XML and Databases
Outline

• Semistructured data and XML – introduction
• DTD Schema
• XML Schema
• Xpath – core query language for XML
• XQuery – full-featured query language for XML
Semistructured Data

- A typical piece of data on the Web:
  
  <dt>Name: John Doe</dt>
  
  <dd>Id: 1111111111</dd>
  
  <dd>Address:  
    <ul>
      <li>Number: 123</li>
      <li>Street: Main</li>
    </ul>
  </dd>

  <dt>Name: Joe Public</dt>
  
  <dd>Id: 2222222222</dd>
  
  … … … … …
XML – 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)
XML vs. HTML

• Like HTML, but any number of different tags can be used (up to the document author)

• 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

• Unlike HTML, it is intolerant to bugs
  • Browsers will render buggy HTML pages
  • *XML processors* are not supposed to process buggy XML documents
Example

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

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

- Elements are nested
- Root element contains all others
More Terminology

John is a nice fellow

---

Opening tag

Closing tag: What is open must be closed

Parent of Address, Ancestor of number

Nested element, child of Person

Child of Address, Descendant of Person

“standalone” text, not useful as data

Content of Person:

<Person Name = “John” Id = “111111111”> 

John is a nice fellow

<Address>

<Number>21</Number>

<Street>Main St.</Street>

</Address>

… … …

</Person>
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 a value
  – The value must be quoted (with “ or ‘)
• XML processors are not supposed to try and fix ill-formed documents (unlike HTML browsers)
Identifying and Referencing with Attributes

• An attribute can be declared to have type
  • **ID** – unique identifier of an element
    – If attr1 & attr2 are both of type ID, then it is illegal to have
      \[ \langle \text{something} \text{ attr1=“abc”} \rangle \ldots \langle \text{somethingelse attr2=“abc”} \rangle \]
      within the same document
  • **IDREF** – references to unique element identifiers (in particular, an XML document with IDREFs is not a tree)
    – If attr1 has type ID and attr2 has type IDREF then we **can** have:
      \[ \langle \text{something} \text{ attr1=“abc”} \rangle \ldots \langle \text{somethingelse attr2=“abc”} \rangle \]
  • **IDREFS** – a list of references, if attr1 is ID and attr2 is IDREFS, then we **can** have
    – \[ \langle \text{something attr1=“abc”} \rangle \ldots \langle \text{somethingelse attr1=“cde”} \rangle \ldots \langle \text{someotherthing attr2=“abc cde”} \rangle \]
Example: Report Document with Cross-References

<?xml version="1.0" ?>
<Report Date="2002-12-12">
  <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="666666666">
      <Name><First>Joe</First><Last>Public</Last></Name>  
      <Status>U3</Status>
      <CrsTaken CrsCode="CS308" Semester="F1994" />
      <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>
</Report>

…… continued … …
<Classes>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
    <ClassRoster Members="666666666 987654321"/>
  </Class>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="111111111"/>
  </Class>
  <Class>
    <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="111111111 666666666"/>
  </Class>
</Classes>

…… continued … …
<Courses>
  <Course CrsCode = “CS308” >
    <CrsName>Market Analysis</CrsName>
  </Course>
  <Course CrsCode = “MAT123” >
    <CrsName>Market Analysis</CrsName>
  </Course>
</Courses>

</Report>
Outline

• Semistructured data and XML – introduction
• DTD Schema
• XML Schema
• Xpath – core query language for XML
• XQuery – full-featured query language for XML
Document Type Definition (DTD)

- A *DTD* is a grammar specification for an XML document
- DTDs are optional – don’t need to be specified
- A document that conforms (i.e., parses) w.r.t. its DTD is said to be *valid*
<!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 #IMPLIED>  
  <!ATTLIST Student StudId ID #REQUIRED>  
  <!ATTLIST Course CrsCode ID #REQUIRED>  
  <!ATTLIST CrsTaken CrsCode IDREF #REQUIRED>  
  <!ATTLIST ClassRoster Members IDREFS #IMPLIED>  
]>
Limitations of DTDs

