XML and Web Data

Chapter 15

1

What's in This Module?

- Semistructured data
- XML & DTD – introduction
- XML Schema – user-defined data types, integrity constraints
- XPath & XPointer – core query language for XML
- XSLT – document transformation language
- XQuery – full-featured query language for XML
- SQL/XML – XML extensions of SQL

2

Why XML?

- XML is a standard for data exchange that is taking over the World
- All major database products have been retrofitted with facilities to store and construct XML documents
- There are already database products that are <u>specifically designed</u> to work with XML documents rather than relational or object-oriented data
- XML is closely related to object-oriented and so-called *semistructured* data

3

Semistructured Data

- An HTML document to be displayed on the Web:

```
<dt>Name: John Doe
    <dd>Id: s111111111
    <dd>Address:  <ul>
                      <li>Number:  123</li>
                      <li>Street:     Main</li>
                  </ul>
</dt>
<dt>Name: Joe Public
     <dd>Id:  s222222222
     … … … …
</dt>
```

HTML does not distinguish between attributes and values

4

Semistructured Data (cont'd.)

- To make the previous student list suitable for machine consumption on the Web, it should have these characteristics:
  - Be *object-like*
  - Be *schemaless* (not guaranteed to conform exactly to any schema, but different objects have some commonality among themselves)
  - Be *self-describing* (some schema-like information, like attribute names, is part of data itself)
- Data with these characteristics are referred to as *semistructured*.

5

What is Self-describing Data?

- Non-self-describing (relational, object-oriented):

Data part:

(#123, ["Students", {["John", s111111111, [123,"Main St"]],
["Joe", s222222222, [321, "Pine St"]] }
])

Schema part:

PersonList[*ListName*: String,
Contents: [*Name*: String,
Id: String,
Address: [*Number*: Integer, *Street*: String]]
]

6

What is Self-Describing Data? (contd.)

- *Self-describing*:
  - Attribute names embedded in the data itself, *but are distinguished* from values
  - Doesn't need schema to figure out what is what (but schema might be useful nonetheless)

(#12345,
  [*ListName*: "Students",
   Contents: { [*Name*: "John Doe",
                  Id: "s111111111",
                  Address: [*Number*: 123, *Street*: "Main St."]] ,
                [*Name*: "Joe Public",
                 Id: "s222222222",
                 Address: [*Number*: 321, *Street*: "Pine St."]] }
])

7

XML – The De Facto Standard for Semistructured Data

- XML: eXtensible Markup Language
  - Suitable for semistructured data and has become a standard:
    - Easy to describe object-like data
    - Self-describing
    - Doesn't require a schema (but can be provided optionally)
- We will study:
  - DTDs – an older way to specify schema
  - XML Schema – a newer, more powerful (and <u>much</u> more complex!) way of specifying schema
  - Query and transformation languages:
    - XPath
    - XSLT
    - XQuery
    - SQL/XML

8

Overview of XML

- Like HTML, but any number of different tags can be used (up to the document author) – extensible
- Unlike HTML, no semantics behind the tags
  - For instance, HTML's <table>…</table> means: render contents as a table; in XML: doesn't mean anything special
  - Some semantics can be specified using XML Schema (types); some using stylesheets (browser rendering)
- Unlike HTML, is intolerant to bugs
  - Browsers will render buggy HTML pages
  - *XML processors* are not supposed to process buggy XML documents

Example

attributes

```
<?xml  version="1.0" ?>

<PersonList Type="Student"   Date="2002-02-02" >
     <Title  Value="Student List" />
     <Person>
     … …..
     </Person>
     <Person>
     … …..
     </Person>
</PersonList>
```

elements

Empty element

Root element

Element (or tag) names

- Elements are nested
- Root element contains all others

More Terminology

Opening tag

<Person Name = "John" Id = "s111111111">

Content of Person

John is a nice fellow

"standalone" text, not very useful as data, non-uniform

<Address>
<Number>21</Number>
<Street>Main St.</Street>
</Address>
… … …

Nested element, child of Person

Child of Address, *Descendant of* Person

Parent of Address, *Ancestor of* number

</Person>

Closing tag:
What is open must be closed

11

Conversion from XML to Objects

- Straightforward:

<Person Name="Joe">
<Age>44</Age>
<Address><Number>22</Number><Street>Main</Street></Address>
</Person>

Becomes:

(#345, [*Name*: "Joe",
Age: 44,
Address: [*Number*: 22, *Street*: "Main"]
])

12

Conversion from Objects to XML

- Also straightforward
- Non-unique:
  - Always a question if a particular piece (such as Name) should be an element in its own right or an attribute of an element
  - *Example*: A reverse translation could give

<Person>
<Name>Joe</Name>
<Age>44</Age>
<Address>
<Number>22</Number>
<Street>Main</Street>
</Address>
</Person>

<Person Name="Joe">
… ……

This or this

Differences between XML Documents and Objects

- XML's origin is document processing, not databases
  - Allows things like standalone text (useless for databases)

    <foo> Hello <moo>123</moo> Bye </foo>
  - XML data is ordered, while database data is not:

    <something><foo>1</foo><bar>2</bar></something>

    is different from

    <something><bar>2</bar><foo>1</foo></something>

    but these two complex values are same:

    [*something*: [*bar*:1, *foo*:2]]

    [*something*: [*foo*:2, *bar*:1]]

Differences between XML Documents and Objects (cont'd)

- Attributes aren't needed – just bloat the number of ways to represent the same thing:

More concise

<foo bar="12">ABC</foo>

vs.

<foobar><foo>ABC</foo><bar>12</bar></foobar>

More uniform, database-like

15

Well-formed XML Documents

- Must have a *root element*
- Every *opening tag* must have matching *closing tag*
- Elements must be *properly nested*
  - <foo><bar></foo></bar> is a no-no
- An *attribute* name can occur *at most once* in an opening tag. If it occurs,
  - It *must have an explicitly specified value* (Boolean attrs, like in HTML, are not allowed)
  - The value *must be quoted* (with " or ')
- *XML processors are not supposed to try and fix ill-formed documents (unlike HTML browsers)*

16

Identifying and Referencing with Attributes

- An attribute can be declared (in a DTD – see later) to have type:
  - *ID* – unique identifier of an element
    - If attr1 & attr2 are both of type ID, then it is illegal to have <something attr1=“*abc*”> …<somethingelse attr2=“ *abc*”> within the same document
  - *IDREF* – references a unique element with matching ID attribute (in particular, an XML document with IDREFs is not a tree)
    - If attr1 has type ID and attr2 has type IDREF then we <u>can</u> have: <something attr1=“*abc*”> …<somethingelse attr2=“ *abc*”>
  - *IDREFS* – a list of references, if attr1 is ID and attr2 is IDREFS, then we can have
    - <something attr1=“*abc*”>…<somethingelse attr1=“ *cde*”>…
      <someotherthing attr2=“*abc cde*”>

17

Example: Report Document with Cross-References

```
<?xml version=“1.0” ?>
<Report Date=“2002-12-12”>                          ID
  <Students>
    <Student StudId=“s111111111”>
      <Name><First>John</First><Last>Doe</Last></Name>  <Status>U2</Status>
      <CrsTaken CrsCode=“CS308” Semester=“F1997” />
      <CrsTaken CrsCode=“MAT123” Semester=“F1997” />
    </Student>
    <Student StudId=“s666666666”>
      <Name><First>Joe</First><Last>Public</Last></Name>  <Status>U3</Status>
      <CrsTaken CrsCode=“CS308” Semester=“F1994” />
      <CrsTaken CrsCode=“MAT123” Semester=“F1997” />
    </Student>
    <Student StudId=“s987654321”>
      <Name><First>Bart</First><Last>Simpson</Last></Name>  <Status>U4</Status>
      <CrsTaken CrsCode=“CS308” Semester=“F1994” />
    </Student>
  </Students>                                       IDREF
 …..  continued … …
```

18

Report Document (cont'd.)

IDREFS

```
<Classes>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
    <ClassRoster Members="s666666666 987654321"/>
  </Class>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="s111111111"/>
  </Class>
  <Class>
    <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="s111111111 s666666666"/>
  </Class>
</Classes>
```

…… *continued* … …

19

Report Document (cont'd.)

ID

```
<Courses>
  <Course CrsCode = "CS308">
    <CrsName>Market Analysis</CrsName>
  </Course>
  <Course CrsCode = "MAT123">
    <CrsName>Market Analysis</CrsName>
  </Course>
</Courses>
</Report>
```

20

XML Namespaces

- A mechanism to prevent name clashes between components of same or different documents
- Namespace declaration
  - *Namespace* – a symbol, typically a URL (*doesn't need to point to a real page*)
  - *Prefix* – an abbreviation of the namespace, a convenience; works as an alias
  - Actual name (element or attribute) – *prefix*:*name*
  - Declarations/prefixes have *scope* similarly to begin/end
- Example:

```
<item xmlns = "http://www.acmeinc.com/jp#supplies"
      xmlns:toy = "http://www.acmeinc.com/jp#toys">
   <name>backpack</name>
   <feature>
      <toy:item><toy:name>cyberpet</toy:name></toy:item>
   </feature>
</item>
```

Default namespace

toy namespace

reserved keyword

21

Namespaces (cont'd.)

