XML and Databases

Chapter 17

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

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 specifically designed to work with XML documents rather than relational or object-oriented data
• XML is closely related to object-oriented and so-called semistructured data

Semistructured Data

• A typical piece of data on the Web:
  
  <dt>
  <dd> Name: John Doe
  <dd> Id: 111111111
  <dd> Address: <ul>
  <li> Number: 123
  <li> Street: Main
  </ul>
  </dt>
  
  <dt>
  <dd> Name: Joe Public
  <dd> Id: 222222222
  … … …
  </dt>

Semistructured Data (contd.)

• To make the previous student list suitable for machine consumption on the Web, it should have these characteristics:
  • Be object-like
  • Be schemaless (doesn’t guarantee 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)

What is Self-describing Data?

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

  Data part:
  
  ("Students", ["John", 111111111, [123, "Main St"]],
  ["Joe", 222222222, [321, "Pine St"]])

  Schema part:
  
  PersonList | ListName: String,
  Contents: [ Name: String,
  Id: String,
  Address: [Number: Integer, Street: String] ]
  }

Mark up
What is Self-Describing Data?  
(cont'd.)

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

```
(ListName: “Students”,
Contents: [ [ Name: “John Doe”,
   Id: “111111111”,
   Address: [Number: 123, Street: “Main St.”] ],
 [Name: “Joe Public”,
   Id: “222222222”,
   Address: [Number: 321, Street: “Pine St.”] ]]
)
```

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
    - Selfdescribing
    - 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 much more complex!) way of specifying schema
  - Query and transformation languages:
    - XPath
    - XSLT
    - XQuery

Overview of XML

- 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
  - Some semantics can be specified using XML Schema (structure), some using stylesheets (rendering)
- Unlike HTML, is intolerant to bugs
  - Browsers will render buggy HTML pages
  - XML processors are not supposed to process buggy XML documents

```
<?xml version=“1.0” ?>
<PersonList Type=“Student” Date=“2002-02-02” >
  <Title Value=“Student List” /> 
  <Person>…  …  …</Person>
  <Person>…  …  …</Person>
</PersonList>
```

Example

- Elements are nested
- Root element contains all others

More Terminology

- Opening tag
- Closing tag: What is open must be closed

Conversion from XML to Objects

- **Straightforward**
  ```
  <Person Name=“Joe”>
    <Age>44</Age>
    <Address><Number>22</Number><Street>Main</Street>
  </Person>
  ```
  ```
  (#545, [Name: “Joe”,
    Age: 44,
    Address: [Number: 22, Street: “Main”]
  ])
  ```
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>Joe</Name>
  <Age>44</Age>
  <Number>22</Number>
  <Street>Main</Street>
</Address>
</Person>
```

Differences between XML Documents and Objects

- XML’s origin is document processing, not databases
  - Allows things like standalone text (useless for databases)
  - Attributes aren’t needed – just bloat the number of ways to represent the same thing
  - 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]]`

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>` is a no-no
- An attribute name can occur at most once in an opening tag. It it occurs,
  - It must have a 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)

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 `<something attr1="abc">… <somethingelse attr2="abc"/>` 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: `<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"/>

Example: Report Document with Cross-References

```
<?xml version="1.0" ?>
<Report Date="2002-12-12">
  <Students>
    <Student StudId="111111111"/>
    <Student StudId="666666666"/>
    <Student StudId="987654321"/>
  </Students>
  <Classes>
    <Class>
      <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
    </Class>
    <Class>
      <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
    </Class>
  </Classes>
</Report>
```

Report Document (contd.)

```
<Classes>
  <Class>
    <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
    <ClassRoster Members="666666666, 987654321"/>
  </Class>
  <Class>
    <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
    <ClassRoster Members="111111111, 666666666"/>
  </Class>
</Classes>
```
XML Namespaces

- A mechanism to prevent name clashes between components of same or different documents
- 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>
  ```

- Default name space
  - Reserved keyword

- New default; over shadows old declaration

- Namespaces (contd.)