- Doesn’t understand namespaces (discussed next)
  - All element names are global: can’t have one Name type for people and another for companies:
    ```xml
    <!ELEMENT Name (Last, First)>
    <!ELEMENT Name (#PCDATA)>
    ```
    both can’t be in the same DTD

- 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
Outline

• Semistructured data and XML – introduction
• DTD Schema
• XML Schema
• Xpath – core query language for XML
• XQuery – full-featured query language for XML
XML Schema

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

- **XML Namespaces, Include**
- **Simple Types**
  - Constructors: List, Union,
- **Complex Types**
  - Compositors: sequence, all, choice,
- **Extension and Restriction (of types)**
- **Schema Keys, Foreign Keys.**
XML Namespaces

• Namespaces = “Domains” for type names
• Essentially, a mechanism to prevent name clashes
• Namespace declaration
  • *Namespace* – a symbol, typically a URL
  • *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:

```xml
<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>
```
Namespaces (contd.)

- Scopes of declarations are color-coded:

```xml
<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
Namespaces (contd.)

• 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.,
  - `xsd` for http://www.w3.org/2001/XMLSchema
  - `xsl` for http://www.w3.org/1999/XSL/Transform
  - Etc.
Schema and Namespaces

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

- `http://www.w3.org/2001/XMLSchema` – namespace for keywords used in the official XML Schema specifications, e.g., "schema", `targetNamespace`, etc.
- `targetNamespace` – defines the namespace for the schema being defined by the above `<schema>…</schema>` document. Only makes sense as an attribute of ‘schema’ element.
Instance Document

• “Report” document whose structure is being defined by the earlier schema document

```xml
<?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>
```

• `xsi:schemaLocation` says: the schema for the namespace `http://xyz.edu/Admin` is found in `http://xyz.edu/Admin.xsd`
Building Schemas from Components

```xml
<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 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 is defined in the XMLSchema namespace – different from the earlier xsi:schemaLocation
XML Schema

• XML Namespaces, Include
• Simple Types
  – Constructors: List, Union,
• Complex Types
  – Compositors: sequence, all, choice,
• Extension and Restriction (of types)
• Schema Keys, Foreign Keys.
Simple Types

• **Primitive types:** decimal, integer, boolean, string, ID, IDREF, etc.

• **Type constructors:** *list* and *union*
  
  • A simple way to derive types from primitive types:

    ```xml
    <simpleType name="myIntList">
      <list itemType="integer" />
    </simpleType>
    
    <simpleType name="phoneNumber">
      <union memberTypes="phone7digits phone10digits" />
    </simpleType>
    ```
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 specs)
Some Simple Types Used in the Report Document

```xml
<simpleType name="studentId">
  <restriction base="ID">
    <pattern value="[0-9]{9}" />
  </restriction>
</simpleType>

<simpleType name="studentIds">
  <list itemType="studentRef" />
</simpleType>

<simpleType name="studentRef">
  <restriction base="IDREF">
    <pattern value="[0-9]{9}" />
  </restriction>
</simpleType>
```
Simple Types for Report Document (contd.)

```xml
<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>
```
Schema Using Simple Types

```xml
<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>
```

Why is a namespace prefix needed here? (think)
XML Schema

- XML Namespaces, Include
- Simple Types
  - Constructors: List, Union,
- Complex Types
  - Compositors: sequence, all, choice,
- Extension and Restriction (of types)
- Schema Keys, Foreign Keys.
Complex Types

• Allows to define element types that have complex internal structure
  – Very verbose syntax
  – Can define both child elements and attributes
  – Supports ordered and unordered collections of elements
Example: `studentType`

```xml
<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>
```
Compositors: Sequences, Bags, Alternatives