- Scopes of declarations are color-coded:

```
<item xmlns="http://www.foo.org/abc"
      xmlns:cde="http://www.bar.com/cde">
   <name>…</ name>
   <feature>
      <cde:item><cde:name>…</ cde:name><cde:item>
   </feature>
   <item  xmlns="http://www.foobar.org/"
          xmlns:cde="http://www.foobar.org/cde" >
      <name>…</ name>
      <cde:name>…</ cde:name>
   </item>
</item>
```

New default; overshadows old default

Redeclaration of cde*; overshadows old declaration*

22

Namespaces (cont'd.)

- xmlns="http://foo.com/bar" *doesn't* mean there is a document at this URL: using URLs is just a convenient convention; and a namespace is just an identifier
- Namespaces aren't part of XML 1.0, but all XML processors understand this feature now
- A number of prefixes have become "standard" and some XML processors might understand them without any declaration. E.g.,
  - **xs** for http://www.w3.org/2001/XMLSchema
  - **xsl** for http://www.w3.org/1999/XSL/Transform
  - Etc.

23

Document Type Definition (DTD)

- A *DTD* is a grammar specification for an XML document
- DTDs are optional – don't need to be specified
  - If specified, DTD can be part of the document (at the top); or it can be given as a URL
- A document that conforms (i.e., parses) w.r.t. its DTD is said to be *valid*
  - XML processors are not required to check validity, even if DTD is specified
  - But they are required to test well-formedness

24

DTDs (cont'd)

- DTD specified as part of a document:

```
<?xml  version="1.0" ?>
<!DOCTYPE  Report [
     …… …
]>
<Report> ………</Report>
```

- DTD specified as a standalone thing

```
<?xml  version="1.0" ?>
<!DOCTYPE  Report  "http://foo.org/Report.dtd">
<Report> ………</Report>
```

25

DTD Components

- <!ELEMENT *elt-name*
  (…*contents*…)/EMPTY/ANY >

Element's contents

- <!ATTLIST *elt-name attr-name*
  CDATA/ID/IDREF/IDREFS
  #IMPLIED/#REQUIRED
  >

An attr for elt

Type of attribute

Optional/mandatory

- Can define other things, like macros (called *entities* in the XML jargon)

26

DTD Example

```
<!DOCTYPE  Report [
    <!ELEMENT  Report  (Students, Classes, Courses)>
    <!ELEMENT  Students  (Student*)>
    <!ELEMENT  Classes  (Class*)>
    <!ELEMENT  Courses  (Course*)>
    <!ELEMENT  Student  (Name, Status, CrsTaken*)>
    <!ELEMENT  Name  (First,Last)>
    <!ELEMENT  First  (#PCDATA)>
    … … …
    <!ELEMENT  CrsTaken  EMPTY>
    <!ELEMENT  Class (CrsCode,Semester,ClassRoster)>
    <!ELEMENT  Course  (CrsName)>
    … … …
    <!ATTLIST  Report  Date  CDATA #IMPLIED>
    <!ATTLIST  Student  StudId  ID  #REQUIRED>
    <!ATTLIST  Course  CrsCode  ID  #REQUIRED>
    <!ATTLIST  CrsTaken  CrsCode  IDREF  #REQUIRED>
    <!ATTLIST  ClassRoster  Members  IDREFS  #IMPLIED>
]>
```

Zero or more

Has text content

Empty element, no content

Same attribute in different elements

27

Limitations of DTDs

- Doesn't understand namespaces
- Very limited assortment of data types (just strings)
- Very weak w.r.t. consistency constraints (ID/IDREF/IDREFS only)
- Can't express unordered contents conveniently
- All element names are global: can't have one Name type for people and another for companies:

  ```
  <!ELEMENT   Name   (Last, First)>
  <!ELEMENT   Name   (#PCDATA)>
  ```

  both can't be in the same DTD

28

XML Schema

- Came to rectify some of the problems with DTDs
- Advantages:
  - Integrated with namespaces
  - Many built-in types
  - User-defined types
  - Has local element names
  - Powerful key and referential constraints
- Disadvantages:
  - Unwieldy – much more complex than DTDs

29

Schema Document and Namespaces

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
          targetNamespace="http://xyz.edu/Admin">
     … … …
</schema>
```

- Uses standard XML syntax.
- http://www.w3.org/2001/XMLSchema – namespace for keywords used in a schema document (*not* an instance document), e.g., "*schema*", *targetNamespace*, etc.
- targetNamespace – names the namespace defined by the above schema.

30

Instance Document

- Report document whose structure is being defined by the earlier schema document

Default namespace for instance document

Namespace for XML Schema names that occur in instance documents rather than their schemas

```
<?xml  version = "1.0" ?>
<Report  xmlns="http://xyz.edu/Admin"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://xyz.edu/Admin
                                        http://xyz.edu/Admin.xsd" >
      …  same contents as in the earlier Report document …
</Report>
```

Schema namespace

Schema location

- xsi:schemaLocation says: the schema for the namespace http://xyz.edu/Admin is found in http://xyz.edu/Admin.xsd
- Document schema & its location are <u>not binding</u> on the XML processor; it can decide to use another schema, or none at all

31

Building Schemas from Components

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xyz.edu/Admin" >
    <include  schemaLocation="http://xyz.edu/StudentTypes.xsd">
    <include  schemaLocation="http://xyz.edu/ClassTypes.xsd">
    <include  schemaLocation="http://xyz.edu/CourseTypes.xsd">
   … …..
</schema>
```

- <include…> works like #include in the C language
  - Included schemas must have the same targetNamespace as the including schema
- schemaLocation – tells where to find the piece to be included
  - Note: this schemaLocation keyword is from the XMLSchema namespace – different from xsi:schemaLocation in previous slide, which was in XMLSchema-instance namespace

32

Simple Types

- *Primitive types*: *decimal*, *integer*, *Boolean*, *string*, ID, IDREF, etc. (defined in XMLSchema namespace)
- *Type constructors*: *list* and *union*
  - A simple way to derive types from primitive types (disregard the namespaces for now):

```
<simpleType  name="myIntList">
     <list  itemType="integer" />
</simpleType>

<simpleType  name="phoneNumber" >
    <union  memberTypes="phone7digits  phone10digits" />
</simpleType>
```

33

Deriving Simple Types by Restriction

```
<simpleType  name="phone7digits" >
    <restriction  base="integer" >
            <minInclusive  value="1000000" />
            <maxInclusive  value="9999999" />
      </restriction>
</simpleType>
<simpleType  name="emergencyNumbers" >
    <restriction  base="integer" >
            <enumeration  value="911" />
            <enumeration  value="333" />
    </restriction>
</simpleType>
```

- Has more type-building primitives (see textbook and specs)

34

Some Simple Types Used in the Report Document

```
<simpleType name="studentId">
        <restriction base="ID">
            <pattern value="s[0-9]{9}"/>
        </restriction>
</simpleType>
<simpleType name="studentIds">
        <list itemType="adm:studentRef"/>
</simpleType>

<simpleType name="studentRef">
        <restriction base="IDREF">
            <pattern value="s[0-9]{9}"/>
        </restriction>
</simpleType>
```

targetNamespace = http://xyz.edu/Admin
xmlns:adm= http://xyz.edu/Admin

XML ID types always start with a letter

Prefix for the target namespace

35

Simple Types for Report Document (contd.)

```
<simpleType name="courseCode">
        <restriction base="ID">
                <pattern value="[A-Z]{3}[0-9]{3}"/>
        </restriction>
</simpleType>
<simpleType name="courseRef">
        <restriction base="IDREF">
                <pattern value="[A-Z]{3}[0-9]{3}"/>
        </restriction>
</simpleType>
<simpleType name="studentStatus">
        <restriction base="string">
                <enumeration value="U1"/>
                .........
                <enumeration value="G5"/>
        </restriction>
</simpleType>
```

36

Schema Document That Defines Simple Types

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:adm="http://xyz.edu/Admin"
            targetNamespace="http://xyz.edu/Admin">
   … … …
   <element  name="CrsName"  type="string"/>
   <element  name="Status"  type=" adm:studentStatus" />
   … … …
   <simpleType  name="studentStatus" >
        … … …
   </simpleType>
</schema>
```

element declaration using derived type

Why is a namespace prefix needed here? (think)

37

Complex Types

- Allows the definition of element types that have complex internal structure
- Similar to class definitions in object-oriented databases
  - Very verbose syntax
  - Can define both child elements and attributes
  - Supports ordered and unordered collections of elements

38

Example: studentType

```
<element name="Student" type=" adm:studentType"/>
<complexType name="studentType">
    <sequence>
        <element name="Name" type=" adm:personNameType"/>
        <element name="Status" type=" adm:studentStatus"/>
        <element name="CrsTaken" type=" adm:courseTakenType"
                minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="StudId" type=" adm:studentId"/>
</complexType>