- Scopes of declarations are color-coded:

  ```xml
  <item xmlns="http://www.foo.org/abc"
        xmlns:cd:="http://www.bar.com/cde">
    <name>¼ </name>
    <feature>
      <cd: item>
        <cd:name>¼ </cd:name>
      </cd: item>
    </feature>
  </item>
  ```

- 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.

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

- XML Namespaces

- DTD specified as part of a document:

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

- DTD specified as a standalone thing

  ```xml
  <?xml version="1.0" ?>
  <Report> ... ... </Report>
  ```
DTD Components

- `<!ELEMENT elt-name (…contents…)>`
- `<!ATTLIST elt-name attr-name ID/IDREF/IDREFS EMPTY/#IMPLIED/#REQUIRED>`
- Can define other things, like macros (called *entities*)

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

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

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>`...

Instance Document

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

```xml
<Report xmlns="http://xyz.edu/Admin">
  ... same contents as in the earlier Report document ...
</Report>
```

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

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:
    ```
    <simpleType name="myIntList">
      <list itemType="integer" />
    </simpleType>
    <simpleType name="phoneNumber">
      <union memberTypes="phone7digits phone10digits" />
    </simpleType>
    ```

Some Simple Types Used in the Report Document

```
<element name="CrsName" type="string" />
<element name="Status" type="adm:studentStatus" />
```

Simple Types for Report Document (contd.)

```
<element name="CrsCode" type="adm:studentStatus" />
<element name="CrsRef" type="string" />
<element name="studentStatus" type="adm:studentStatus" />
```

Instance Document That Uses 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" />
</schema>
```

Why is a namespace prefix needed here? (Think)
Complex Types

- Allows to define 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

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>
```

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

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

Alternative Types

- Assume addresses can have P.O.Box or street name/number:
  ```xml
  <complexType name="addressType">
    <choice>
      <element name="POBox" type="string"/>
      <sequence>
        <element name="Name" type="string"/>
        <element name="Number" type="string"/>
      </sequence>
    </choice>
    <element name="City" type="string"/>
  </complexType>
  ```

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, say Name, within different types and with different internal structures
Local Element Names: Example

```xml
<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>
```

Importing XML Schemas

- Import is used to share schemas developed by different groups at different sites
  - 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)
  - Import:
    - Schemas are under the control of different groups
    - Target namespaces are different
    - The import statement must tell the including schema what that target namespace is

Import of Schemas (cont’d)

```xml
<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>
```

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:fooAdmin="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>
</schema>
```

Type Restriction: Example

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

  <complexType name="courseType">
    <complexContent>
      <restriction base="xyzCrs:courseType">
        <sequence>
          <element name="Name" type="xyzCrs:personNameType" />
          <element name="Status" type="xyzCrs:studentStatus" />
          <element name="CrsTaken" type="xyzCrs:courseTakenType" minOccurs="0" maxOccurs="60" />
        </sequence>
      </restriction>
    </complexContent>
  </complexType>
</schema>
```
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">
        <sequence>
            <element name="Students" type="adm:studentList" />
            <element name="Classes" type="adm:classOfferings" />
            <element name="Courses" type="adm:courseCatalog" />
        </sequence>
    </complexType>
</schema>

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

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

(Very) Basic XPath – for Key Specification
- Objects selected by the various XPath expressions are color coded
- Objects selected by the various XPath expressions are color coded

(Offerings) -- current reference point
    <Offering>
        <CrsCode.Section>12</CrsCode.Section>-
        <CrsCode>CS532</CrsCode>-
        <CrsCode>Spring</CrsCode>-
        <Year>2002</Year>
    </Offering>
    <Offering>
        <CrsCode.Section>12</CrsCode.Section>-
        <CrsCode>CS305</CrsCode>-
        <CrsCode>Fall</CrsCode>-
        <Year>2002</Year>
    </Offering>