• **Compositors:**
  – *sequence, all, choice* are required when element has at least 1 child element (= *complex content*)
  – *sequence* -- have already seen
  – *all* – can express unordered sequences (bags)
  – *choice* – can express alternative types
Compositor “all” (Bags)

• Suppose the order of components in addresses is unimportant:

```xml
<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
Compositor “choice”

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

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

This or that
XML Schema

• XML Namespaces, Include
• Simple Types
  – Constructors: List, Union,
• Complex Types
  – Compositors: sequence, all, choice,
• Extension and Restriction (of types)
• Schema Keys, Foreign Keys.
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)
Type Extension: Example

```xml
<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">
        <complexContent>
            <extension base="xyzCrs:CourseType">
                <element name="syllabus" type="string"/>
            </extension>
        </complexContent>
    </complexType>

    <element name="Course" type="fooAdm:courseType"/>

    ...........
</schema>
```
Type Restriction: Example

```xml
<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">
        <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>
        <element name="Student" type="fooAdm:studentType" />
    </complexType>
</schema>
```

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>
</schema>
XML Schema

- XML Namespaces, Include
- Simple Types
  - Constructors: List, Union,
- Complex Types
  - Compositors: sequence, all, choice,
- Extension and Restriction (of types)
- Schema Keys, Foreign Keys.
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
(Very) Basic XPath – for Key Specification

• Objects selected by the various XPath expressions are color coded

<Offerings>  -- current reference point
<Offering>
  <CrsCode Section="1">CS52</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 + value within Offerings within CrsCode

Offering/CrsCode -- selects all CrsCode element occurrences within Offerings

Offering/Semester/Term -- all Term elements within Offerings within Semester

Offering/Semester/Year -- all Year elements within Offerings within Semester
<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>
        </element>
    </sequence>
</element>

<element name="Courses" … />
<sequence>
    … … key specification goes here – next slide … … 
</complexType>
Example (cont’d)

• A key specification for the previous document:

```xml
<key name="PrimaryKeyIdForClass" />
  <selector xpath="Classes/Class" />
  <field xpath="CrsCode" />
  <field xpath="Semester" />
</key>
```

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

Fields that form the key. The XPath expression is relative to the source collection of objects in select. So, CrsCode is like Classes/Class/CrsCode.
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
Foreign Key: Example

- Students take only offered courses.

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

Fields of the foreign key. The XPath expressions are relative to the source collection.
XML Query Languages

- **XPath** – core query language. Very limited, but very useful: used in XML Schema, XSLT, XQuery, many other XML standards
- **XQuery** – Very powerful, fairly intuitive, SQL-style
XPath

- 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
    - Elements that have no sub-elements or attributes
    - Attributes
    - Text nodes
    - Comments, etc.
XPath Basics

• Expression / – returns root node
• /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)
• Expressions that start with / are absolute path expressions
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)
XPath: Attributes, Text, etc.

- `/Students/Student/@StudentId` – returns all StudentId attribute-children of Student, which are element-children of Students, which are under root
- `/Students/Student/Name/Last/text()` – returns all text-children of Last element-children of …
- `/comment()` – returns comment nodes under root
- XPath provides means to select other document components as well
Overall Idea and Semantics

• An XPath expression is:
  locationStep1/locationStep2/…

• Location step:
  Axis::nodeSelector[predicate]

• Axis:
  • child, parent, ancestor, descendant, ancestor-or-self, descendant-or-self
    [Some of these have abbreviations, viz., .., //,]

• 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”]
Xpath Semantics

- **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
Xpath: Examples 1.

• 2\textsuperscript{nd} course taken by the first student in the list:
  /Students/Student[1]/CrsTaken[2]

• All \texttt{last} CourseTaken elements within each Student element:
  /Students/Student/CrsTaken[last()]
Xpath: Examples 2

- Students who have taken CS532:
  
  ```xml
  //Student[CrsTaken/@CrsCode="CS532"]
  True if: “CS532” ∈ //Student/CrsTaken/@CrsCode
  ```

- Complex example:
  
  ```xml
  //Student[Status=“U3” and starts-with(./Last, “A”) 
  and contains(concat(./@CrsCode), “ESE”) 
  and not(./Last = ./First) ]
  ```

- Aggregation: sum( ), count( )
  
  ```xml
  //Student[sum(./@Grade) div count(./@Grade) > 3.5]
  ```
Xpath: Examples 3.

• 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
XQuery – XML Query Language

• Integrates XPath with earlier proposed query languages: XQL, XML-QL
• SQL-style, not functional-style
XQuery Basics

• General structure:
  FOR variable declarations
  WHERE condition
  RETURN document

• Example:
  FOR $t$ IN document("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 … …
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>
XQuery Basics (cont’d)

• To produce a well-formed XML document:

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

• FOR binds $t to Transcript elements one by one, filters using WHERE, then places Student-children as e-children of StudentList using RETURN
Document Restructuring with XQuery

- Student list for all classes:

  FOR $c$ IN distinct(document("transcript.xml")/Transcript/CrsTaken)
  RETURN
  <ClassRoster CrsCode = "${c}[@CrsCode]" Semester = "${c}[@Semester]">
    {
      FOR $t$ IN document("transcript.xml")/Transcript
        WHERE $t$/CrsTaken[@CrsCode = $c$/@CrsCode AND @Semester = $c$/@Semester]
      RETURN $t$/Student
        SORTBY ($t$/Student/@StudId)
    }
  </ClassRoster>
  SORTBY ($c$/@CrsCode)
Document Restructuring (cont’d)

• Output elements have the form:

```xml
<ClassRoster>
  <Student StudId="111111111" Name="John Doe" />
  <Student StudId="987654321" Name="Bart Simpson" />
</ClassRoster>
```

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

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

and once when it is bound to

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

*Note: since grades are different – distinct() doesn’t help.*
Document Restructuring (cont’d)

• **Solution**: instead of

  ```
  FOR $c$ IN distinct(document(“transcript.xml”)/Transcript/CrsTaken)
  use

  FOR $c$ IN document(“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. [See later for an alternate solution.]
http://xyz.edu/classes.xml

<Class>
    <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>
Document Restructuring (cont’d)

• To avoid listing classes with no students:

FOR $c$ IN document("classes.xml")/Class
WHERE document("transcripts.xml")
  //CrsTaken[@CrsCode = $c/@CrsCode
  and  @Semester = $c/@Semester
RETURN
  <ClassRoster  CrsCode = {$c/@CrsCode}  Semester = {$c/@Semester}>}
  }
FOR $t$ IN document("transcript.xml")/Transcript
WHERE  $t/CrsTaken/@CrsCode = $c/CrsCode
  AND $t/CrsTaken/@Semester = $c/@Semester
RETURN  $t/Student   SORTBY ($t/Student/@StudId)
} </ClassRoster>
SORTBY ($c/@CrsCode)
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
XQuery Semantics (cont’d)

• **Step 1**: Produce a list of bindings for variables
  – The FOR clause binds each variable to an *ordered* 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
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 $B$:
  
  - $A/v, B/x$
  - $A/v, B/y$
  - $A/v, B/z$
  - $A/w, B/x$
  - $A/w, B/y$
  - $A/w, B/z$
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
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
Xquery – Remaining Slides

- Filter()
- User-defined functions
- Aggregation/Grouping
- Quantification
The filter() Operator

- **filter**(*document, set-of-doc-nodes*)
  - *set-of-doc-nodes* specified using Xpath
  - Delete every node that *does not* occur in *set-of-doc-nodes* from *document*
    - *set-of-doc-nodes* is treated literally as a set of nodes; children not included
  - Connect disconnected children to appropriate ancestor