<complexType name="personNameType">
    <sequence>
        <element name="First" type="string"/>
        <element name="Last" type="string"/>
    </sequence>
</complexType>
```

39

Compositors: Sequences, Sets, Alternatives

- *Compositors*:
  - *sequence, all, choice* are required when element has at least 1 child element (= *complex content*)
- sequence -- have already seen
- all – can specify sets of elements
- choice – can specify alternative types

40

Sets

- Suppose the order of components in addresses is unimportant:

```
<complexType  name="addressType" >
        <all>
                <element  name="StreetName"  type="string" />
                <element  name="StreetNumber"  type="string" />
                <element  name="City"  type="string" />
        </all>
</complexType>
```

- *Problem*: all comes with a host of awkward restrictions. For instance, cannot occur inside a sequence; only sets of elements, not bags.

41

Alternative Types

- Assume addresses can have P.O.Box or street name/number:

```
<complexType  name="addressType">
      <sequence>
         <choice>
               <element  name="POBox"  type="string" />
               <sequence>
                     <element  name="Name"  type="string" />
                     <element  name="Number"  type="string" />
               </sequence>
         </choice>
         <element  name="City"  type="string" />
      </sequence>
</complexType>
```

This or that

42

Local Element Names

- A DTD can define only global element name:
  - Can have at most one <!ELEMENT foo …> statement per DTD
- In XML Schema, names have scope like in programming languages – the nearest containing complexType definition
  - Thus, can have the same element name (e.g., *Name)*, within different types and with different internal structures

43

Local Element Names: Example

```
<complexType name=“studentType”>
   <sequence>
        <element name=“Name” type=“adm: personNameType” />
        <element name=“Status” type=“adm: studentStatus” />
        <element name=“CrsTaken” type=“adm: courseTakenType”
                           minOccurs=“0” maxOccurs=“unbounded” />
   </sequence>
   <attribute name=“StudId” type=“adm: studentId” />
</complexType>
<complexType name=“courseType”>
   <sequence>
        <element name=“Name” type=“string” />
   </sequence>
   <attribute name=“CrsCode” type=“adm: courseCode” />
</complexType>
```

Same element name,
different types,
inside different complex types

44

Importing XML Schemas

- Import is used to share schemas developed by different groups at different sites
- Include vs. import:
  - *Include*:
    - Included schemas are usually under the control of the same development group as the including schema
    - Included and including schemas must have the same target namespace (because the text is physically included)
    - schemaLocation attribute required
  - *Import*:
    - Schemas are under the control of different groups
    - Target namespaces are different
    - The import statement must tell the importing schema what that target namespace is
    - schemaLocation attribute optional

45

Import of Schemas (cont'd)

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xyz.edu/Admin"
            xmlns:reg="http://xyz.edu/Registrar"
            xmlns:crs="http://xyz.edu/Courses" >
    <import  namespace="http://xyz.edu/Registrar"
              schemaLocation="http://xyz.edu/Registrar/StudentType.xsd" />
    <import  namespace="http://xyz.edu/Courses"  />
        … … …
        … … …
</schema>
```

Prefix declarations for imported namespaces

required

optional

46

Extension and Restriction of Base Types

- Mechanism for modifying the types in imported schemas
- Similar to subclassing in object-oriented languages
- *Extending* an XML Schema type means adding elements or adding attributes to existing elements
- *Restricting* types means tightening the types of the existing elements and attributes (i.e., replacing existing types with subtypes)

47

Type Extension: Example

```
<schema  xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:xyzCrs="http://xyz.edu/Courses"
         xmlns:fooAdm="http://foo.edu/Admin"
         targetNamespace="http://foo.edu/Admin">
    <import  namespace="http://xyz.edu/Courses"/>

    <complexType  name="courseType">                          Extends
        <complexContent>                                      by adding
            <extension  base="xyzCrs:courseType">
                <element  name="syllabus"  type="string"/>
            </extension>
        </complexContent>          Defined      Used
    </complexType>
    <element  name="Course"   type=" fooAdm:courseType"/>
    … ……
</schema>
```

48

Type Restriction: Example

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:xyzCrs="http://xyz.edu/Courses"
        xmlns:fooAdm="http://foo.edu/Admin"
        targetNamespace="http://foo.edu/Admin" >
   <import namespace="http://xyz.edu/Courses"/>

   <complexType name="studentType" >
      <complexContent>
         <restriction base="xyzCrs:studentType" >
            <sequence>
               <element name="Name" type=" xyzCrs:personNameType"/>
               <element name="Status" type=" xyzCrs:studentStatus"/>
               <element name="CrsTaken" type=" xyzCrs:courseTakenType"
                        minOccurs="0"  maxOccurs="60 "/>
            </sequence>
            <attribute name="StudId" type=" xyzCrs:studentId"/>
         </restriction>
      </complexContent>
   </complexType>
   <element name="Student" type=" fooAdm:studentType"/>
```

Must repeat the original definition

Tightened type: the original was "unbounded"

Structure of an XML Schema Document

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:adm="http://xyz.edu/Admin"
        targetNamespace="http://xyz.edu/Admin">
   <element name="Report" type="adm:reportType"/>

   <complexType name="reportType" >
      .........
   </complexType>
   <complexType name=...>
      .........
   </complexType>
   ... .....
</schema>
```

Root type

Root element

Definition of root type

Definition of types mentioned in the root type; Types can also be included or imported

Anonymous Types

- So far all types were ***named***
  - Useful when the same type is used in more than one place
- When a type definition is used exactly once, ***anonymous*** types can save space

"element" *used to be empty element – now isn't*

No type name

```
<element  name="Report">
   <complexType>
        <sequence>
           <element  name="Students"  type="adm:studentList" />
           <element  name="Classes"  type="adm:classOfferings" />
           <element  name="Courses"  type="adm:courseCatalog" />
        </sequence>
   </complexType>
</element>
```

51

Integrity Constraints in XML Schema

- A DTD can specify only very simple kinds of key and referential constraint; only using attributes
- XML Schema also has ID, IDREF as primitive data types, but these can also be used to type elements, not just attributes
- In addition, XML Schema can express complex key and foreign key constraints (shown next)

52

Schema Keys

- A *key* in an XML document is a sequence of components, which might include elements and attributes, which uniquely identifies document components in a *source collection* of objects in the document
- *Issues*:
  - Need to be able to identify that source collection
  - Need to be able to tell which sequences form the key
- For this, XML Schema uses *XPath* – a simple XML query language. (Much) more on XPath later

53

(Very) Basic XPath – for Key Specification

- Objects selected by the various XPath expressions are color coded

```
<Offerings>        –– current reference point
   <Offering>
        <CrsCode Section="1">CS532</CrsCode>
        <Semester><Term>Spring</Term><Year>2002</Year></Semester>
   </Offering>
   <Offering>
        <CrsCode Section="2">CS305</CrsCode>
        <Semester><Term>Fall</Term><Year>2002</Year></Semester>
   </Offering>
</Offerings>
```

Offering/CrsCode/@Section – *selects occurrences of attribute* Section *within* CrsCode *within* Offerings
Offering/CrsCode – *selects all* CrsCode *element occurrences within* Offerings
Offering/Semester/Term –*all* Term *elements within* Semester *within* Offerings
Offering/Semester/Year –*all* Year *elements within* Semester *within* Offerings

54

Keys: Example

```
<complexType name="reportType">
  <sequence>
    <element name="Students" …/>
    <element name="Classes" >
       <complexType>
         <sequence>
           <element name="Class" minOccurs="0" maxOccurs="unbounded"   >
              <sequence>
                 <element name="CrsCode" …/>
                 <element name="Semester"…/>
                 <element name="ClassRoster" …/>
              </sequence>
           </element>
         </sequence>
       </complexType>
       … … key specification goes here – next slide … …
    </element>
    <element name="Courses" …/>
  <sequence>
</complexType>
```

55

Example (cont’d)

- A key specification for the previous document:

```
<key name="PrimaryKeyForClass" >
     <selector xpath="Class"/>

     <field xpath="CrsCode"/>
     <field xpath="Semester"/>

</key>
```

Defines ***source collection*** *of objects to which the key applies. The XPath expression is relative to element to which the key is local*

field *must return exactly one value per object specified by* selector

Fields that form the key.
The XPath expression is relative to the source collection of objects specified in selector.
So, CrsCode *is actually* Classes/Class/CrsCode

56

Foreign Keys

- Like the REFERENCES clause in SQL, but more involved
- Need to specify:
  - *Foreign key*:
    - *Source collection* of objects
    - Fields that form the foreign key
  - *Target key*:
    - A previously defined *key* (or *unique*) specification, which is comprised of:
      - *Target collection* of objects
      - Sequence of fields that comprise the key

57

Foreign Key: Example

- Every class must have at least one student

```
<keyref  name="NoEmptyClasses"  refer="adm:PrimaryKeyForClass" >
      <selector  xpath="Student/CrsTaken" />
      <field  xpath="@CrsCode" />
      <field  xpath="@Semester" />
</keyref>
```

Source collection

Target key

Fields of the foreign key. XPath expressions are relative to the source collection

The above keyref declaration is part of element declaration for Students

58

XML Query Languages

- XPath – core query language. Very limited, a glorified selection operator. Very useful, though: used in XML Schema, XSLT, XQuery, many other XML standards
- XSLT – a functional style document transformation language. Very powerful, <u>very</u> complicated
- XQuery – W3C standard. Very powerful, fairly intuitive, SQL-style
- SQL/XML – attempt to marry SQL and XML, part of SQL:2003

59

Why Query XML?