Keys: Example
<complexType name="reportType">
    <sequence>
        <element name="Students" />
        <element name="Classes" />
    </sequence>
    <element name="Courses" />
</complexType>

Keys: Example
<complexType name="reportType">
    <sequence>
        <element name="Students" />
        <element name="Classes" />
    </sequence>
    <element name="Courses" />
</complexType>

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

Keys: Example
<complexType name="reportType">
    <sequence>
        <element name="Students" />
        <element name="Classes" />
    </sequence>
    <element name="Courses" />
</complexType>
Example (cont’d)

• A key specification for the previous document:
  <key name="PrimaryKeyForClass">
    <selector xpath="Classes/Class" />
    <field xpath="CrsCode" />
    <field xpath="Semester" />
  </key>

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

• Every class must have at least one student
  <keyref name="NoEmptyClasses" refer="adm:PrimaryKeyForClass">
    <selector xpath="Students/Student/CrsTaken" />
    <field name="@CrsCode" />
    <field name="@Semester" />
  </keyref>

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, very complicated
• XQuery – upcoming standard. Very powerful, fairly intuitive, SQL-style

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

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
    • Elements that have no subelements or attributes
    • Attributes
    • Text nodes
    • Comments
    • Other things that we didn’t discuss (processing instructions, …)
**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

**Attributes, Text, etc.**

- /Students/Student/@StudentId – returns all StudentId a-children of Student, which are e-children of Students, which are under 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

**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

**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/def – all def 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)
Overall Idea and Semantics

- 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"]

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

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

More on Navigation Primitives

- 2nd course taken by the first student in the list:
  /Students/Student[1]/CrsTaken[2]
- All last CourseTaken elements within each Student element:
  /Students/Student/CrsTaken[last()]

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 different
- The * wildcard:
  - * — any element: Student/*[text()]
  - @* — any attribute: Students/@*

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] ⊆ 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.
**XPath Queries – Examples**

- Students who have taken CS532:
  
  ```xml
  //Student[CrsTaken/@CrsCode='CS532']
  ```

- 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 Queries (cont’d)**

- Testing whether a subnode exists:
  ```xml
  //Student[CrsTaken/@Grade]  ± students who have a grade (for some course)
  ```

- Complex example:
  ```xml
  //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, `|`:
  ```xml
  //CrsTaken[@Semester='F2001'] | //Class[Semester='F1990']  ± union lets us define heterogeneous collections of nodes
  ```

**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

```xml
http://yours.edu/Report.xml
```

```
//Student[CrsTaken/@CrsCode='CS532' and CrsTaken/@Semester='S2002']
```

**XSL: 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:

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

**Simple Example**

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

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

- *Result:*

```xml
<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?*
More Complex (Still Simple) Stylesheet

```xml
<StudentList xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xml: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>
```

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

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

Full Syntax vs. Simplified Syntax

- Simplified syntax:
  ```xml
  <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:
  ```xml
  <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()">
      <!-- empty template needed to block default template for text -->
    </xsl:template>
  </xsl:stylesheet>
  ```

Recursive Stylesheets

- A bunch of templates of the form:
  ```xml
  <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 smallest 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 e-t-children and one for a-children — later

Recursive Stylesheet Example

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

```xml
<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()">
    <!-- empty template needed to block default template for text -->
  </xsl:template>
</xsl:stylesheet>
```
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:
  ```xml
  <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:
    ```xml
    <FullName>Last, First</FullName>
    ```

Example (cont’d)

- Consider this expanded document:
  ```xml
  <Report>
    <Students>
      <Student StudId="111111111" />
      ...
      <Student StudId="987654321" />
    </Students>
    ...
    <Courses>
      <Course CrsCode="CS308"/>
      <CrsName>Software Engineering</CrsName>
    </Course>
    ...
  </Report>
  ```
- 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
  
  For text/attribute nodes the XSLT default is
  ```xml
  <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

