at the DTD)

Designing a Schema from DTD

What happens to queries:

```
FOR $x IN /company/product[description]
RETURN  <answer> { $x/name, $x/description } </answer>
```



```
SELECT Product.name, Product.description
FROM    Product
WHERE Product.description IS NOT NULL
```

Storing XML as a Graph

Sometimes we don't have a DTD:

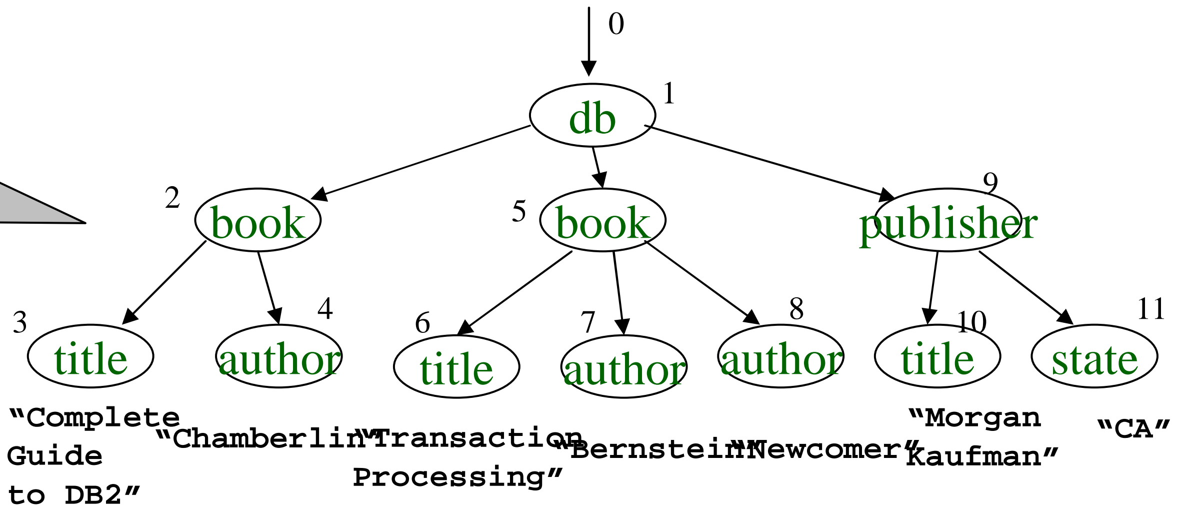
- How can we store the XML data ?

Every XML instance is a *tree*

- Store the edges in an Edge table
- Store the #PCDATA in a Value table

Storing XML as a Graph

Can be ANY XML data (don't know DTD)



Edge

Source	Tag	Dest
0	db	1
1	book	2
2	title	3
2	author	4
1	book	5
5	title	6
5	author	7
. . .	. . .	. . .

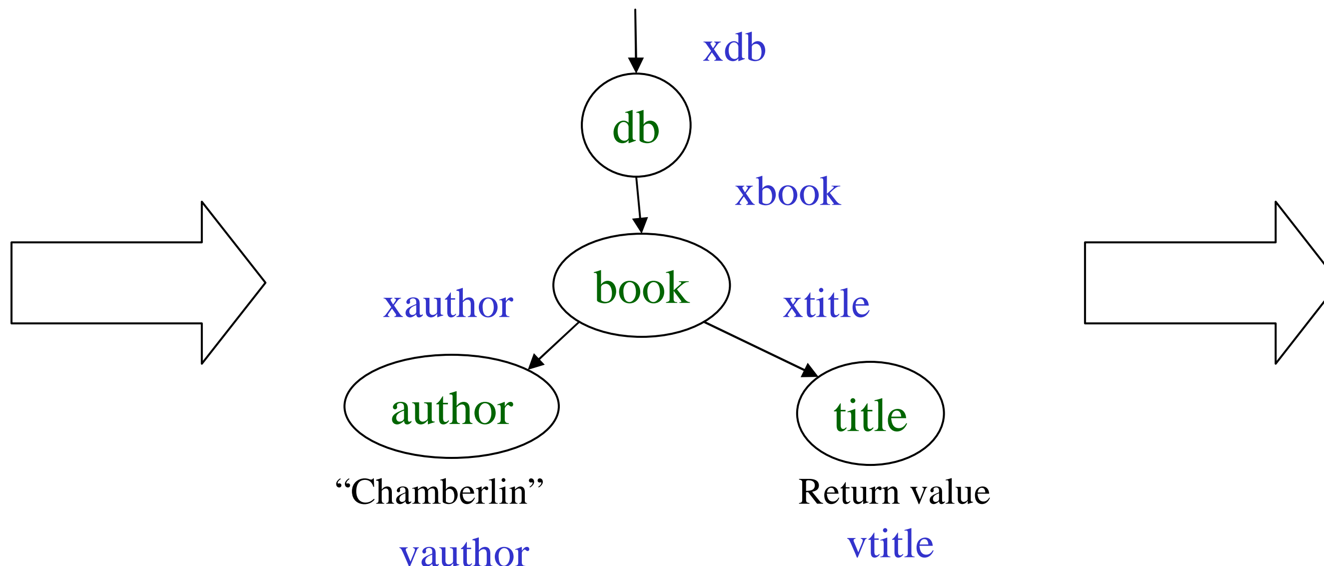
Value

Source	Val
3	Complete guide . . .
4	Chamberlin
6	. . .
. . .	. . .