*Red nodes are the ones not in set-of-doc-nodes*
The filter() operator (cont’d)

• *Example*: filtering http://xyz.edu/classes.html
  
  filter(//Class,  //Class | //Class/CrsName)

  yields:

  <Class><CrsName>SE</CrsName></Class>
  <Class><CrsName>Circuits</CrsName></Class>
  <Class><CrsName>Databases</CrsName></Class>
  <Class><CrsName>TP</CrsName></Class>
  <Class><CrsName>Algebra</CrsName></Class>

  *Deleted nodes:*
  
  /Classes/Class/@CrsCode,  /Classes/Class/@Semester,  /Classes/Class/Instructor
Removing Duplicate Courses Using filter()

\[
\text{LET } \$ct := \text{document("transcript.xml")/Transcript/CrsTaken}
\]

\[
\text{FOR } \$c \text{ IN distinct(filter($ct, $ct | $ct/\text{@CrsCode} | $ct/\text{@Semester}))}
\]

\[
\text{RETURN } \langle \text{ClassRoster} \text{ CrsCode} = \{$c/\text{@CrsCode}\} \text{ Semester} = \{$c/\text{@Semester}\} >
\]

\[
\{ \\
\text{FOR } \$t \text{ IN document("transcript.xml")/Transcript} \\
\text{WHERE } \$t/CrsTaken/\text{@CrsCode} = \$c/CrsCode \\
\text{AND } \$t/CrsTaken/\text{@Semester} = \$c/\text{@Semester} \\
\text{RETURN } \$t/\text{Student} \\
\text{SORTBY } \(\$t/\text{Student/\text{@StudId}}\)
\}
\]

\[
</\text{ClassRoster}>
\]

\[
\text{SORTBY } \(\$c/\text{@CrsCode}\)
\]

Declare variable, a syntactic sugar

Get rid of @Grade and thus of duplicate CrsTaken elements
User-defined Functions

• Can define functions, even recursive ones
• Functions can be called from within a FLWR 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, …
XQuery Functions: Example

• Count the number of $e$-children recursively:

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

- **Function signature**
- **XQuery expression**
- **Built-in functions** sum, count, empty
Roster Example (again) Using Functions

DEFINE FUNCTION extractClasses(element $e) RETURNS element* {
    FOR $c IN $e//CrsTaken
    RETURN <Class CrsCode={$c/@CrsCode} Semester={$c/@Semester} />
}

<Rosters>
    FOR $c IN distinct(FOR $d IN document("transcript.xml") RETURN extractClasses($d))
    RETURN <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester} >
    {
        LET $trs := document("transcript.xml")
        FOR $t IN $trs//Transcript[CrsTaken/@CrsCode=$c/@CrsCode and CrsTaken/@Semester=$c/@Semester]
        RETURN $t/Student
        SORTBY ($t/Student/@StudId)
    }
</ClassRoster>
</Rosters>
Grouping and Aggregation

- No grouping operator
- Uses built-in aggregate functions count, avg, sum, etc. (some borrowed from XPath)
Aggregation Example

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

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

• The grouping effect is achieved because $c is bound to a new set of nodes for each binding of $t
Quantification in XQuery

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

Example:

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

Almost equivalent to:

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

– Not equivalent, if students can take same course twice!
Quantification (cont’d)

- Retrieve all classes (from classes.xml) where each student took MAT123

FOR $c$ IN document(classes.xml)//Class
LET $g := \{ \text{-- Transcript records that correspond to class } c \}$
FOR $t$ IN document("transcript.xml")//Transcript
WHERE $t$/CrsTaken/@Semester = $c$/@Semester
AND $t$/CrsTaken/@CrsCode = $c$/CrsCode
RETURN $t$
}
WHERE EVERY $tr$ IN $g$ SATISFIES
NOT empty($tr$/CrsTaken/@CrsCode="MAT123")
RETURN $c$ SORTBY($c$/@CrsCode)