XSLT Evaluation Algorithm (cont’d)

- Application of template can cause these changes:
  
  **Case A**: CN is replaced by a subtree
  ```xml
  Example: CN = Students node in our document. Assume our stylesheet has the following template instead of the initial template (it thus becomes best-matching):
  ```
  ```xml
  <xsl:template match="/Students">
    <StudentList>
      <xsl:apply-templates />
    </StudentList>
  </xsl:template>
  ```
  Then:
  ```xml
  - CN is replaced with StudentList
  - Each child of CN (Students node) is copied over to the output tree as a child of StudentList
  ```
XSLT Evaluation Algorithm – Application of a Template (cont’d)

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

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

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

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)

The Effect of apply-templates on Document Tree

XSLT Evaluation Algorithm – Subtleties

- apply-templates instruction can have select attribute:

```xml
<xsl:apply-templates select="node()"/>
```

- ```<xsl:apply-templates select="@* | text()"/>```  – instead of the et-children of CN, take at-children

```xml
<xsl:apply-templates select="."/>
```

- ```<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

Advanced Example

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

  ```xml
  <Student StudId="11111111">
    <Name>John Doe</Name>
    <CrsTaken CrsCode="CS308" Semester="F1997"/>
  </Student>
  ```

  would become:
  ```xml
  <Student>
    <StudId>11111111</StudId>
    <Name>John Doe</Name>
    <CrsTaken CrsCode="CS308">Semester="F1997"</CrsTaken>
    <Student/>
  </Student>
  ```

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

       <xsl:element>
       If the current node is foo, bar, will output:
       ```xml
       <foo>bar</foo>
       ```
       Where am I?

       ```xml
       </foo>bar
       ```
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

```xml
<xs:copy>
    ... XSLT instructions, which fill in the body of the element being copied over ...
</xs:copy>
```

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)

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, but typically easier
- XQuery 1.0 standard late in 2002

XQuery Basics

```
<Transcripts>
    <Transcript>
        <Student StudId="111111111" Name="John Doe"/>
        <CrsTaken CrsCode="CS308" Semester="F1997" Grade="B"/>
        <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="A"/>
    </Transcript>
    <Transcript>
        <Student StudId="987654321" Name="Bart Simpson"/>
        <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B"/>
    </Transcript>
</Transcripts>
```
Document Restructuring with XQuery

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

```
FOR $c IN distinct(document("transcript.xml")/CrsTaken)
RETURN
  <ClassRoster CrsCode={$/CrsTaken/@CrsCode} Semester={$/CrsTaken/@Semester}>
  FOR $t IN document("transcript.xml")/Transcript
  WHERE $t/CrsTaken/@CrsCode = $c/CrsTaken/@CrsCode
  AND $t/CrsTaken/@Semester = $c/@Semester
  RETURN $t/Student
SORTBY ($t/Student/@StudId)
</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
  `<CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>

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

Solution: instead of

```
FOR $c IN distinct(document("transcript.xml")/CrsTaken)
RETURN
  <ClassRoster CrsCode={$/CrsTaken/@CrsCode} Semester={$/CrsTaken/@Semester}>
    FOR $t IN document("classes.xml")/CrsTaken
      WHERE $t/@CrsCode = $c/@CrsCode
      RETURN $t/Student
    SORTBY ($t/Student/@StudId)
  </ClassRoster>
```

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.

XQuery Basics (cont’d)

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

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

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

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

```
FOR $c$ IN document("classes.xml")/CrsTaken
  WHERE document("transcripts.xml")//CrsTaken[@CrsCode = $c/@CrsCode]
    and $c/@Semester = $c/@Semester
RETURN <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester}>
```

For classes that aren’t empty:

```
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)
```

So far the discussion was informal

XQuery semantics defines what the expected result of a query is

Defined analogously to the semantics of SQL

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 $B$:
  - $A/v$, $B/x$
  - $A/v$, $B/y$
  - $A/v$, $B/z$
  - $A/w$, $B/x$
  - $A/w$, $B/y$
  - $A/w$, $B/z$

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" /></CrsTaken>
    - $B/x$, where $x$ = <Class CrsCode="CS308" /></Class>
    - Then $w$/CrsTaken/@CrsCode = $x$/Class/@CrsCode, so the WHERE condition is satisfied & binding retained

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
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

Example: filtering http://xyz.edu/classes.html

```
filter(/Class, /Class | /Class/CrsName)
```

`filter()` operator (cont’d)

- `Filter(document, set-of-doc-nodes)`
  - set-of-doc-nodes is treated literally as a set of nodes; children not included
  - Connect disconnected children to appropriate ancestor

Fixing the Restructuring Query Using filter()

```
LET $c := document(‘transcript.xml’) / Transcript / CrsTaken
FOR $c IN distinct(filter($c, $c/@CrsCode | $c/@Semester))
RETURN <ClassRoster CrsCode = {$c/@CrsCode} Semester = {$c/@Semester}>
  { FOR $t IN document(‘transcript.xml’) / Transcript / CrsTaken/@CrsCode = $c/@CrsCode AND $t/@Semester = $c/@Semester
    RETURN $t/Student SORTBY ($t/Student/@StudId)
  }<ClassRoster>
SORTBY ($c/@CrsCode)
```

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:

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

Class Rosters (again) Using Functions

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

```
FOR $c IN distinct FOR $d IN document(‘transcript.xml’) RETURN extractClasses($d)
RETURN <ClassRoster Courier= {$c/@CrsCode} Semester = {$c/@Semester}>
  { LET $trs := document(‘transcript.xml’) FOR $t IN $trs / Transcript / CrsTaken/@CrsCode = $c/@CrsCode AND $t/@Semester = $c/@Semester
    RETURN $t/Student SORTBY ($t/Student/@StudId)
  }<ClassRoster>
</Rosters>
```
Converting Attributes to Elements with XQuery

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

```xquery
DEFINE FUNCTION convertAttributes( $a ) RETURNS element {
    RETURN element { name( $a ) } { value( $a ) }
}

FUNCTION convertElement( $e ) RETURNS AnyElement {
    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( document( "my-document" ) )
```

Integration with XML Schema and Namespaces

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

```xml
SCHEMA "http://types.r.us" at http://types.r.us/types.xsd
NAMESPACE trs = "http://types.r.us"
NAMESPACE xsd = "http://www.w3.org/2001/XMLSchema"
FUNCTION doSomething( trs:FOO $x )
    RETURNS xsd:string {
        ...
    }
```

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)

Aggregation Example

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

```xquery
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 ) ) } />
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:

```xquery
FOR $t IN document( "transcripts.xml" )//Transcript
    WHERE SOME $c In $t/CrsTaken Satisfies $c/@CrsCode = "MAT123"
RETURN $t/Student
```

Almost equivalent to:

```xquery
FOR $t IN document( "transcripts.xml" )//Transcript,
    $c In $t/CrsTaken
WHERE $c/@CrsCode = "MAT123"
RETURN $t/Student
```

- Not equivalent, if students can take same course twice!

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 different nodes are output even if these instantiations are textually identical.
  - 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
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 Sc IN document(classes.xml)//Class
LET g = (  -- Transcript records that correspond to class Sc
FOR st IN document("transcript.xml")//Transcript
WHERE st/CrsTaken/@Semester = Sc/@Semester
AND st/CrsTaken/@CrsCode = Sc/CrsCode
RETURN st )
WHERE EVERY st IN g SATISFIES
    NOT empty(st[CrsTaken/@CrsCode="MAT123"])
RETURN Sc SORTBY(Sc/@CrsCode)