Storing XML as a Graph

What happens to queries:

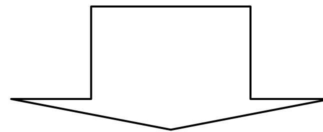
```
FOR $x IN /db/book[author/text()="Chamberlin"]
RETURN $x/title
```



Storing XML as a Graph

What happens to queries:

A 6-way join !!!



```
SELECT vtitle.value
FROM    Edge xdb, Edge xbook, Edge xauthor, Edge xtitle,
        Value vauthor, Value vtitle
WHERE xdb.source = 0                          and xdb.tag = 'db'
      and xdb.dest = xbook.source             and xbook.tag = 'book'
      and xbook.dest = xauthor.source         and xauthor.tag = 'author'
      and xbook.dest = xtitle.source          and xtitle.tag = 'title'
      and xauthor.dest = vauthor.source       and vauthor.value = 'Chamberlin"
      and xtitle.dest = vtitle.source
```

Storing XML as a Graph

Edge relation summary:

- Same relational schema for every XML document:

  Edge(Source, Tag, Dest)

  Value(Source, Val)

- Generic: works for *every* XML instance
- But inefficient:
  - Repeat tags multiple times
  - Need many joins to reconstruct data

XML in SQL Server 2005

- Create tables with attributes of type XML
- Use Xquery in SQL queries
- Rest of the slides are from:

Shankar Pal et al., *Indexing XML data stored in a relational database*, VLDB'2004

```
CREATE TABLE DOCS (
      ID int primary key,
      XDOC xml)
```

```
SELECT ID, XDOC.query('
   for $s in  /BOOK[@ISBN= "1-55860-438-3"]//SECTION
   return <topic>{data($s/TITLE)} </topic>')
FROM DOCS
```

XML Methods in SQL

- Query() = returns XML data type
- Value() = extracts scalar values
- Exist() = checks conditions on XML nodes
- Nodes() = returns a rowset of XML nodes that the Xquery expression evaluates to

Examples

- From here: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql90/html/sql2k5xml.asp

XML Type

```
CREATE TABLE docs (
    pk INT PRIMARY KEY,
    xCol XML not null
)
```

Inserting an XML Value

```
INSERT INTO docs VALUES (2,
'<doc id="123">
    <sections>
   <section num="1"><title>XML Schema</title></section>
   <section num="3"><title>Benefits</title></section>
   <section num="4"><title>Features</title></section>
    </sections>
</doc>')
```

Query( )

```
SELECT pk, xCol.query('/doc[@id = 123]//section')
FROM   docs
```

Exists( )

```
SELECT xCol.query('/doc[@id = 123]//section')
FROM   docs
WHERE  xCol.exist ('/doc[@id = 123]') = 1
```

Value( )

```
SELECT xCol.value(
  'data((/doc//section[@num = 3]/title)[1])', 'nvarchar(max)')
FROM docs
```

Nodes( )

```
SELECT nref.value('first-name[1]', 'nvarchar(50)')
                    AS  FirstName,
            nref.value('last-name[1]', 'nvarchar(50)')
                     AS LastName
FROM   @xVar.nodes('//author') AS R(nref)
WHERE  nref.exist('.[first-name != "David"]') = 1
```