- Need to extract parts of XML documents
- Need to transform documents into different forms
- Need to relate – join – parts of the same or different documents

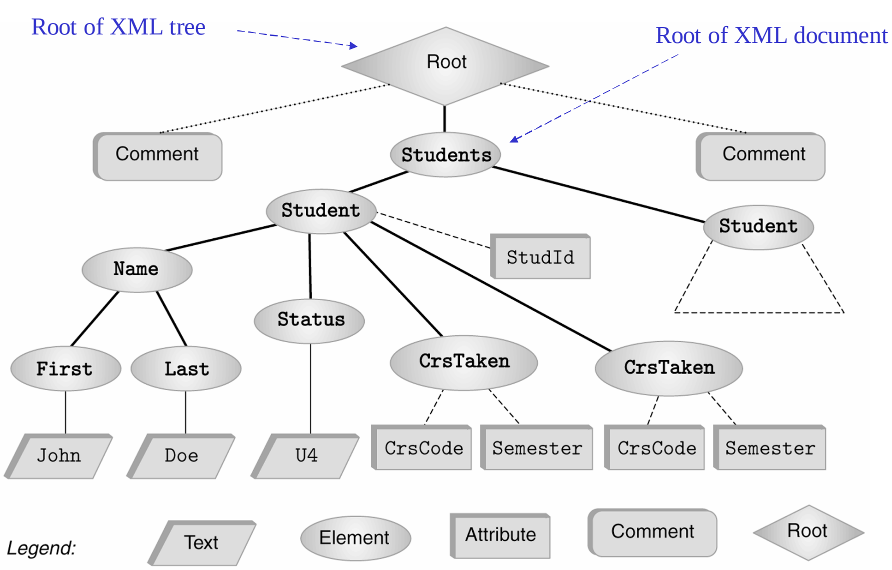
60

XPath

- Analogous to path expressions in object-oriented languages (e.g., OQL)
- Extends path expressions with query facility
- XPath views an XML document as a tree
  - Root of the tree is a *new* node, which doesn't correspond to anything in the document
  - Internal nodes are elements
  - Leaves are either
    - Attributes
    - Text nodes
    - Comments
    - Other things that we didn't discuss (processing instructions, …)

61

XPath Document Tree



62

Document Corresponding to the Tree

- A fragment of the report document that we used frequently

```
<?xml  version=“1.0” ?>
<!-- Some  comment -->
<Students>
    <Student StudId=“111111111” >
        <Name><First>John</First><Last>Doe</Last></Name>
        <Status>U2</Status>
        <CrsTaken  CrsCode=“CS308”  Semester=“F1997” />
        <CrsTaken  CrsCode=“MAT123”  Semester=“F1997” />
    </Student>
    <Student StudId=“987654321” >
        <Name><First>Bart</First><Last>Simpson</Last></Name>
        <Status>U4</Status>
        <CrsTaken  CrsCode=“CS308”  Semester=“F1994” />
    </Student>
</Students>
<!-- Some other comment -->
```

63

Terminology

- *Parent*/*child* nodes, as usual
- Child nodes (that are of interest to us) are of types *text*, *element*, *attribute*
  - We call them *t-children*, *e-children*, *a-children*
  - Also, *et-children* are child-nodes that are either elements or text, *ea-children* are child nodes that are either elements or attributes, etc.
- Ancestor/descendant nodes – as usual in trees

64

XPath Basics

- An XPath expression takes a document tree as input and returns a multi-set of nodes of the tree
- Expressions that *start* with / are ***absolute path expressions***
  - – Expression / – returns root node of XPath tree
  - – /Students/Student – returns all Student-elements that are children of Students elements, which in turn must be children of the root
  - – /Student – returns empty set (no such children at root)

65

XPath Basics (cont'd)

- ***Current*** (or ***context*** node) – exists during the evaluation of XPath expressions (and in other XML query languages)
- . – denotes the current node; .. – denotes the parent
  - foo/bar – returns all bar-elements that are children of foo nodes, which in turn are children of the current node
  - ./foo/bar – same
  - ../abc/cde – all cde e-children of abc e-children of the parent of the current node
- Expressions that don't start with / are ***relative*** (to the current node)

66

Attributes, Text, etc.

Denotes an attribute

- /Students/Student/@StudentId – returns all StudentId a-children of Student, which are e-children of Students, which are children of the root
- /Students/Student/Name/Last/text( ) – returns all t-children of Last e-children of …
- /comment( ) – returns comment nodes under root
- XPath provides means to select other document components as well

67

Overall Idea and Semantics

This is called *full* syntax.
We used *abbreviated* syntax before.
Full syntax is better for describing meaning. Abbreviated syntax is better for programming.

- An XPath expression is:
  locationStep1/locationStep2/…
- *Location step*:
  Axis::nodeSelector[predicate]
- Navigation *axis*:
  - *child, parent* – have seen
  - *ancestor, descendant, ancestor-or-self, descendant-or-self* – will see later
  - some other
- *Node selector*: node name or wildcard; e.g.,
  – ./child::Student (we used ./Student, which is an abbreviation)
  – ./child::* – any e-child (abbreviation: ./*)
- *Predicate*: a selection condition; e.g.,
  Students/Student[CourseTaken/@CrsCode = “CS532”]

68

XPath Semantics

- The meaning of the expression locationStep1/locationStep2/… is the set of all document nodes obtained as follows:
  - Find all nodes reachable by locationStep1 from the current node
  - For each node *N* in the result, find all nodes reachable from *N* by locationStep2; take the union of all these nodes
  - For each node in the result, find all nodes reachable by locationStep3, etc.
  - The value of the path expression on a document is the set of all document nodes found after processing the last location step in the expression

69

Overall Idea of the Semantics (Cont'd)

- locationStep1/locationStep2/… means:
  - Find all nodes specified by locationStep1
  - For each such node N:
    - Find all nodes specified by locationStep2 using N as the current node
    - Take union
  - For each node returned by locationStep2 do the same
- locationStep = axis::node[predicate]
  - Find all nodes specified by axis::node
  - Select only those that satisfy predicate

70

More on Navigation Primitives

- 2nd CrsTaken child of 1st Student child of Students:

  /Students/Student[1]/CrsTaken[2]
- All <u>last</u> CourseTaken elements within each Student element:

  /Students/Student/CrsTaken[last( )]

71

Wildcards

- Wildcards are useful when the exact structure of document is not known
- *Descendant-or-self* axis, // : allows to descend down any number of levels (including 0)
  - //CrsTaken – all CrsTaken nodes under the root
  - Students//@Name – all Name attribute nodes under the elements Students, who are children of the current node
  - *Note*:
    - ./Last and Last are same
    - .//Last and //Last are <u>different</u>
- The * wildcard:
  - * – any element: Student/*/text()
  - @* – any attribute: Students//@*

72

XPath Queries (selection predicates)

- Recall: Location step = Axis::nodeSelector[predicate]
- Predicate:
  - XPath expression = const | built-in function | XPath expression
  - XPath expression
  - built-in predicate
  - a Boolean combination thereof
- Axis::nodeSelector[predicate] $\subseteq$ Axis::nodeSelector but contains only the nodes that satisfy predicate
- Built-in predicate: special predicates for string matching, set manipulation, etc.
- Built-in function: large assortment of functions for string manipulation, aggregation, etc.

73

XPath Queries – Examples

- Students who have taken CS532:

  //Student[CrsTaken/@CrsCode="CS532"]

  True if : "CS532" $\in$ //Student/CrsTaken/@CrsCode

- Complex example:

  //Student[Status="U3" and starts-with(.//Last, "A")
  and contains(concat(.//@CrsCode), "ESE")
  and not(.//Last = .//First)]

- Aggregation: sum( ), count( )

  //Student[sum(.//@Grade) div count(.//@Grade) > 3.5]

74

Xpath Queries (cont'd)

- Testing whether a subnode exists:
  - //Student[CrsTaken/@Grade] – students who have a grade (for some course)
  - //Student[Name/First or CrsTaken/@Semester

    or Status/text() = "U4"] – students who have either a first name or have taken a course in some semester or have status U4
- Union operator, | :

  //CrsTaken[@Semester="F2001"] | //Class[Semester="F1990"]

  – union lets us define *heterogeneous* collections of nodes

75

XPointer

- XPointer = URL + XPath
  - A URL on steroids
- Syntax:

  url # xpointer (XPathExpr1) xpointer (XPathExpr2) …
  - Follow *url*
  - Compute XPathExpr1
    - Result non-empty? – return result
    - Else: compute XPathExpr2; and so on
- Example: you might click on a link and run a query against your Registrar's database

  http://yours.edu/Report.xml#xpointer(

  //Student[CrsTaken/@CrsCode="CS532"

  and CrsTaken/@Semester="S2002"])

76

XSLT: XML Transformation Language

- Powerful programming language, uses *functional programming paradigm*
- Originally designed as a stylesheet language: this is what “S”, “L”, and “T” stand for
  - The idea was to use it to display XML documents by transforming them into HTML
  - For this reason, XSLT programs are often called *stylesheets*
  - Their use is not limited to stylesheets – can be used to query XML documents, transform documents, etc.
- In wide use, but semantics is very complicated

XSLT Basics

- One way to apply an XSLT program to an XML document is to specify the program as a stylesheet in the document *preamble* using a *processing instruction*:

Processing instruction

Preamble

```
<?xml version=“1.0” ?>
… … …
<?xml-stylesheet type=“text/xsl”
                 href=“http://xyz.edu/Report/report.xsl” ?>
… … …
<Report Date=“2002-11-11”>
… … …
</Report>
```

Simple Example

- *Extract the list of all students* from **this (hyperlinked) document**

```
<?xml version="1.0" ?>
<StudentList xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
             xsl:version="1.0">
    <xsl:copy-of select="//Student/Name" />
</StudentList>
```

Standard XSLT namespace

Result document skeleton

XSLT instruction – copies the result of path expression to stdout

- Result:

```
<StudentList>
  <Name><First>John</First><Last>Doe</Last></Name>
  <Name><First>Bart</First><Last>Simpson</Last></Name>
</StudentList>
```

- Quiz: *Can we use the XSLT namespace as the default namespace in a stylesheet*? What problem might arise?

79

More Complex (Still Simple) Stylesheet

```
<StudentList xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
             xsl:version="1.0">
    <xsl:for-each select="//Student">
      <xsl:if test="count(CrsTaken) &gt; 1" >
             <FullName>
               <xsl:value-of select="*/Last" /> ,
               <xsl:value-of select="*/First" />
             </FullName>
      </xsl:if>
    </xsl:for-each>
</StudentList>
```

Extracts **contents** *of element, not the element itself (unlike copy-of)*

Result:

```
<StudentList>
  <FullName>
        Doe, John
  </FullName>
</StudentList>
```

80

XSLT Pattern-based Templates

- Where the real power lies
  - … and also where the peril lurks
- *Issue*: how to process XML documents by descending into their structure
- Previous syntax was just a shorthand for template syntax – next slide

81

Full Syntax vs. Simplified Syntax

- Simplified syntax:

```
<StudentList xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
             xsl:version="1.0">
    <xsl:for-each select="//Student">
       … …..
    </xsl:for-each>
</StudentList>
```

- Full syntax:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xsl:version="1.0">
    <xsl:template match="/" >
       <StudentList>
          <xsl:for-each select="//Student">
             … …..
          </xsl:for-each>
       </StudentList>
    </xsl:template>
</xsl:stylesheet>
```

82

Recursive Stylesheets

- A bunch of templates of the form:

  <xsl:template match="*XPath-expression*">

  … *tags, XSLT instructions* …

  </xsl:template>
- Template is applied to the node that is ***current*** in the evaluation process (will describe this process later)
- Template is used if its XPath expression is ***matched***:
  - "Matched" means: *current node* ∈ *result set of XPath expression*
  - If several templates match: use the *best matching template* – template with the <u>smallest</u> (by inclusion) XPath expression result set
  - If several of those: other rules apply (see XSLT specs)
  - If *no* template matches, use the matching *default* template
    - There is one default template for *et*-children and one for *a*-children – later

83

Recursive Traversal of Document

- <xsl:apply-templates/> – XSLT instruction that drives the recursive process of descending into the document tree
- Constructs the list of *et*-children of the current node
- For each node in the list, applies the best matching template
- A typical initial template:

```
<xsl:template match="/">
        <StudentList>
                <xsl:apply-templates />
        </StudentList>
</xsl:template>
```

Start with the root *node – typically the first template to be used in a stylesheet*

  - Outputs <StudentList> – </StudentList> tag pair
  - Applies templates to the *et*-children of the current node
  - Inserts whatever output is produced in-between <StudentList> and </StudentList>

84

Recursive Stylesheet Example

- As before: *list the names of students with* > 1 *courses*:

```
<?xml version="1.0" ?>
<xsl:stylesheet  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 xsl:version="1.0" >
   <xsl:template  match="/" >
        <StudentList>
                <xsl:apply-templates/>
        </StudentList>
   </xsl:template >
   <xsl:template  match="//Student" >
      <xsl:if   test="count(CrsTaken)  &gt;  1" >
        <FullName>
            <xsl:value-of   select="*/Last" />
            <xsl:value-of   select="*/First" />
        </FullName>
      </xsl if>
   </xsl:template>
   <xsl:template  match="text()" >
   </xsl:template>
</xsl:stylesheet>
```

Initial template

The workhorse, does all the job

Empty template – no-op. Needed to block default template for text – later.

85

Example Dissected

- *Initial template*: starts off, applies templates to *et*-children. The only *et*-child is *Students* element
- Stylesheet has no matching template for *Students*!
- Use *default template*: For *e*-nodes *or* root (/) the default is to go down to the *et*-children:

```
<xsl:template  match = "* | / " >
        <xsl:apply-templates />
</xsl:template>
```

- Children of *Students* node are two *Student* nodes – the "workhorse" template matches!
  - For each such (*Student*) node output:
    <FullName>*Last, First*</FullName>

86

Example (cont’d)

- Consider this *expanded* document :

```
<Report>
  <Students>
      <Student StudId=“111111111” >
             … … …
      </Student>
      <Student StudId=“987654321” >
             … … …
      </Student>
  </Students>
  <Courses>
      <Course  CrsCode=“CS308” >
          <CrsName>Software Engineering</CrsName>
       </Course>
      ………
  </Courses>
</Report>
```

Old part

New part

- Then the previous stylesheet has another branch to explore

Example (cont’d)

- No stylesheet template applies to *Courses*-element, so use the default template
- No explicit template applies to children, *Course*-elements – use the default again
- Nothing applies to *CrsName* – use the default
- The child of *CrsName* is a text node. If we used the default here: For text/attribute nodes the XSLT default is

```
<xsl:template  match=“text( ) | @*” >
        <xsl:value-of  select=“.” />
</xsl:template>
```

i.e., output the contents of text/attribute – we don’t want this!

This is why we provided the empty template for text nodes – to suppress the application of the default template

XSLT Evaluation Algorithm

- Very involved
- Not even properly defined in the official XSLT specification!
- More formally described in a research paper by Wadler – can only hope that vendors read this
- Will describe simplified version – will omit the *for-each* statement

89

XSLT Evaluation Algorithm (cont'd)

- Create root node, *OutRoot*, for the output document
- Copy root of the input document, *InRoot*, to output document: $InRoot^R$. Make $InRoot^R$ a child of *OutRoot*
    - Set current node variable: *CN* := *InRoot*
    - Set current node list: *CNL* := <*InRoot*>
  - *CN* : always the 1st node in *CNL*
  - When a node *N* is placed on *CNL*, its copy, N^R, goes to the output document (becomes a child of some node – see later)
    - N^R is a marker for where subsequent actions apply in the output document
    - Might be deleted or replaced later
- Find the *best matching template* for *CN* (or default template, if nothing applies)
- Apply this template to *CN* – next slide

90

XSLT Evaluation Algorithm – Application of a Template

- Application of template can cause these changes:

Case A: CN^R is replaced by a subtree

Example: *CN* = *Students* node in **our document**. Assume **our stylesheet** has the following template instead of the initial template (it thus becomes best-matching):

```
<xsl:template match="//Students" >
        <StudentList>
                <xsl:apply-templates />
        </StudentList>
</xsl:template>
```

Then:

- CN^R is replaced with *StudentList*
- Each child of *CN* (*Students* node) is copied over to the output tree as a child of *StudentList*

91

XSLT Evaluation Algorithm – Application of a Template (cont'd)

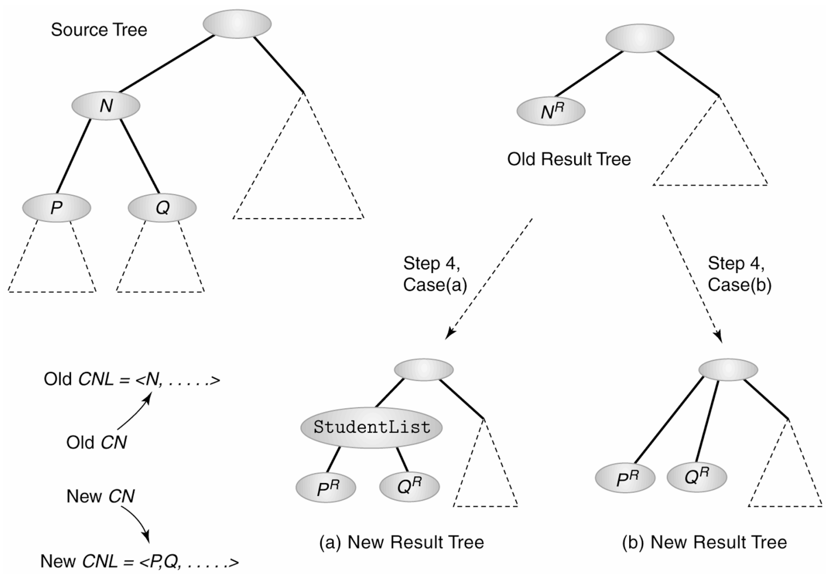
Case B: CN^R is deleted and its children become children of the parent of CN^R

Example: The default template, below, deletes CN^R when applied to any node:

```
<xsl:template match="* | /" >
        <xsl:apply-templates />
</xsl:template>
```

92

The Effect of **apply-templates** on Document Tree



XSLT Evaluation Algorithm (cont'd)

- In both cases (A & B):
  - If *CN* has no *et*-children, *CNL* becomes shorter
  - If it does have children, *CNL* is longer or stays the same length
  - The order in which *CN*'s children are placed on *CNL* is their order in the source tree
  - The new 1st node in *CNL* becomes the new *CN*
- Algorithm terminates when *CNL* is empty
  - Be careful – might not terminate (see next)

94

XSLT Evaluation Algorithm –Subtleties

- apply-templates instruction can have select attribute:
  - <xsl:apply-templates select="node()" /> – equivalent to the usual <xsl:apply-templates />
  - <xsl:apply-templates select="@* | text()" /> – instead of the *et*-children of *CN*, take *at*-children
  - <xsl:apply-templates select=".." /> – take the parent of *CN*
  - <xsl:apply-templates select="." /> – will cause an infinite loop!!
- Recipe to guarantee termination: make sure that *select* in apply-templates selects nodes only from a subtree of *CN*

95

Advanced Example

- *Example*: take any document and replace attributes with elements. So that

```
<Student StudId="111111111">
  <Name>John Doe</Name>
  <CrsTaken CrsCode="CS308" Semester="F1997" />
</Student>
```

would become:

```
<Student>
  <StudId>111111111</StudId>
  <Name>John Doe</Name>
  <CrsTaken>
    <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
  </CrsTaken>
</Student>
```

96

Advanced Example (cont'd)

- *Additional requirement*: don't rely on knowing the names of the attributes and elements in input document – should be completely general. Hence:

1. Need to be able to output elements whose name is not known in advance (we don't know which nodes we might be visiting)
   - Accomplished with xsl:element instruction and Xpath functions current( ) and name( ):

```
<xsl:element name="name(current())">
        Where am I?
</xsl:element>
```

If the current node is *foobar*, will output:

```
<foobar>
        Where am I?
</foobar>
```

97

Advanced Example (cont'd)

2. Need to be able to copy the current element over to the output document
   – The *copy-of* instruction won't do: it copies elements over with all their belongings. But remember: *we don't want attributes to remain attributes*
   – So, use the copy instruction
     – Copies the current node to the output document, but without any of its children

```
<xsl:copy>
   … XSLT instructions, which fill in the body
     of the element being copied over …
</xsl:copy>
```

98

Advanced Example (cont'd)

```
<xsl:stylesheet … … .>.
  <xsl:template  match="node()">
        <xsl:copy>
           <xsl:apply-templates  select="@*" />
           <xsl:apply-templates />
        </xsl:copy>
  <xsl:template>
   <xsl:template  match="@*">
        <xsl:element  name="name(current( ))" >
                 <xsl:value-of  select="." />
        </xsl:element>
  <xsl:template>
</xsl:stylesheet>
```

Process elements/text

Process a-children of current element

Process et-children of current element

Deal with attributes separately

Convert attribute to element

```
<…Attr="foo" >
```

becomes

```
<Attr>foo</Attr>
```

99

Limitations of XSLT as a Query Language

- Programming style unfamiliar to people trained on SQL
- Most importantly: Hard to do joins, i.e., *real* queries
  - Requires the use of variables (we didn't discuss)
  - Even harder than a simple nested loop (which one would use in this case in a language like C or Java)

100

XQuery – XML Query Language

- Integrates XPath with earlier proposed query languages: XQL, XML-QL
- SQL-style, not functional-style
- Much easier to use as a query language than XSLT
- Can do pretty much the same things as XSLT amd more, but typically easier
- 2004: XQuery 1.0

XQuery Basics

- General structure:

FOR *variable declarations*
WHERE *condition*
RETURN *document*

XQuery expression

- Example:

comment

```
(: students who took  MAT123  :)
FOR  $t  IN  doc("http://xyz.edu/transcript.xml")//Transcript
WHERE    $t/CrsTaken/@CrsCode = "MAT123"
RETURN  $t/Student
```

- Result:

```
<Student  StudId="111111111"  Name="John Doe" />
<Student  StudId="123454321"  Name="Joe Blow" />
```

This document on next slide

transcript.xml

```
<Transcripts>

  <Transcript>
      <Student StudId=“111111111” Name=“John Doe” />
      <CrsTaken CrsCode=“CS308” Semester=“F1997” Grade=“B” />
      <CrsTaken CrsCode=“MAT123” Semester=“F1997” Grade=“B” />
      <CrsTaken CrsCode=“EE101” Semester=“F1997” Grade=“A” />
      <CrsTaken CrsCode=“CS305” Semester=“F1995” Grade=“A” />
  </Transcript>

  <Transcript>
      <Student StudId=“987654321” Name=“Bart Simpson” />
      <CrsTaken CrsCode=“CS305” Semester=“F1995” Grade=“C” />
      <CrsTaken CrsCode=“CS308” Semester=“F1994” Grade=“B” />
  </Transcript>
```

… … *cont’d* ……

103

transcript.xml (cont’d)

```
  <Transcript>
      <Student StudId=“123454321” Name=“Joe Blow” />
      <CrsTaken CrsCode=“CS315” Semester=“S1997” Grade=“A” />
      <CrsTaken CrsCode=“CS305” Semester=“S1996” Grade=“A” />
      <CrsTaken CrsCode=“MAT123” Semester=“S1996” Grade=“C” />
  </Transcript>

  <Transcript>
      <Student StudId=“023456789” Name=“Homer Simpson” />
      <CrsTaken CrsCode=“EE101” Semester=“F1995” Grade=“B” />
      <CrsTaken CrsCode=“CS305” Semester=“S1996” Grade=“A” />
  </Transcript>

</Transcripts>
```

104

XQuery Basics (cont'd)

- Previous query doesn't produce a well-formed XML document; the following does:

```
<StudentList>
{
    FOR  $t IN  doc("transcript.xml")//Transcript
    WHERE  $t/CrsTaken/@CrsCode = "MAT123"
    RETURN  $t/Student
}
</StudentList>
```

Query inside XML

- FOR binds $t to Transcript elements one by one, filters using WHERE, then places Student-children as *e*-children of StudentList using RETURN

105

Document Restructuring with XQuery

- *Reconstruct lists of students taking each class using the* Transcript *records*:

```
FOR  $c  IN  distinct-values(doc("transcript.xml")//CrsTaken )
RETURN
   <ClassRoster   CrsCode = {$c/@CrsCode}   Semester = {$c/@Semester}>
   {
     FOR  $t  IN  doc("transcript.xml")//Transcript
     WHERE   $t/CrsTaken/[@CrsCode = $c/@CrsCode  and
                              @Semester = $c/@Semester]
     RETURN   $t/Student
                  ORDER BY  $t/Student/@StudId
   }
   </ClassRoster>
   ORDER BY  $c/@CrsCode
```

Query inside RETURN – similar to query inside SELECT in OQL

106

Document Restructuring (cont'd)

- *Output elements have the form*:

  <ClassRoster CrsCode="CS305" Semester="F1995">
  <Student StudId="111111111" Name="John Doe"/>
  <Student StudId="987654321" Name="Bart Simpson"/>
  </ClassRoster>

- *Problem*: the above element will be output *twice* – once when $c is bound to

  <CrsTaken CrsCode="CS305" Semester="F1995" Grade=" A"/>

  and once when it is bound to

  Bart Simpson's   John Doe's

  <CrsTaken CrsCode="CS305" Semester="F1995" Grade=" C"/>

  Note: grades are different – distinct-values( ) won't eliminate transcript records that refer to same class!

107

Document Restructuring (cont'd)

- *Solution*: instead of

  FOR $c IN distinct-values(doc("transcript.xml")//CrsTaken)

  use

  Document on next slide

  FOR $c IN doc("classes.xml")//Class

  where classes.xml lists course offerings (course code/semester) *explicitly* (no need to extract them from transcript records).

  Then $c is bound to each class exactly once, so each class roster will be output exactly once

108

http://xyz.edu/classes.xml

```
<Classes>
   <Class  CrsCode="CS308"  Semester="F1997">
        <CrsName>SE</CrsName> <Instructor>Adrian Jones</Instructor>
   </Class>
   <Class  CrsCode="EE101"  Semester="F1995">
        <CrsName>Circuits</CrsName> <Instructor>David Jones</Instructor>
   </Class>
   <Class  CrsCode="CS305"  Semester="F1995">
        <CrsName>Databases</CrsName> <Instructor>Mary Doe</Instructor>
   </Class>
   <Class  CrsCode="CS315"  Semester="S1997">
        <CrsName>TP</CrsName> <Instructor>John Smyth</Instructor>
   </Class>
   <Class  CrsCode="MAR123"  Semester="F1997">
        <CrsName>Algebra</CrsName> <Instructor>Ann White</Instructor>
   </Class>
</Classes>
```

109

Document Restructuring (cont'd)

- *More problems*: the above query will list classes with no students. Reformulation that avoids this:

Test that classes aren't empty