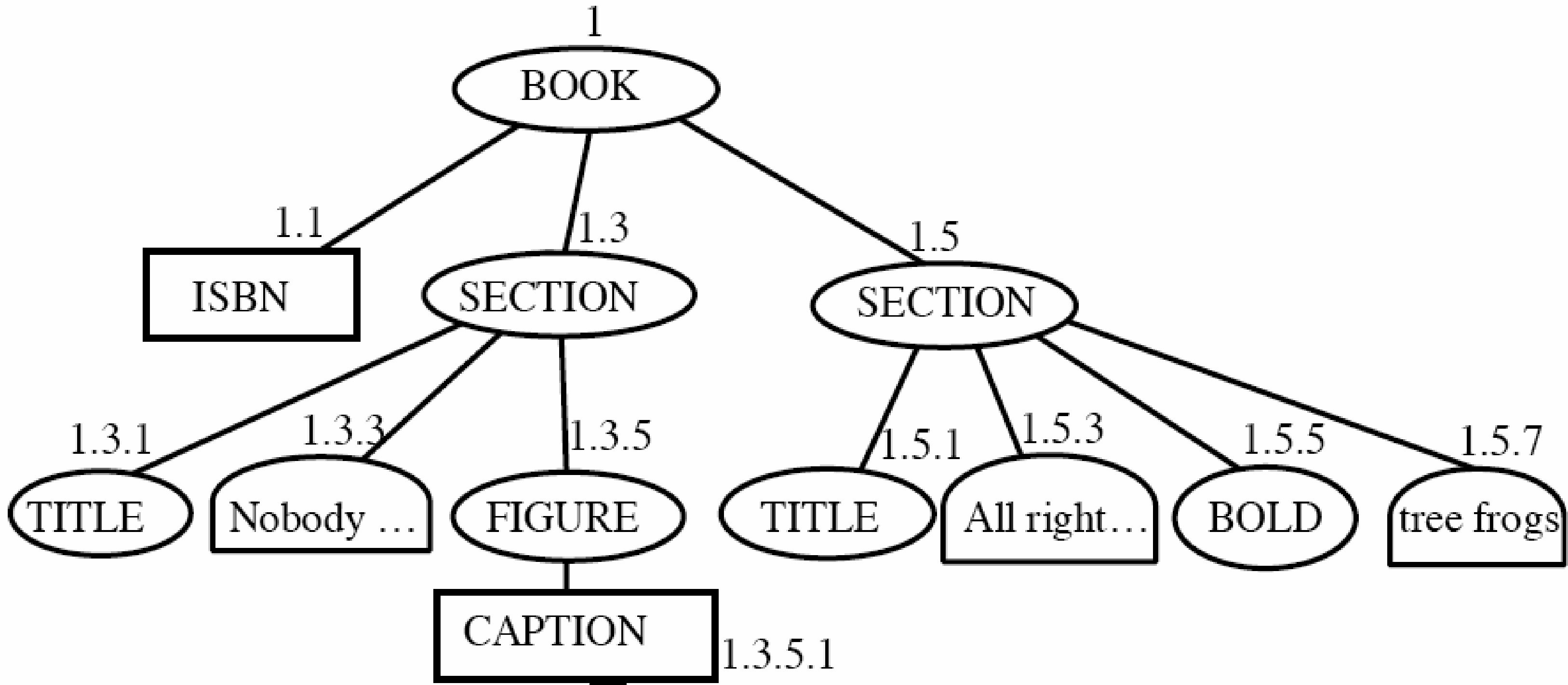
Nodes( )

```
SELECT nref.value('@genre', 'varchar(max)') LastName
FROM docs CROSS APPLY xCol.nodes('//book') AS R(nref)
```

Internal Storage

- XML is "shredded" as a table
- A few important ideas:
  - Dewey decimal numbering of nodes; store in clustered B-tree indes
  - Use only odd numbers to allow insertions
  - Reverse PATH-ID encoding, for efficient processing of postfix expressions like //a/b/c
  - Add more indexes, e.g. on data values

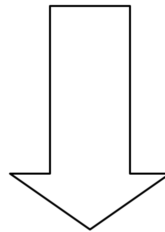
```
<BOOK ISBN="1-55860-438-3">
  <SECTION>
    <TITLE>Bad Bugs</TITLE>
    Nobody loves bad bugs.
    <FIGURE CAPTION="Sample bug"/>
  </SECTION>
  <SECTION>
   <TITLE>Tree Frogs</TITLE>
   All right-thinking people
   <BOLD> love </BOLD>
   tree frogs.
  </SECTION>
</BOOK>
```


1
BOOK
1.1
ISBN
1.3
SECTION
1.5
SECTION
1.3.1
TITLE
1.3.3
Nobody …
1.3.5
FIGURE
CAPTION
1.3.5.1
1.5.1
TITLE
1.5.3
All right…
1.5.5
BOLD
1.5.7
tree frogs

ORDPATH	TAG	NODE_ TYPE	VALUE	PATH_ ID
1	1 (BOOK)	1 (Element)	Null	#1
1.1	2 (ISBN)	2 (Attribute)	'1-55860-438-3'	#2#1
1.3	3 (SECTION)	1 (Element)	Null	#3#1
1.3.1	4 (TITLE)	1 (Element)	'Bad Bugs'	#4#3#1
1.3.3	10 (TEXT)	4 (Value)	'Nobody loves Bad bugs.'	#10#3#1
1.3.5	5 (FIGURE)	1 (Element)	Null	#5#3#1
1.3.5.1	6 (CAPTION)	2 (Attribute)	'Sample bug'	#6#3#1
1.5	3 (SECTION)	1 (Element)	Null	#3#1
1.5.1	4 (TITLE)	1 (Element)	'Tree frogs'	#4#3#1
1.5.3	10 (TEXT)	4 (Value)	'All right-thinking people'	#10#3#1
1.5.5	7 (BOLD)	1 (Element)	'love '	#7#3#1
1.5.7	10 (TEXT)	4 (Value)	'tree frogs'	#10#3#1

Infoset Table

```
/BOOK[@ISBN = "1-55860-438-3"]/SECTION
```



```
SELECT SerializeXML (N2.ID, N2.ORDPATH)
FROM infosettab N1  JOIN infosettab N2 ON (N1.ID = N2.ID)
WHERE N1.PATH_ID = PATH_ID(/BOOK/@ISBN)
    AND N1.VALUE = '1-55860-438-3'
    AND N2.PATH_ID = PATH_ID(BOOK/SECTION)
    AND Parent (N1.ORDPATH) = Parent (N2.ORDPATH)
```