```
FOR $c IN doc("classes.xml")//Class
WHERE doc("transcripts.xml")
          //CrsTaken[@CrsCode = $c/@CrsCode
                    and @Semester = $c/@Semester]
RETURN
   <ClassRoster  CrsCode = {$c/@CrsCode}  Semester = {$c/@Semester}>
   {
     FOR $t IN doc("transcript.xml")//Transcript
     WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
                       @Semester = $c/@Semester]
     RETURN $t/Student  ORDER BY $t/Student/@StudId
   } </ClassRoster>
 ORDER BY $c/@CrsCode
```

110

XQuery Semantics

- So far the discussion was informal
- XQuery *semantics* defines what the expected result of a query is
- Defined analogously to the semantics of SQL

111

XQuery Semantics (cont'd)

- *Step 1*: Produce a list of bindings for variables
  - The FOR clause binds each variable to a *list* of nodes specified by an XQuery expression.

    The expression can be:
    - An XPath expression
    - An XQuery query
    - A function that returns a list of nodes
  - End result of a FOR clause:
    - Ordered list of tuples of document nodes
    - Each tuple is a binding for the variables in the FOR clause

112

XQuery Semantics (cont'd)

Example (bindings):

- Let FOR declare $A and $B
- Bind $A to document nodes {v,w}; $B to {x,y,z}
- Then FOR clause produces the following list of bindings for $A and $B:
  - $A/v, $B/x
  - $A/v, $B/y
  - $A/v, $B/z
  - $A/w, $B/x
  - $A/w, $B/y
  - $A/w, $B/z

113

XQuery Semantics (cont'd)

- *Step 2*: filter the bindings via the WHERE clause
  - Use each tuple binding to substitute its components for variables; retain those bindings that make WHERE true
  - Example: WHERE $A/CrsTaken/@CrsCode = $B/Class/@CrsCode
    - Binding: $A/w, where w = <CrsTaken CrsCode="CS308" …/>
$B/x, where x = <Class CrsCode="CS308" …/>
    - Then w/CrsTaken/@CrsCode = x/Class/@CrsCode, so the WHERE condition is satisfied & binding retained

114

XQuery Semantics (cont'd)

- *Step 3*: Construct result
  - For each retained tuple of bindings, instantiate the RETURN clause
  - This creates a fragment of the output document
  - Do this for each retained tuple of bindings in sequence

115

User-defined Functions

- Can define functions, even recursive ones
- Functions can be called from within an XQuery expression
- Body of function is an XQuery expression
- Result of expression is returned
  - Result can be a primitive data type (integer, string), an element, a list of elements, a list of arbitrary document nodes, …

116

XQuery Functions: Example

- Count the number of *e*-children recursively:

Function signature

```
DECLARE FUNCTION countNodes($e AS element()) AS integer {
   RETURN
     IF empty($e/*) THEN 0
     ELSE
        sum(FOR $n IN $e/* RETURN countNodes($n))
          + count($e/*)
}
```

XQuery expression

Built-in functions sum, count, empty

117

Class Rosters (again) Using Functions

```
DECLARE FUNCTION extractClasses($e AS element()) AS element()* {
   FOR $c IN $e//CrsTaken
   RETURN <Class CrsCode={$c/@CrsCode} Semester={$c/@Semester} />
}
<Rosters>
   FOR $c IN
      distinct-values(FOR $d IN doc("transcript.xml") RETURN extractClasses($d) )
   RETURN
      <ClassRoster  CrsCode = {$c/@CrsCode}  Semester = {$c/@Semester} >
      {
        LET $trs := doc("transcript.xml")
        FOR $t IN $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and
                                   CrsTaken/@Semester=$c/@Semester]
        RETURN $t/Student
        ORDER BY $t/Student/@StudId
      }
      </ClassRoster>
</Rosters>
```

118

Converting Attributes to Elements with XQuery

- An XQuery reformulation of a **previous XSLT query** – much more straightforward (but ignores text nodes)

```
DECLARE FUNCTION convertAttributes($a AS attribute()) AS element() {
    RETURN element {name($a)} {data($a)}
}
DECLARE FUNCTION convertElement($e AS node()) AS element() {
    RETURN element {name($e)}
        {
          { FOR $a IN $e/@* RETURN convertAttribute ($a) } ,
          IF empty($e/*) THEN $e/text( )
          ELSE { FOR $n IN $e/* RETURN convertElement($n) }
        }
}
RETURN convertElement(doc("my-document")/*)
```

Concatenate results

Computed element

*The actual query***:**
Just a RETURN statement!!

119

Integration with XML Schema and Namespaces

- Let type *sometype* be defined in http://types.r.us/types.xsd:

```
IMPORT SCHEMA namespace trs = "http://types.r.us"
                        AT "http://types.r.us/types.xsd";
DECLARE NAMESPACE foo = "http://foo.org/foo";
DECLARE FUNCTION local:doSomething($x AS trs:sometype)
                                   AS xs:string {
FOR $i IN fn:doc(…);
    … … …
RETURN
    <foo:something>
       …….
    </foo:something>
}
```

Namespace

Location

Predefined namespace

Another namespace

Built-in functions have predefined namespaces

Local functions have predefined namespace

120

Grouping and Aggregation

- Does not use separate grouping operator
  - Recall that OQL does not need one either
  - Subqueries inside the RETURN clause obviate this need (like subqueries inside SELECT did so in OQL)
- Uses built-in aggregate functions count, avg, sum, etc. (some borrowed from XPath)

121

Aggregation Example

- *Produce a list of students along with the number of courses each student took*:

```
FOR $t IN fn:doc("transcripts.xml")//Transcript,
        $s IN $t/Student
LET  $c := $t/CrsTaken
RETURN
      <StudentSummary  StudId = {$s/@StudId}  Name = {$s/@Name}
                            TotalCourses = {fn:count(fn:distinct-values($c))} />
ORDER BY  StudentSummary/@TotalCourses
```

- The *grouping effect* is achieved because $c is bound to a *new* set of nodes for *each* binding of $t

122

Quantification in XQuery

- XQuery supports explicit quantification: SOME (∃) and EVERY (∀)
- *Example*:

```
FOR $t IN fn:doc("transcript.xml")//Transcript
WHERE SOME $ct IN $t/CrsTaken
                SATISFIES $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

"Almost" equivalent to:

```
FOR $t IN fn:doc("transcript.xml")//Transcript,
        $ct IN $t/CrsTaken
WHERE $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

– *Not equivalent, if students can take same course twice*!

123

Implicit Quantification

- Note: in SQL, variables that occur in FROM, but not SELECT are implicitly quantified with ∃
- In XQuery, variables that occur in FOR, but not RETURN are similar to those in SQL. However:
  - In XQuery variables are bound to document nodes
    - Two nodes may look textually the same (e.g., two different instances of the same course element), but they are still different nodes and thus different variable bindings
    - Instantiations of the RETURN expression produced by binding variables to <u>different nodes</u> are output <u>even if these instantiations are textually identical</u>
  - In SQL a variable can be bound to the same value only once; identical tuples are not output twice (in theory)
  - *This is why the two queries in the previous slide are not equivalent*

124

Quantification (cont'd)

- Retrieve all classes (from classes.xml) where each student took MAT123
  - Hard to do in SQL (before SQL-99) because of the lack of explicit quantification

```
FOR $c IN fn:doc(classes.xml)//Class
LET $g := {          (: Transctipt records that correspond to class $c :)
   FOR $t IN fn:doc("transcript.xml")//Transcript
   WHERE $t/CrsTaken/@Semester = $c/@Semester
              AND $t/CrsTaken/@CrsCode = $c/@CrsCode
   RETURN $t
          }
WHERE EVERY $tr IN $g SATISFIES
                      NOT fn:empty($tr[CrsTaken/@CrsCode="MAT123])
RETURN $c ORDER BY $c/@CrsCode
```

125

SQL/XML – Extending Reach of SQL to XML Data

- In the past, SQL was extended for O-O:
  - added values for reference, tuple(row type), and collection(arrays), …
  - took over ODL and OQL standards of ODMG
- Currently, SQL is being extended for XML:
  - adding data types and functions to handle XML
  - will it bring the demise of XQuery?

126

Why SQL/XML

- Publish contents of SQL tables or entire DB as XML documents – need convention for translating primitive SQL data types
- Create XML documents out of SQL query results – need extension of SQL queries to create XML elements
- Store XML documents in relational DBs and query them – need extension of SQL to use XPath to access the elements of tree structures

127

Publishing Relations as XML Documents

- Current proposal:
  - no built-in functions to convert tables to XML
  - but can create arbitrary XML documents using extended SELECT statements
- Encoding relational data in XML:
  - Entire relation: an element named after the relation
  - Each row: an element named row
  - Each attribute: an element named after the attribute

128

Publishing Relations as XML Doc: Tables

Professor

Id	*Name*	*DeptId*
1024	Bob Smith	CS
3093	Amy Doe	EE
…		

```
<Professor>
  <row>
    <Id>1024</Id><Name>Bob Smith</Name><DeptId>CS<DeptId>
  </row>
  < row>
    <Id>3093</Id><Name>Amy Doe</Name><DeptId>EE<DeptId>
  </row >
  ...
</Professor>
```

129

Publishing Relations as XML Documents

SQL:

```
CREATE TABLE Professor
  Id: INTEGER,
  Name: CHAR(50),
  DeptId: CHAR(3)
```

XML Schema:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
           targeNamespace="http://xyz.edu/Admin">
  <element name="Professor">
    <complexType>
      <sequence>
        <element name="row" minOccurs="0" maxOccurs="unbounded">
          <complexType>
            <sequence>
              <element name="Id" type="integer"/>
              <element name="Name" type="CHAR_50"/>
              <element name="DeptId" type="CHAR_3"/>
            </sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

130

Publishing Relations as XML Doc: Schema

- CHAR_*len*: standard conventions in SQL/XML for CHAR(*len*) in SQL.
  - For instance, CHAR_50 is defined as

```
<simpleType>
  <restriction base="string">
    <length value="50">
  </restriction>
</simpleType>
```

- A lot of the SQL/XML standard deals with such data conversion, and with user-defined types of XML, which are defined in SQL using CREATE DOMAIN.

131

Creating XML from Queries: Functions

XMLELEMENT, XMLATTRIBUTES

- An SQL query does not return XML directly. Produces *tables* that can have columns of type **XML**.

```
SELECT  P.Id,  XMLELEMENT (
   NAME  "Prof",                                  -- element name
   XMLATTRIBUTES (P.DeptId AS "Dept"),            -- attributes
   P.Name                                         -- content
 ) AS  Info
FROM  Professor P
```

Produces tuples of the form

```
1024, <Prof Dept="CS">Bob Smith</Prof>
3093, <Prof Dept="EE">Amy Doe</Prof>
```

132

Creating XML Using Queries: Functions XMLELEMENT, XMLATTRIBUTES

- XMLELEMENT can be nested:

```
SELECT XMLELEMENT (NAME "Prof"
                        XMLELEMENT(NAME "Id", P.Id),
                        XMLELEMENT(NAME "Name", P.Name),
                        XMLELEMENT(NAME "DeptId", P.DeptId),
                       ) AS ProfElement
FROM   Professor  P
```

Produces tuples of the form

```
<Prof>
   <Id>1024</Id><Name>Bob Smith</Name><DeptId>CS</DeptId>
</Prof>
<Prof>
   <Id>3093</Id><Name>Amy Doe</Name><DeptId>EE</DeptId>
</Prof>
```

133

Creating XML Using Queries: Function XMLQUERY

```
SELECT XMLQUERY ('  <Prof>
                      <Id>{$I}</Id><Name>{$N}</Name><DeptId>{$D}</DeptId>
                     </Prof>'          -- template with placeholder variables
                 PASSING BY VALUE
                     P.Id AS I,          -- values of  I substitute for placehoders
                     P.Name AS N,
                     P.DeptId AS D
                 RETURNING SEQUENCE
                ) AS  ProfElement
FROM Professor P
```

Can be expressions

- Placeholder can occur in positions of XML elements and attributes
- Expressions can be XML-generating expressions or SELECT statements
  - In the example above, could have

```
SELECT QUERY(‘<Prof>
                 {$I} <Name>{$N}</Name> …
               </Prof>’
               PASSING BY VALUE XMLELEMENT(NAME "Id", P.Id)   AS   I
               …
```

- In general, the argument to XMLQUERY can include any XQuery expression (XPath or a full query)

134

Creating XML from Queries: Grouping without GROUP BY

- In XQuery: group elements as children of another element by putting a subquery in RETURN clause of parent query.
- In SQL/XML: group by putting SELECT inside XMLELEMENT in the SELECT clause of parent.
- Example: group the CrsTaken by student Ids

```
SELECT XMLELEMENT (
    NAME "Student",
    XMLATTRIBUTES(S.Id AS "Id"),
    (SELECT XMLELEMENT(NAME "CrsTaken",
                        XMLATTRIBUTES(T.CrsCode AS "CrsCode",
                                      T.Semester AS "Semester") )
     FROM Transcript T
     WHERE S.Id=T.StudId))
FROM Student S
```

Returns a set of 1-tuples, not list of elements. Waiting for the standard to resolve how to convert.

Creating XML from Queries: Grouping and XMLAGG

- Same example: group CrsTaken by student ids

```
SELECT XMLELEMENT (
  NAME "Student",
  XMLATTRIBUTES(S.Id AS "Id"),
  XMLAGG(XMLELEMENT(Name "CrsTaken",
                    XMLATTRIBUTES(T.CrsCode AS "CrsCode",
                                  T.Semester AS "Semester") )
         ORDER BY T.CrsCode) )
FROM Student S, Transcript T
WHERE S.Id = T.StudId
GROUP BY S.Id
```

Storing XML in Relational DB: Data Type XML

- Not stored as a string, but natively as a *tree structure*. Supports navigation via efficient storage and indexing.

```
CREATE TABLE StudentXML (
  Id        INTEGER,
  Details   XML )
```

where Details attribute contains things of the form

```
<Student>
  <Name><First>Amy</First><Last>Doe</Last></Name>
  <Status>U4</Status>
  <CrsTaken CrsCode="305" Semester="F2003"/>
  <CrsTaken CrsCode="336" Semester="F2003"/>
</Student>
```

137

Storing XML in Relational DB: Data Type XML

- To validate, use

```
CREATE TABLE StudentXML (
  Id          INTEGER,
  Details   XML,
  CHECK(Details IS VALID ACCORDING TO SCHEMA
                   ' http://xyz.edu/student.xsd'))
```

assuming the schema is stored at http://xyz.edu/student.xsd

138

Querying XML Stored in Relations: XMLEXISTS

- Tells whether the set of nodes returned by XPath expression is empty.
- Example: return Ids and names of students who have taken *a* course

```
SELECT   S.Id,   XMLEXTRACT(S.Details, '  //Name'  )
FROM     StudentXML  S
WHERE    XMLEXISTS(XMLQUERY(
                   '$D//CrsTaken'
                   PASSING BY REF  S.Details AS D
                   RETURNING SEQUENCE))
```

139

Querying XML Stored in Relations (cont'd)

- Use XQuery expressions and XMLEXIST function.
- XMLQUERY can be both in SELECT and WHERE clauses.
- Example: return Ids and names of students who have status U3 and took MAT123:

```
SELECT  S.Id,  XMLQUERY(S.Details, '$D//Name'
                  PASSING BY REF  S.Details AS D
                  RETURNING SEQUENCE)
FROM   StudentXML  S
WHERE XMLEXISTS(XMLQUERY(
           'WHERE  $D//Status/text( ) ="U3"   AND
                   $D//CrsTaken/@CrsCode = "MAT124"
            RETURN $D'
           PASSING BY REF  S.Details AS  D
           RETURNING SEQUENCE ))
```

140

Modifying Data in SQL/XML:
XMLPARSE

- XML stored as appropriately indexed tree structure, but in SQL is specified as a sequence of characters – so need to parse:

```
INSERT  INTO  StudentXML(Id, Details)
VALUES(12343,
            XMLPARSE(
            ' <Student>
               <Name><First>Bob</First><Last>Smith</Last></Name>
               <Status>U4</Status>
               <CrsTake CrsCode="CS305" Semester="F2003"/>
               <CrsTake CrsCode="CS339" Semester="S2004"/>
             </Student>'    ) )
```

141

Modifying Data in SQL/XML:
IS VALID ACCORDING TO SCHEMA

- To validate inserted document:

```
INSERT  INTO  StudentXML(Id, Details)
VALUES(12343,
            XMLPARSE(
            ' <Student>
               <Name><First>Bob</First><Last>Smith</Last></Name>
               <Status>U4</Status>
               <CrsTake CrsCode="CS305" Semester="F2003"/>
               <CrsTake CrsCode="CS339" Semester="S2004"/>
             </Student>‘ )
           IS VALID ACCORDING TO SCHEMA ‘http://xyz.edu/Students.xsd’
```

142

XMLSERIALIZE: Reverse of XMLPARSE

- To convert XML back to a string.
  - Typically used to talk to a host language that does not understand XML
- XMLSERIALIZE is often used in embedded SQL in conjunction with *cursors*
  - Example: return Ids and names of professors. Professors' names are returned as '<Prof>Joe</Prof>'.

```
EXEC SQL DECLARE  GetProfessor  CURSOR FOR
 SELECT  P.Id,  XMLSERIALIZE(XMLELEMENT(Name 'Prof',  P.Name))
 FROM  Professor  P
```

This can then be processed by

```
EXEC SQL  GetProfessor  INTO  :profId, :name
```

143